This chapter covers

- Exploring Spring’s core modules
- Decoupling application objects
- Managing cross-cutting concerns with AOP
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 were 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 little resemblance to the original JavaBeans specification.

Despite the fact that many successful applications have been built based on EJB, EJB never achieved its intended purpose: to simplify enterprise application development. It is true that EJB’s declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. However, in a different way, EJBs complicate development 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.

Today, Java component development has returned to its roots. New programming techniques, including aspect-oriented programming (AOP) and dependency injection (DI), are giving JavaBeans much of the power previously reserved for EJBs. These techniques furnish plain-old Java objects (POJOs) 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.

In all fairness, even EJBs have evolved to promote a POJO-based programming model. Employing ideas such as DI and AOP, the latest EJB specification is significantly simpler than its predecessors. For many developers, though, this move is too little, too late. By the time the EJB 3 specification had entered the scene, other POJO-based development frameworks had already established themselves as de facto standards in the Java community.

Leading the charge for lightweight POJO-based development is the Spring Framework, which we’ll be exploring throughout this book. In this chapter, we’re going to explore the Spring Framework at a high level, giving you a taste of what Spring is all about. This chapter will give you a good idea of the types of problems Spring solves and will set the stage for the rest of the book. First things first—let’s find out what Spring is.

1.1 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. 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, I’ll use the word “bean” when referring to conventional JavaBeans and “EJB” when referring to Enterprise JavaBeans. I’ll also throw around the term “POJO” (plain-old Java object) from time to time.

Spring does many things, but when you strip it down to its base parts, Spring is a lightweight dependency injection and aspect-oriented container and framework. That’s quite a mouthful, but it nicely summarizes Spring’s core purpose. To make more sense of Spring, let’s break this description down:

- **Lightweight**—Spring is lightweight in terms of both size and overhead. The bulk of the Spring Framework can be distributed in a single JAR file that weighs in at just over 2.5 MB. And the processing overhead required by Spring is negligible. What’s more, Spring is nonintrusive: objects in a
Spring-enabled application often have no dependencies on Spring-specific classes.

- **Dependency Injection**—Spring promotes loose coupling through a technique known as dependency injection (DI). When DI is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. You can think of DI 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.

- **Aspect-oriented**—Spring comes with rich support for aspect-oriented programming (AOP) 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 lifecycle and configuration of application objects. In Spring, you can declare how each of your application objects should be created, how they should be configured, and how they should be associated with each other.

- **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.

To restate: When you strip Spring down to its base parts, what you get is a framework that helps you develop loosely coupled application code. Even if that were all that Spring did, the benefits of loose coupling (maintainability and testability) would make Spring a worthwhile framework to build applications on.

But Spring is more. The Spring Framework comes with several modules that build on the foundation of DI and AOP to create a feature-filled platform on which to build applications.

### 1.1.1 Spring modules

The Spring Framework is made up of several well-defined modules (see figure 1.1). When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But you don’t have to base your
What is Spring?

application fully on the Spring Framework. You are free to choose the modules that suit your application and look to other options when Spring doesn’t fit the bill. In fact, Spring offers integration points with several other frameworks and libraries so you won’t have to write them yourself.

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.

Let’s take a look at each of Spring’s modules in figure 1.1, one at a time, to see how each fits into the overall Spring picture.

**The core container**
At the very base of figure 1.1, you’ll find Spring’s core container. Spring’s core container provides the fundamental functionality of the Spring Framework. This module contains the BeanFactory, which is the fundamental Spring container and the basis on which Spring’s DI is based.
CHAPTER 1

Springing into action

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

**Application context module**

Spring’s application context builds on the core container. 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 lifecycle events, and validation.

In addition, this module supplies many enterprise services such as email, 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. Like DI, AOP supports loose coupling of application objects. With AOP, however, applicationwide concerns (such as transactions and security) are decoupled from the objects to which they are applied.

Spring’s AOP module offers several approaches to building aspects, including building aspects based on AOP Alliance interfaces (http://aopalliance.sf.net) and support for AspectJ. We’ll dig into Spring’s AOP support in chapter 4.

**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.

We’ll see how Spring’s template-based JDBC abstraction can greatly simplify JDBC code when we look at Spring data access in chapter 5.
Object-relational mapping (ORM) integration module
For those who prefer using an object-relational mapping (ORM) tool over straight JDBC, Spring provides the ORM module. Spring’s ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions. Spring doesn’t attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps. Spring’s transaction management supports each of these ORM frameworks as well as JDBC.

In addition to Spring’s template-based JDBC abstraction, we’ll look at how Spring provides a similar abstraction for ORM and persistence frameworks in chapter 5.

Java Management Extensions (JMX)
Exposing the inner workings of a Java application for management is a critical part of making an application production ready. Spring’s JMX module makes it easy to expose your application’s beans as JMX MBeans. This makes it possible to monitor and reconfigure a running application.

We’ll take a look at Spring’s support for JMX in chapter 12.

Java EE Connector API (JCA)
The enterprise application landscape is littered with a mishmash of applications running on an array of disparate servers and platforms. Integrating these applications can be tricky. The Java EE Connection API (better known as JCA) provides a standard way of integrating Java applications with a variety of enterprise information systems, including mainframes and databases.

In many ways, JCA is much like JDBC, except where JDBC is focused on database access, JCA is a more general-purpose API connecting to legacy systems. Spring’s support for JCA is similar to its JDBC support, abstracting away JCA’s boilerplate code into templates.

The Spring MVC framework
The Model/View/Controller (MVC) paradigm is a commonly accepted approach to building web applications such that the user interface is separate from the application logic. Java has no shortage of MVC frameworks, with Apache Struts, JSF, WebWork, and Tapestry among the most popular MVC choices.

Even though Spring integrates with several popular MVC frameworks, it also comes with its own very capable MVC framework that promotes Spring’s loosely coupled techniques in the web layer of an application.

We’ll dig into Spring MVC in chapters 13 and 14.
CHAPTER 1
Springing into action

Spring Portlet MVC
Many web applications are page based—that is, each request to the application results in a completely new page being displayed. Each page typically presents a specific piece of information or prompts the user with a specific form. In contrast, portlet-based applications aggregate several bits of functionality on a single web page. This provides a view into several applications at once.

If you’re building portlet-enabled applications, you’ll certainly want to look at Spring’s Portlet MVC framework. Spring Portlet MVC builds on Spring MVC to provide a set of controllers that support Java’s portlet API.

Spring’s web module
Spring MVC and Spring Portlet MVC require special consideration when loading the Spring application context. Therefore, Spring’s web module provides special support classes for Spring MVC and Spring Portlet MVC.

The web module also contains support for several web-oriented tasks, such as multipart file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Apache Struts and Java Server Faces (JSF).

Remoting
Not all applications work alone. Oftentimes, it’s necessary for an application to leverage the functionality of another application to get its work done. When the other application is accessed over the network, some form of remoting is used for communication.

Spring’s remoting support enables you to expose the functionality of your Java objects as remote objects. Or if you need to access objects remotely, the remoting module also makes simple work of wiring remote objects into your application as if they were local POJOs. Several remoting options are available, including Remote Method Invocation (RMI), Hessian, Burlap, JAX-RPC, and Spring’s own HTTP Invoker.

In chapter 8, we’ll explore the various remoting options supported in Spring.

Java Message Service (JMS)
The downside to remoting is that it depends on network reliability and that both ends of the communication be available. Message-oriented communication, on the other hand, is more reliable and guarantees delivery of messages, even if the network and endpoints are unreliable.

Spring’s Java Message Service (JMS) module helps you send messages to JMS message queues and topics. At the same time, this module also helps you create
message-driven POJOs that are capable of consuming asynchronous messages. We’ll see how to use Spring to send messages in chapter 10.

Although Spring covers a lot of ground, it’s important to realize that Spring avoids reinventing the wheel whenever possible. Spring leans heavily on existing APIs and frameworks. For example, as we’ll see later in chapter 5, Spring doesn’t implement its own persistence framework—instead, it fosters integration with several capable persistence frameworks, including simple JDBC, iBATIS, Hibernate, and JPA.

Now that you’ve seen the big picture, let’s see how Spring’s DI and AOP features work. We’ll get our feet wet by wiring our first bean into the Spring container.

### 1.2 A Spring jump start

Dependency injection is the most basic thing that Spring does. But what does DI look like? In the grand tradition of programming books, I’ll start by showing you how Spring works with the proverbial “Hello World” example. Unlike the original Hello World program, however, this example will be modified a bit to demonstrate the basics of Spring.

The first class that the “Springified” Hello World example needs is a service class whose purpose is to print the familiar greeting. Listing 1.1 shows the GreetingService interface, which defines the contract for the service class.

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

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

GreetingServiceImpl (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.

```java
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(this.greeting);
    }
}
```
The GreetingServiceImpl class has a single property: greeting. This property is simply a String that holds the message that will be printed when the sayGreeting() method is called. You may have noticed that 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.

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. As shown here, the <property> element tells
the Spring container to call `setGreeting()`, passing in “Buenos Dias!” (for a bit of Spanish flair) when instantiating the bean.

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:

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

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

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

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

```java
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.

```
package com.springinaction.chapter01.hello;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class HelloApp {
    public static void main(String[] args) throws Exception {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("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
CHAPTER 1

Springing into action

reference in hand, it finally calls the sayGreeting() method. When you run the
Hello application, it prints (not surprisingly)

Buenos Dias!

This is about as simple a Spring-enabled application as I can come up with. Despite its simplicity, however, 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 DI.

1.3 Understanding dependency injection

Although Spring does a lot of things, DI 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, DI is not nearly as complex as it sounds. In fact, by applying DI in your projects, you’ll find that your code will become significantly simpler, easier to understand, and easier to test.

But what does “dependency injection” mean?

1.3.1 Injecting dependencies

Originally, dependency injection was commonly referred to by another name:
inversion of control. But 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
dependencies that is being inverted. Based on that revelation, he coined the
phrase “dependency injection,” a term that better describes what is going on.

Any nontrivial application (pretty much anything more complex than Hel-
loWorld.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). This can lead to highly coupled
and hard-to-test code.

When applying DI, objects are given their dependencies at creation time by some external
entity that coordinates each object in the system. In other words, dependencies are injected into
objects. So, DI means an inversion of responsibility with regard to how an object obtains refer-
ences to collaborating objects (see figure 1.2).
The key benefit of DI is loose coupling. If an object only knows about its dependencies by their interface (not their implementation or how they were instantiated) then the dependency can be swapped out with a different implementation without the depending object knowing the difference.

For example, if the `Foo` class in figure 1.2 only knows about its `Bar` dependency through an interface then the actual implementation of `Bar` is of no importance to `Foo`. `Bar` could be a local POJO, a remote web service, an EJB, or a mock implementation for a unit test—`Foo` doesn’t need to know or care.

If you’re like me, you’re probably anxious to see how this works in code. I aim to please, so without further delay...

### 1.3.2 Dependency injection in action

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 who 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  A Knight of the Round Table bean

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

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

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

    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.

```
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. Not only does it ensure 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.

```
package com.springinaction.chapter01.knight;
import junit.framework.TestCase;
public class KnightOfTheRoundTableTest extends TestCase {
    public void testEmbarkOnQuest() throws GrailNotFoundException {
        KnightOfTheRoundTable knight =
            new KnightOfTheRoundTable("Bedivere");
        HolyGrail grail = knight.embarkOnQuest();
        assertNotNull(grail);
        assertTrue(grail.isHoly());
    }
}
```
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 whom?**
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 1.3). Therefore, you have no way to test the knight class in isolation. As it 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 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:

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

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

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

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

public class HolyGrailQuest implements Quest {
    public HolyGrailQuest() {} 
    public Object embark() throws QuestFailedException {
        // Do whatever it means to embark on a quest 
        return new HolyGrail();
    }
}
```

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

```java
private Quest quest;

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

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

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

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:

```java
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;
```
public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();
}

public Object embarkOnQuest() throws QuestFailedException {
    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 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:

```java
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;
    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }
    public Object embarkOnQuest() throws QuestFailedException {
        return quest.embark();
    }
    public void setQuest(Quest quest) {
        this.quest = quest;
    }
}
```

Notice the difference? Compare figure 1.4 with figure 1.3 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 one configuration, you might give it a HolyGrailQuest. In a different configuration, maybe a different Quest implementation, such as RescueDamselQuest, will be given to the knight. Similarly, in a test case you would give it a mock implementation of Quest.
In a nutshell, that is what DI is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves.

**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?

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`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/
➥ spring-beans-2.0.xsd">
  <bean id="quest" class="com.springinaction.chapter01.knight.HolyGrailQuest"/>
  <bean id="knight" class="com.springinaction.chapter01.knight.KnightOfTheRoundTable">
    <constructor-arg value="Bedivere"/>
    <property name="quest" ref="quest"/>
  </bean>
</beans>
```

Listing 1.8 Wiring a quest into a knight in the Spring configuration XML

![Diagram](image)

The knight is no longer responsible for getting its own quest. Instead, it is given (injected with) a quest through its `setQuest()` method.
This is just a simple approach to wiring beans. Don’t worry too much about the details 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.

```java
Listing 1.9 Running the knight example
package com.springinaction.chapter01.knight;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io FileSystemResource;

public class KnightApp {
    public static void main(String[] args) throws Exception {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("knight.xml"));
        Knight knight = (Knight) factory.getBean("knight");
        knight.embarkOnQuest();
    }
}
```

Once the application has a reference to the `Knight` 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 dependency injection, but now let’s see how you can use DI in your real-world enterprise applications.

### 1.3.3 Dependency injection in enterprise applications

Suppose that you’ve been tasked with writing an online shopping application. Included in the application is an order service component to handle all functions related to placing an order. Figure 1.5 illustrates several ways that a web layer
Checkout component (which, perhaps, could be a WebWork action or a Tapestry page) might access the order service.

A simple, but naive, approach would be to directly instantiate the order service when it’s needed 1. Aside from directly coupling the web layer to a specific service class, this approach will result in wasteful creation of the OrderServiceImpl class, when a shared, stateless singleton will suffice.

If the order service is implemented as a 2.x EJB, you would access the service by first retrieving the home interface through JNDI 2, which would then be used to access an implementation of the EJB’s service interface. In this case, the web layer is no longer coupled to a specific interface, but it is coupled to JNDI and to the EJB 2.x programming model.

As an EJB 3 bean, the order service could be looked up from JNDI directly 3 (without going through a home interface). Again, there’s no coupling to a specific implementation class, but there is a dependence on JNDI.

With or without EJB, you might choose to hide the lookup details behind a service locator 4. This would address the coupling concerns seen with the other approaches, but now the web layer is coupled to the service locator.

The key issue with all of these approaches is that the web layer component is too involved in obtaining its own dependencies. It knows too much about where the order service comes from and how it’s implemented.
If knowing too much about your dependencies leads to tightly coupled code, it stands to reason that knowing as little as possible about your dependencies leads to loosely coupled code. Consider figure 1.6, which shows how the Checkout component could be given an OrderService instead of asking for one. Now let’s see how this would be implemented using DI:

```java
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 (which is an interface) is given to the class through the setOrderService() method. The web component does not know or care where the OrderService comes from. It could be injected by Spring or it could be manually injected by an explicit call to setOrderService(). It also has no idea as to how OrderService is implemented—it only knows about it through the OrderService interface. With DI, your application objects are freed from the burden of fetching their own dependencies and are able to focus on their tasks, trusting that their dependencies will be available when needed.

Dependency injection is a boon to loosely coupled code, making it possible to keep your application objects at arm’s length from each other. However, we’ve only scratched the surface of the Spring container and DI. In chapters 2 and 3, you’ll see more ways to wire objects in the Spring container.

Dependency injection is only one technique that Spring offers to POJOs in support of loose coupling. Aspect-oriented programming provides a different kind of decoupling power by separating application-spanning functionality (such as security and transactions) from the objects they affect. Let’s take a quick look at Spring’s support for AOP.
1.4 Applying aspect-oriented programming

Although DI 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.

1.4.1 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 1.7 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.

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. In short, aspects ensure that POJOs remain plain.
Applying aspect-oriented programming

It may help to think of aspects as blankets that cover many components of an application, as illustrated in figure 1.8. At its core, an application consists of modules that implement the business functionality. With AOP, you can then cover your core application with layers of functionality. These layers can be applied declaratively throughout your application in a flexible manner without your core application even knowing they exist. This is a powerful concept, as it keeps the security, transaction, and logging concerns from littering the application’s core business logic.

To demonstrate how aspects can be applied in Spring, let’s revisit the knight example, adding a basic logging aspect.

Figure 1.7 Calls to systemwide concerns such as logging and security are often scattered about in modules where those concerns are not their primary concern.

Figure 1.8 Using AOP, systemwide concerns blanket the components that they impact. This leaves the application components to focus on their specific business functionality.
1.4.2 AOP in action

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.

Hmm…a minstrel who sings about a knight, eh? That doesn’t sound too hard. Getting started, you create a Minstrel class, as shown in listing 1.10.

```java
package com.springinaction.chapter01.knight;
import org.apache.log4j.Logger;
public class Minstrel {
    private static final Logger SONG = Logger.getLogger(Minstrel.class);
    public void singBefore(Knight knight) {
        SONG.info("Fa la la; Sir " + knight.getName() + " is so brave!");
    }
    public void singAfter(Knight knight) {
        SONG.info("Tee-hee-he; Sir " + knight.getName() + " did embark on a quest!");
    }
}
```

In keeping with the dependency injection way of thinking, you alter KnightOfTheRoundTable to be given an instance of Minstrel:

```java
public class KnightOfTheRoundTable implements Knight {
    private Minstrel minstrel;
    public void setMinstrel(Minstrel minstrel) {
        this.minstrel = minstrel;
    }
    public HolyGrail embarkOnQuest() throws QuestFailedException {
        minstrel.singBefore(this);
        HolyGrail grail = quest.embark();
        minstrel.singAfter(this);
        return grail;
    }
}
```
That should do it! Oh wait… there’s only one small problem. As it is, each knight must stop and tell the minstrel to sing a song before the knight can continue with his quest (as in figure 1.9). Then after the quest, the knight must remember to tell the minstrel to continue singing of his exploits. Having to remember to stop and tell a minstrel what to do can certainly impede a knight’s quest-embarking.

Ideally, a minstrel would take more initiative and automatically sing songs without being explicitly told to do so. A knight shouldn’t know (or really even care) that his 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 turn the minstrel into an aspect that adds his song-writing services to a knight. Then the minstrel’s services would cover the functionality of the knight—all without the knight even knowing that the minstrel is there, as shown in figure 1.10.

As it turns out, it’s rather easy to turn the Minstrel class in listing 1.10 into an aspect using Spring’s AOP support. Let’s see how.

**Weaving the aspect**

There are several ways to implement aspects in Spring, and we’ll dig into all of them in chapter 4. But for the sake of this example, we’ll use the new AOP namespace introduced in Spring 2.0. To get started, you’ll want to be sure that you declare the namespace in the context definition XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/
                          spring-beans-2.0.xsd
                          http://www.springframework.org/schema/aop
                          spring-aop-2.0.xsd"/>
```

With the namespace declared, we’re ready to create the aspect. The bit of XML in listing 1.11 declares a minstrel as a bean in the Spring context and then creates an aspect that advises the knight bean.

There’s a lot going on in listing 1.11, so let’s break it down one bit at a time:

- The first thing we find is a `<bean>` declaration, creating a minstrel bean in Spring. This is the Minstrel class from listing 1.10. Minstrel doesn’t have any dependencies, so there’s no need to inject it with anything.

- Next up is the `<aop:config>` element. This element indicates that we’re about to do some AOP stuff. Most of Spring’s AOP configuration elements must be contained in `<aop:config>`.

- Within `<aop:config>` we have an `<aop:aspect>` element. This element indicates that we’re declaring an aspect. The functionality of the aspect is defined in the bean that is referred to by the `ref` attribute. In this case, the minstrel bean, which is a Minstrel, will provide the functionality of the aspect.
Applying aspect-oriented programming

- An aspect is made up of pointcuts (places where the aspect functionality will be applied) and advice (how to apply the functionality). The `<aop:pointcut>` element defines a pointcut that is triggered by the execution of an `embarkOnQuest()` method. (If you’re familiar with AspectJ, you may recognize the pointcut as being expressed in AspectJ syntax.)

- Finally, we have two bits of AOP advice. The `<aop:before>` element declares that the `singBefore()` method of Minstrel should be called before the pointcut, while the `<aop:after>` element declares that the `singAfter()` method of Minstrel should be called after the pointcut. The pointcut in both cases is a reference to `questPointcut`, which is the execution of `embarkOnQuest()`.

That’s all there is to it! We’ve just turned Minstrel into a Spring aspect. Don’t worry if this doesn’t make complete sense yet—you’ll see plenty more examples of Spring AOP in chapter 4 that should help clear this up. For now, there are two important points to take away from this example.

First, Minstrel is still a POJO—there’s nothing about Minstrel that indicates that it is to be used as an aspect. Instead, Minstrel was turned into an aspect declaratively in the Spring context.

Second, and perhaps more important, the knight no longer needs to tell the minstrel to sing about his exploits. As an aspect, the minstrel will take care of that automatically. In fact, the knight doesn’t even need to know of the minstrel’s existence. Consequently, the `KnightOfTheRoundTable` class can revert back to a simpler form as before:

```java
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;

    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }

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

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

Using AOP to chronicle a knight’s activities has been a lot of fun. But Spring’s AOP can be used for even more practical things than composing ageless sonnets about
knights. As you’ll see later, Spring employs AOP to provide enterprise services such as declarative transactions (chapter 6) and security (chapter 7).

1.5 Summary

You should now have a pretty good idea of what Spring brings to the table. Spring aims to make enterprise Java development easier and to promote loosely coupled code. Vital to this is dependency injection and AOP.

In this chapter, we got a small taste of dependency injection in Spring. DI is a way of associating application objects such that the objects don’t need to know where their dependencies come from or how they’re implemented. Rather than acquiring dependencies on their own, dependent objects are given the objects that they depend on. Because dependent objects often only know about their injected objects through interfaces, coupling is kept very low.

In addition to dependency injection, we also saw a glimpse of Spring’s AOP support. AOP enables you to centralize logic that would normally be scattered throughout an application in one place—an aspect. When Spring wires your beans together, these aspects can be woven in at runtime, effectively giving the beans new behavior.

Dependency injection and AOP are central to everything in Spring. Thus you must understand how to use these principal functions of Spring to be able to use the rest of the framework. In this chapter, we’ve just scratched the surface of Spring’s DI and AOP features. Over the next few chapters, we’ll dig deeper into DI and AOP. Without further ado, let’s move on to chapter 2 to learn how to wire objects together in Spring using dependency injection.