

covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls



MANNING

Chapter 4. Aspect-oriented Spring	1
Section 4.1. What's aspect-oriented programming?	2
Section 4.2. Selecting join points with pointcuts	8
Section 4.3. Declaring aspects in XML	10
Section 4.4. Annotating aspects	19
Section 4.5. Injecting AspectJ aspects	24
Section 4.6. Summary	27

Aspect-oriented Spring

This chapter covers

- Basics of aspect-oriented programming
- Creating aspects from POJOs
- Using `@AspectJ` annotations
- Injecting dependencies into `AspectJ` aspects

As I'm writing this chapter, Texas (where I reside) is going through several days of record-high temperatures. It's hot. In weather like this, air conditioning is a must. But the downside of air conditioning is that it uses electricity, and electricity costs money. There's little we can do to avoid paying for a cool and comfortable home. That's because every home has a meter that measures every kilowatt, and once a month someone comes by to read that meter so that the electric company accurately knows how much to bill us.

Now imagine what would happen if the meter went away and nobody came by to measure our electricity usage. Suppose that it were up to each homeowner to contact the electric company and report their electricity usage. Although it's possible that some obsessive homeowners would keep careful record of their lights, televisions, and air conditioning, most wouldn't bother. Most would estimate their usage and others wouldn't bother reporting it at all. It's too much trouble to monitor electrical usage and the temptation to not pay is too great.

Electricity on the honor system might be great for consumers, but it would be less than ideal for the electric companies. That's why we all have electric meters on our homes and why a meter reader drops by once per month to report the consumption to the electric company.

Some functions of software systems are like the electric meters on our homes. The functions need to be applied at multiple points within the application, but it's undesirable to explicitly call them at every point.

Monitoring electricity consumption is an important function, but it isn't foremost in most homeowners' minds. Mowing the lawn, vacuuming the carpet, and cleaning the bathroom are the kinds of things that homeowners are actively involved in. Monitoring the amount of electricity used by their house is a passive event from the homeowner's point of view. (Although it'd be great if mowing the lawn were also a passive event—especially on these hot days.)

In software, several activities are common to most applications. Logging, security, and transaction management are important, but should they be activities that your application objects are actively participating in? Or would it be better for your application objects to focus on the business domain problems they're designed for and leave certain aspects to be handled by someone else?

In software development, functions that span multiple points of an application are called *cross-cutting concerns*. Typically, these cross-cutting concerns are conceptually separate from (but often embedded directly within) the application's business logic. Separating these cross-cutting concerns from the business logic is where aspect-oriented programming (AOP) goes to work.

In chapter 2, you learned how to use dependency injection (DI) to manage and configure your application objects. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

Logging is a common example of the application of aspects. But it's not the only thing aspects are good for. Throughout this book, you'll see several practical applications of aspects, including declarative transactions, security, and caching.

This chapter explores Spring's support for aspects, including how to declare regular classes to be aspects and how to use annotations to create aspects. In addition, you'll see how AspectJ—another popular AOP implementation—can complement Spring's AOP framework. But first, before we get carried away with transactions, security, and caching, let's see how aspects are implemented in Spring, starting with a primer on a few of AOP's fundamentals.

4.1 What's aspect-oriented programming?

As stated earlier, aspects help to modularize cross-cutting concerns. In short, a cross-cutting concern can be described as any functionality that affects multiple points of an application. Security, for example, is a cross-cutting concern, in that many methods in an application can have security rules applied to them. Figure 4.1 gives a visual depiction of cross-cutting concerns.

Figure 4.1 represents a typical application that's broken down into modules. Each module's main concern is to provide services for its particular domain. But each module also requires similar ancillary functionalities, such as security and transaction management.

A common object-oriented technique for reusing common functionality is to apply inheritance or delegation. But inheritance can lead to a brittle object hierarchy if the same base class is used throughout an application, and delegation can be cumbersome because complicated calls to the delegate object may be required.

Aspects offer an alternative to inheritance and delegation that can be cleaner in many circumstances. With AOP, you still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you're applying the new feature. Cross-cutting concerns can now be modularized into special classes called *aspects*. This has two benefits. First, the logic for each concern is now in one place, as opposed to being scattered all over the code base. Second, our service modules are now cleaner since they only contain code for their primary concern (or core functionality) and secondary concerns have been moved to aspects.

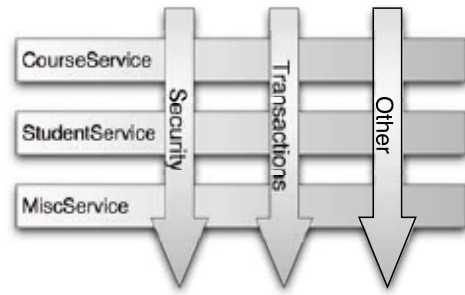


Figure 4.1 Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

4.1.1 Defining AOP terminology

Like most technologies, AOP has formed its own jargon. Aspects are often described in terms of advice, pointcuts, and join points. Figure 4.2 illustrates how these concepts are tied together.

Unfortunately, many of the terms used to describe AOP features aren't intuitive. Nevertheless, they're now part of the AOP idiom, and in order to understand AOP, you must know these terms. Before you walk the walk, you have to learn to talk the talk.

ADVICE

When a meter reader shows up at your house, his purpose is to report the number of kilowatt hours back to the electric company. Sure, he has a list of houses that he must visit and the information that he reports is important. But the actual act of recording electricity usage is the meter reader's main job.

Likewise, aspects have a purpose—a job they're meant to do. In AOP terms, the job of an aspect is called *advice*.

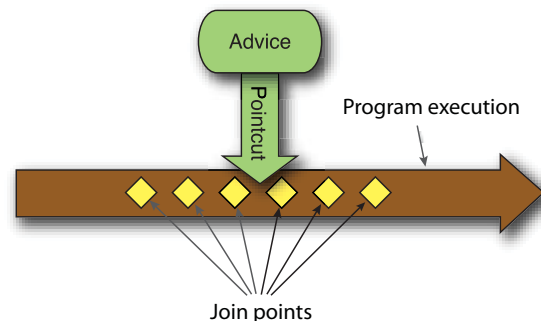


Figure 4.2 An aspect's functionality (advice) is woven into a program's execution at one or more join points.

Advice defines both the *what* and the *when* of an aspect. In addition to describing the job that an aspect will perform, advice addresses the question of when to perform the job. Should it be applied before a method is invoked? After the method is invoked? Both before and after method invocation? Or should it only be applied if a method throws an exception?

Spring aspects can work with five kinds of advice:

- *Before*—The advice functionality takes place before the advised method is invoked.
- *After*—The advice functionality takes place after the advised method completes, regardless of the outcome.
- *After-returning*—The advice functionality takes place after the advised method successfully completes.
- *After-throwing*—The advice functionality takes place after the advised method throws an exception.
- *Around*—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

JOIN POINTS

An electric company services several houses, perhaps even an entire city. Each house will have an electric meter that needs to be read and thus each house is a potential target for the meter reader. The meter reader could potentially read all kinds of devices, but to do his job, he needs to target electric meters that are attached to houses.

In the same way, your application may have thousands of opportunities for advice to be applied. These opportunities are known as join points. A *join point* is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified. These are the points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

POINTCUTS

It's not possible for any one meter reader to visit all houses serviced by the electric company. Instead, each one is assigned a subset of all of the houses to visit. Likewise, an aspect doesn't necessarily advise all join points in an application. *Pointcuts* help narrow down the join points advised by an aspect.

If advice defines the *what* and *when* of aspects, then pointcuts define the *where*. A pointcut definition matches one or more join points at which advice should be woven. Often you specify these pointcuts using explicit class and method names or through regular expressions that define matching class and method name patterns. Some AOP frameworks allow you to create dynamic pointcuts that determine whether to apply advice based on runtime decisions, such as the value of method parameters.

ASPECTS

When a meter reader starts his day, he knows both what he's supposed to do (report electricity usage) and which houses to collect that information from. Thus he knows everything he needs to know to get his job done.

An *aspect* is the merger of advice and pointcuts. Taken together, advice and pointcuts define everything there is to know about an aspect—what it does and where and when it does it.

INTRODUCTIONS

An *introduction* allows you to add new methods or attributes to existing classes. For example, you could create an Auditable advice class that keeps the state of when an object was last modified. This could be as simple as having one method, `setLastModified(Date)`, and an instance variable to hold this state. The new method and instance variable can then be introduced to existing classes without having to change them, giving them new behavior and state.

WEAVING

Weaving is the process of applying aspects to a target object to create a new proxied object. The aspects are woven into the target object at the specified join points. The weaving can take place at several points in the target object's lifetime:

- *Compile time*—Aspects are woven in when the target class is compiled. This requires a special compiler. AspectJ's weaving compiler weaves aspects this way.
- *Classload time*—Aspects are woven in when the target class is loaded into the JVM. This requires a special `ClassLoader` that enhances that target class's byte-code before the class is introduced into the application. AspectJ 5's *load-time weaving (LTW)* support weaves aspects in this way.
- *Runtime*—Aspects are woven in sometime during the execution of the application. Typically, an AOP container will dynamically generate a proxy object that will delegate to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

That's a lot of new terms to get to know. Revisiting figure 4.2, you can now see how advice contains the cross-cutting behavior that needs to be applied to an application's objects. The join points are all the points within the execution flow of the application that are candidates to have advice applied. The pointcut defines where (at what join points) that advice is applied. The key concept you should take from this is that pointcuts define which join points get advised.

Now that you're familiar with some basic AOP terminology, let's see how these core AOP concepts are implemented in Spring.

4.1.2 *Spring's AOP support*

Not all AOP frameworks are created equal. They may differ in how rich their join point models are. Some allow you to apply advice at the field modification level, whereas others only expose the join points related to method invocations. They may also differ in how and when they weave the aspects. Whatever the case, the ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework.

Much has changed in the AOP framework landscape in the past few years. There has been some housecleaning among the AOP frameworks, resulting in some frameworks merging and others going extinct. In 2005, the AspectWerkz project merged with AspectJ, marking the last significant activity in the AOP world and leaving us with three dominant AOP frameworks:

- AspectJ (<http://eclipse.org/aspectj>)
- JBoss AOP (<http://www.jboss.org/jbossaop>)
- Spring AOP (<http://www.springframework.org>)

Since this is a Spring book, we'll focus on Spring AOP. Even so, there's a lot of synergy between the Spring and AspectJ projects, and the AOP support in Spring borrows a lot from the AspectJ project.

Spring's support for AOP comes in four flavors:

- Classic Spring proxy-based AOP
- `@AspectJ` annotation-driven aspects
- Pure-POJO aspects
- Injected AspectJ aspects (available in all versions of Spring)

The first three items are all variations on Spring's proxy-based AOP. Consequently, Spring's AOP support is limited to method interception. If your AOP needs exceed simple method interception (constructor or property interception, for example), you'll want to consider implementing aspects in AspectJ, perhaps taking advantage of Spring DI to inject Spring beans into AspectJ aspects.

What? No classic Spring AOP?

The term *classic* usually carries a good connotation. Classic cars, classic golf tournaments, and classic Coca-Cola are all good things.

But Spring's classic AOP programming model isn't so great. Oh, it was good in its day. But now Spring supports much cleaner and easier ways to work with aspects. When held up against simple declarative AOP and annotation-based AOP, Spring's classic AOP seems bulky and overcomplicated. Working directly with `ProxyFactoryBean` can be wearying.

So I've chosen to not include any discussion of classic Spring AOP in this edition. If you're really curious about how it works, then you may look at the first two editions of this book. But I think you'll find that the new Spring AOP models are much easier to work with.

We'll explore more of these Spring AOP techniques in this chapter. But before we get started, it's important to understand a few key points of Spring's AOP framework.

SPRING ADVICE IS WRITTEN IN JAVA

All of the advice you create within Spring is written in a standard Java class. That way, you get the benefit of developing your aspects in the same integrated development

environment (IDE) you'd use for your normal Java development. What's more, the pointcuts that define where advice should be applied are typically written in XML in your Spring configuration file. This means both the aspect's code and configuration syntax will be familiar to Java developers.

Contrast this with AspectJ. Although AspectJ now supports annotation-based aspects, AspectJ also comes as a language extension to Java. This approach has benefits and drawbacks. By having an AOP-specific language, you get more power and fine-grained control, as well as a richer AOP toolset. But you're required to learn a new tool and syntax to accomplish this.

SPRING ADVISES OBJECTS AT RUNTIME

In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. As illustrated in figure 4.3, the proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean.

Between the time when the proxy intercepts the method call and the time when it invokes the target bean's method, the proxy performs the aspect logic.

Spring doesn't create a proxied object until that proxied bean is needed by the application. If you're using an `ApplicationContext`, the proxied objects will be created when it loads all of the beans from the `BeanFactory`. Because Spring creates proxies at runtime, you don't need a special compiler to weave aspects in Spring's AOP.

SPRING ONLY SUPPORTS METHOD JOIN POINTS

As mentioned earlier, multiple join point models are available through various AOP implementations. Because it's based on dynamic proxies, Spring only supports method join points. This is in contrast to some other AOP frameworks, such as AspectJ and JBoss, which provide field and constructor join points in addition to method pointcuts. Spring's lack of field pointcuts prevents you from creating very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there's no way to apply advice when a bean is instantiated.

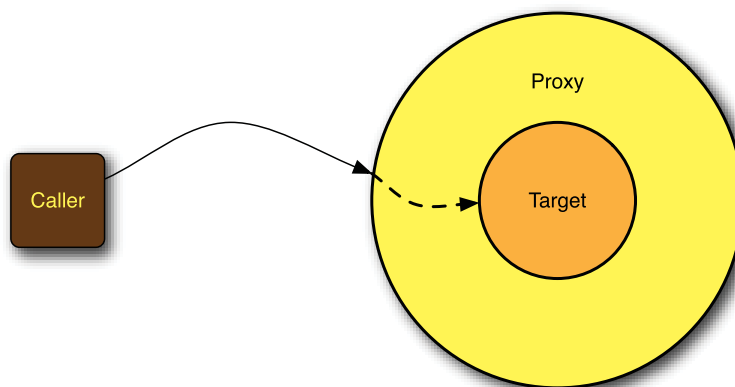


Figure 4.3 Spring aspects are implemented as proxies that wrap the target object. The proxy handles method calls, performs additional aspect logic, and then invokes the target method.

But method interception should suit most, if not all, of your needs. If you find yourself in need of more than method interception, you'll want to complement Spring AOP with AspectJ.

Now you have a general idea of what AOP does and how it's supported by Spring. It's time to get our hands dirty creating aspects in Spring. Let's start with Spring's declarative AOP model.

4.2 Selecting join points with pointcuts

As mentioned before, pointcuts are used to pinpoint where an aspect's advice should be applied. Along with an aspect's advice, pointcuts are among the most fundamental elements of an aspect. Therefore, it's important to know how to write pointcuts.

In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language. If you're already familiar with AspectJ, then defining pointcuts in Spring should feel natural. But in case you're new to AspectJ, this section will serve as a quick lesson on writing AspectJ-style pointcuts. For a more detailed discussion on AspectJ and AspectJ's pointcut expression language, I strongly recommend Ramnivas Laddad's *AspectJ in Action, Second Edition*.

The most important thing to know about AspectJ pointcuts as they pertain to Spring AOP is that Spring only supports a subset of the pointcut designators available in AspectJ. Recall that Spring AOP is proxy-based and certain pointcut expressions aren't relevant to proxy-based AOP. Table 4.1 lists the AspectJ pointcut designators that are supported in Spring AOP.

Table 4.1 Spring leverages AspectJ's pointcut expression language for defining Spring aspects.

AspectJ designator	Description
<code>args()</code>	Limits join point matches to the execution of methods whose arguments are instances of the given types
<code>@args()</code>	Limits join point matches to the execution of methods whose arguments are annotated with the given annotation types
<code>execution()</code>	Matches join points that are method executions
<code>this()</code>	Limits join point matches to those where the bean reference of the AOP proxy is of a given type
<code>target()</code>	Limits join point matches to those where the target object is of a given type
<code>@target()</code>	Limits matching to join points where the class of the executing object has an annotation of the given type
<code>within()</code>	Limits matching to join points within certain types
<code>@within()</code>	Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
<code>@annotation</code>	Limits join point matches to those where the subject of the join point has the given annotation

Attempting to use any of AspectJ's other designators will result in an `IllegalArgumentException` being thrown.

As you browse through the supported designators, note that the execution designator is the only one that actually performs matches. The other designators are used to limit those matches. This means that execution is the primary designator you'll use in every pointcut definition you write. You'll use the other designators to constrain the pointcut's reach.

4.2.1 Writing pointcuts

For example, the pointcut expression shown in figure 4.4 can be used to apply advice whenever an `Instrument`'s `play()` method is executed:

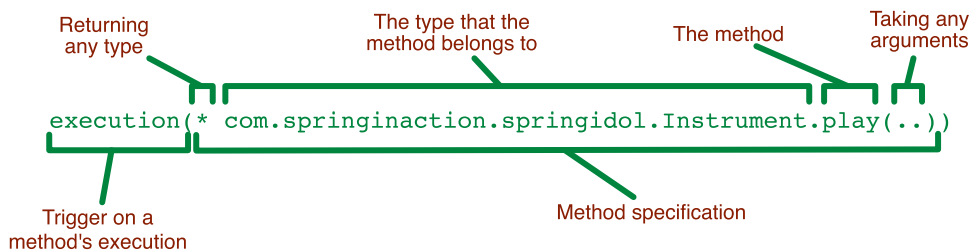


Figure 4.4 Selecting the `Instrument`'s `play()` method with an AspectJ pointcut expression

We used the `execution()` designator to select the `Instrument`'s `play()` method. The method specification starts with an asterisk, which indicates that we don't care what type the method returns. Then we specify the fully qualified class name and the name of the method we want to select. For the method's parameter list, we use the double-dot (`..`), indicating that the pointcut should select any `play()` method, no matter what the argument list is.

Now let's suppose that we want to confine the reach of that pointcut to only the `com.springinaction.springidol` package. In that case, we could limit the match by tacking on a `within()` designator, as shown in figure 4.5.

Note that we used the `&&` operator to combine the `execution()` and `within()` designators in an "and" relationship (where both designators must match for the pointcut to match). Similarly, we could've used the `||` operator to indicate an "or" relationship. And the `!` operator can be used to negate the effect of a designator.

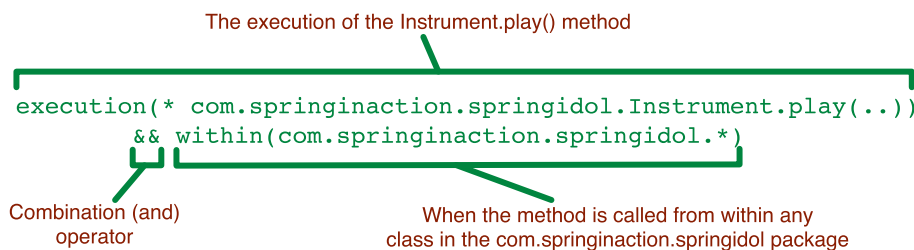


Figure 4.5 Limiting a pointcut's reach using the `within()` designator

Since ampersands have special meaning in XML, you're free to use `&&` in place of `&` when specifying pointcuts in a Spring XML-based configuration. Likewise, `or` and `not` can be used in place of `||` and `!` (respectively).

4.2.2 Using Spring's `bean()` designator

In addition to the designators listed in table 4.1, Spring 2.5 introduced a new `bean()` designator that lets you identify beans by their ID within a pointcut expression. `bean()` takes a bean ID or name as an argument and limits the pointcut's effect to that specific bean.

For example, consider the following pointcut:

```
execution(* com.springinaction.springidol.Instrument.play())
    and bean(eddie)
```

Here we're saying that we want to apply aspect advice to the execution of an `Instruments play()` method, but limited to the bean whose ID is `eddie`.

Narrowing a pointcut to a specific bean may be valuable in some cases, but we can also use negation to apply an aspect to all beans that don't have a specific ID:

```
execution(* com.springinaction.springidol.Instrument.play())
    and !bean(eddie)
```

In this case, the aspect's advice will be woven into all beans whose ID isn't `eddie`.

Now that we've covered the basics of writing pointcuts, let's see how to write the advice and declare the aspects that use those pointcuts.

4.3 Declaring aspects in XML

If you're familiar with Spring's classic AOP model, you'll know that working with `ProxyFactoryBean` is clumsy. The Spring development team recognized this and set out to provide a better way of declaring aspects in Spring. The outcome of this effort is found in Spring's `aop` configuration namespace. The AOP configuration elements are summarized in table 4.2.

Table 4.2 Spring's AOP configuration elements simplify declaration of POJO-based aspects.

AOP configuration element	Purpose
<code><aop:advisor></code>	Defines an AOP advisor.
<code><aop:after></code>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<code><aop:after-returning></code>	Defines an AOP after-returning advice.
<code><aop:after-throwing></code>	Defines an AOP after-throwing advice.
<code><aop:around></code>	Defines an AOP around advice.
<code><aop:aspect></code>	Defines an aspect.
<code><aop:aspectj-autoproxy></code>	Enables annotation-driven aspects using <code>@AspectJ</code> .

Table 4.2 Spring's AOP configuration elements simplify declaration of POJO-based aspects. (continued)

AOP configuration element	Purpose
<code><aop:before></code>	Defines an AOP before advice.
<code><aop:config></code>	The top-level AOP element. Most <code><aop:*></code> elements must be contained within <code><aop:config></code> .
<code><aop:declare-parents></code>	Introduces additional interfaces to advised objects that are transparently implemented.
<code><aop:pointcut></code>	Defines a pointcut.

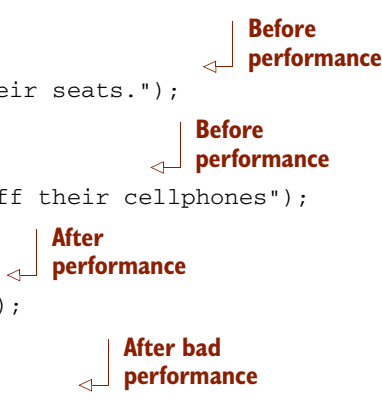
In chapter 2, we demonstrated dependency injection by putting on a talent show called *Spring Idol*. In that example, we wired up several performers as Spring `<bean>`s to show their stuff. It was all greatly amusing, but a show like that needs an audience or else there's little point.

Therefore, to illustrate Spring AOP, let's create an Audience class for our talent show. The following class defines the functions of an audience.

Listing 4.1 The Audience class for our talent competition

```
package com.springinaction.springidol;

public class Audience {
    public void takeSeats() {
        System.out.println("The audience is taking their seats.");
    }
    public void turnOffCellPhones() {
        System.out.println("The audience is turning off their cellphones");
    }
    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }
    public void demandRefund() {
        System.out.println("Boo! We want our money back!");
    }
}
```



As you can see, there's nothing remarkable about the Audience class. It's a basic Java class with a handful of methods. And we can register it as a bean in the Spring application context like any other class:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Despite its unassuming appearance, what's remarkable about Audience is that it has all the makings of an aspect. It just needs a little of Spring's special AOP magic.

4.3.1 Declaring before and after advice

Using Spring's AOP configuration elements, as shown in the following listing, you can turn the audience bean into an aspect.

Listing 4.2 Defining an audience aspect using Spring's AOP configuration elements

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:before pointcut=
      "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="takeSeats" />
    <aop:before pointcut=
      "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="turnOffCellPhones" />
    <aop:after-returning pointcut=
      "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="applaud" />
    <aop:after-throwing pointcut=
      "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="demandRefund" />
  </aop:aspect>
</aop:config>
```

Reference audience bean

Before performance

Before performance

After performance

After bad performance

The first thing to notice about the Spring AOP configuration elements is that most of them must be used within the context of the `<aop:config>` element. There are a few exceptions to this rule, but when it comes to declaring beans as aspects you'll always start with the `<aop:config>` element.

Within `<aop:config>` you may declare one or more advisors, aspects, or pointcuts. In listing 4.2, we declared a single aspect using the `<aop:aspect>` element. The `ref` attribute references the POJO bean that will be used to supply the functionality of the aspect—in this case, `audience`. The bean that's referenced by the `ref` attribute will supply the methods called by any advice in the aspect.

The aspect has four different bits of advice. The two `<aop:before>` elements define *method before advice* that will call the `takeSeats()` and `turnOffCellPhones()` methods (declared by the `method` attribute) of the Audience bean before any methods matching the pointcut are executed. The `<aop:after-returning>` element defines an *after-returning advice* to call the `applaud()` method after the pointcut. Meanwhile, the `<aop:after-throwing>` element defines an *after-throwing advice* to call the `demandRefund()` method if any exceptions are thrown. Figure 4.6 shows how the advice logic is woven into the business logic.

In all advice elements, the `pointcut` attribute defines the pointcut where the advice will be applied. The value given to the `pointcut` attribute is a pointcut defined in AspectJ's pointcut expression syntax.

You'll notice that the value of the `pointcut` attribute is the same for all of the advice elements. That's because all of the advice is being applied to the same pointcut.

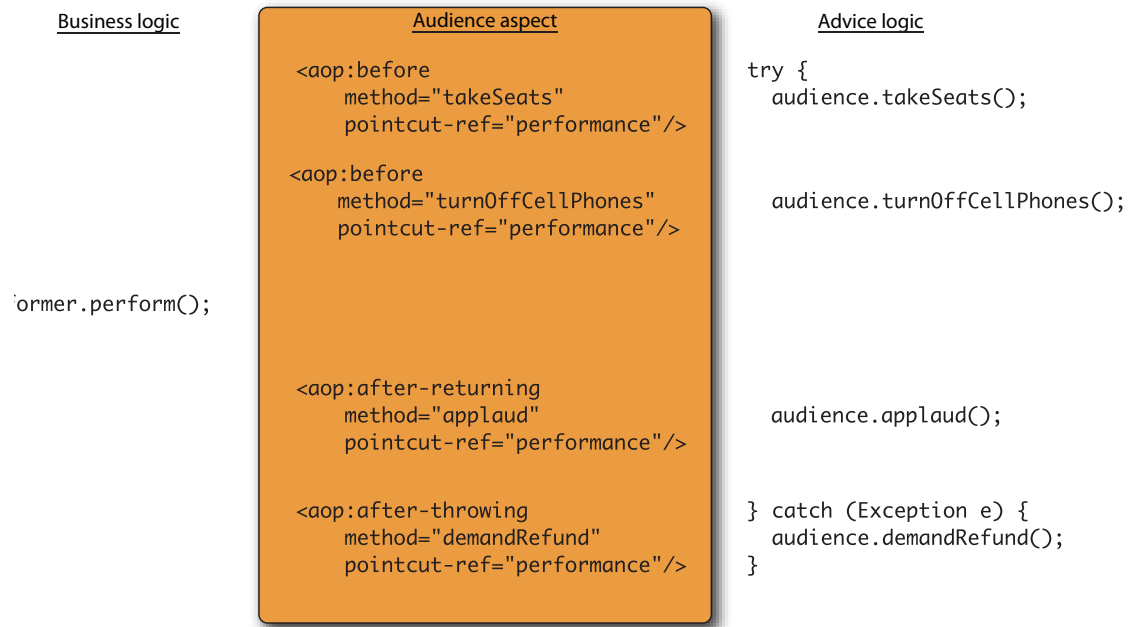


Figure 4.6 The Audience aspect includes four bits of advice which weave advice logic around methods that match the aspect's pointcut.

This presents a DRY (don't repeat yourself) principle violation. If you decide later to change the pointcut, you must change it in four different places.

To avoid duplication of the pointcut definition, you may choose to define a named pointcut using the `<aop:pointcut>` element. The following XML shows how the `<aop:pointcut>` element is used within the `<aop:aspect>` element to define a named pointcut that can be used by all of the advice elements.

Listing 4.3 Defining a named pointcut to eliminate redundant pointcut definitions

```

<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
      "execution(* com.springinaction.springidol.Performer.perform(..)"
    />
    <aop:before
      pointcut-ref="performance"
      method="takeSeats" />
    <aop:before
      pointcut-ref="performance"
      method="turnOffCellPhones" />
    <aop:after-returning
      pointcut-ref="performance"
      method="applaud" />
    <aop:after-throwing
      pointcut-ref="performance"
      method="demandRefund" />
  </aop:aspect>
</aop:config>

```

Define pointcut

Reference pointcut

Now the pointcut is defined in a single location and is referenced across multiple advice elements. The `<aop:pointcut>` element defines the pointcut to have an id of `performance`. Meanwhile, all of the advice elements have been changed to reference the named pointcut with the `pointcut-ref` attribute.

As used in listing 4.3, the `<aop:pointcut>` element defines a pointcut that can be referenced by all advices within the same `<aop:aspect>` element. But you can also define pointcuts that can be used across multiple aspects by placing the `<aop:pointcut>` elements within the scope of the `<aop:config>` element.

4.3.2 Declaring around advice

The current implementation of `Audience` works great. But basic before and after advice have some limitations. Specifically, it's tricky to share information between before advice and after advice without resorting to storing that information in member variables.

For example, suppose that in addition to putting away cell phones and applauding at the end, you also want the audience to keep their eyes on their watches and report how long the performance takes. The only way to accomplish this with before and after advice is to note the start time in before advice and report the length of time in some after advice. But you'd have to store the start time in a member variable. Since `Audience` is a singleton, it wouldn't be thread safe to retain state like that.

Around advice has an advantage over before and after advice in this regard. With around advice, you can accomplish the same thing as you can with distinct before and after advice, but do it in a single method. Since the entire set of advice takes place in a single method, there's no need to retain state in a member variable.

For example, consider the new `watchPerformance()` method.

Listing 4.4 The `watchPerformance()` method provides AOP around advice.

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
        System.out.println("The audience is taking their seats.");
        System.out.println("The audience is turning off their cellphones");
        long start = System.currentTimeMillis();
        joinpoint.proceed();
        long end = System.currentTimeMillis();
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
        System.out.println("The performance took " + (end - start)
            + " milliseconds.");
    } catch (Throwable t) {
        System.out.println("Boo! We want our money back!");
    }
}
```

The first thing you'll notice about this new advice method is that it's given a `ProceedingJoinPoint` as a parameter. This object is necessary, as it's how we'll be able to invoke the advised method from within our advice. The advice method will do

everything it needs to do and, when it's ready to pass control to the advised method, it'll call `ProceedingJoinPoint`'s `proceed()` method.

Note that it's crucial that you remember to include a call to the `proceed()` method. If you don't, then your advice will effectively block access to the advised method. Maybe that's what you want, but chances are good that you do want the advised method to be executed at some point.

What's also interesting is that just as you can omit a call to the `proceed()` method to block access to the advised method, you can also invoke it multiple times from within the advice. One reason for doing this may be to implement retry logic to perform repeated attempts on the advised method should it fail.

In the case of the audience aspect, the `watchPerformance()` method contains all of the functionality of the previous four advice methods, but all of it's contained in this single method, and this method is responsible for its own exception handling. You'll also note that just before the join point's `proceed()` method is called, the current time is recorded in a local variable. Just after the method returns, the elapsed time is reported.

Declaring around advice isn't dramatically different from declaring other types of advice. All you need to do is use the `<aop:around>` element.

Listing 4.5 Defining a named pointcut to eliminate redundant pointcut definitions

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance2" expression=
      "execution(* com.springinaction.springidol.Performer.perform(..))"
    />

    <aop:around
      pointcut-ref="performance2"
      method="watchPerformance()" />
  </aop:aspect>
</aop:config>
```

Declare
around advice

As with the other advice XML elements, `<aop:around>` is given a pointcut and the name of an advice method. Here we're using the same pointcut as before, but have set the method attribute to point to the new `watchPerformance()` method.

4.3.3 Passing parameters to advice

So far, our aspects have been simple, taking no parameters. The only exception is that the `watchPerformance()` method that we wrote for the around advice example took a `ProceedingJoinPoint` as a parameter. Other than that, the advice we've written hasn't bothered to look at any parameters passed to the advised methods. That's been okay, though, because the `perform()` method that we were advising didn't take any parameters.

Nevertheless, there are times when it may be useful for advice to not only wrap a method, but also inspect the values of the parameters passed to that method.

To see how this works, imagine a new type of contestant in the *Spring Idol* competition. This new contestant is a mind reader, as defined by the `MindReader` interface:

```
package com.springinaction.springidol;

public interface MindReader {
    void interceptThoughts(String thoughts);

    String getThoughts();
}
```

A `MindReader` does two basic things: it intercepts a volunteer's thoughts and reports those thoughts. A simple implementation of `MindReader` is the `Magician` class:

```
package com.springinaction.springidol;

public class Magician implements MindReader {
    private String thoughts;

    public void interceptThoughts(String thoughts) {
        System.out.println("Intercepting volunteer's thoughts");
        this.thoughts = thoughts;
    }

    public String getThoughts() {
        return thoughts;
    }
}
```

Now you need to give your mind reader someone whose mind he can read. For that, here's the `Thinker` interface:

```
package com.springinaction.springidol;

public interface Thinker {
    void thinkOfSomething(String thoughts);
}
```

The `Volunteer` class provides a basic implementation of `Thinker`:

```
package com.springinaction.springidol;

public class Volunteer implements Thinker {
    private String thoughts;

    public void thinkOfSomething(String thoughts) {
        this.thoughts = thoughts;
    }

    public String getThoughts() {
        return thoughts;
    }
}
```

The details of `Volunteer` aren't terribly interesting or important. What's interesting is how the `Magician` will intercept the `Volunteer`'s thoughts using Spring AOP.

To pull off this feat of telepathy, you're going to use the same `<aop:aspect>` and `<aop:before>` elements as before. But this time you're going to configure them to pass the advised method's parameters to the advice.


```

<aop:config>
  <aop:aspect ref="magician">
    <aop:pointcut id="thinking"
      expression="execution(*
        com.springinaction.springidol.Thinker.thinkOfSomething(String)
        and args(thoughts)" />

    <aop:before
      pointcut-ref="thinking"
      method="interceptThoughts"
      arg-names="thoughts" />
  </aop:aspect>
</aop:config>

```

The key to the Magician's ESP is found in the pointcut definition and in the `<aop:before>`'s `arg-names` attribute. The pointcut identifies the `Thinker`'s `thinkOfSomething()` method, specifying a `String` argument. And it follows up with an `args` parameter to identify the argument as `thoughts`.

Meanwhile, the `<aop:before>` advice declaration refers to the `thoughts` argument, indicating that it should be passed into the `Magician`'s `interceptThoughts()` method.

Now, whenever the `thinkOfSomething()` method is invoked on the volunteer bean, the `Magician` will intercept those thoughts. To prove it, here's a simple test class with the following method:

```

@Test
public void magicianShouldReadVolunteersMind() {
    volunteer.thinkOfSomething("Queen of Hearts");

    assertEquals("Queen of Hearts", magician.getThoughts());
}

```

We'll talk more about writing unit tests and integration tests in Spring in the next chapter. For now, just note that the test will pass because the `Magician` will always know whatever the `Volunteer` is thinking.

Now let's see how to use Spring AOP to add new functionality to existing objects through the power of introduction.

4.3.4 **Introducing new functionality with aspects**

Some languages, such as Ruby and Groovy, have the notion of open classes. They make it possible to add new methods to an object or class without directly changing the definition of those objects/classes. Unfortunately, Java isn't quite that dynamic. Once a class has been compiled, there's little you can do to append new functionality to it.

But if you think about it, isn't that what we've been doing in this chapter with aspects? Sure, we haven't added any new methods to objects, but we're adding new functionality around the methods that the objects already have. If an aspect can wrap existing methods with additional functionality, why not add new methods to the object? In fact, using an AOP concept known as *introduction*, aspects can attach all new methods to Spring beans.

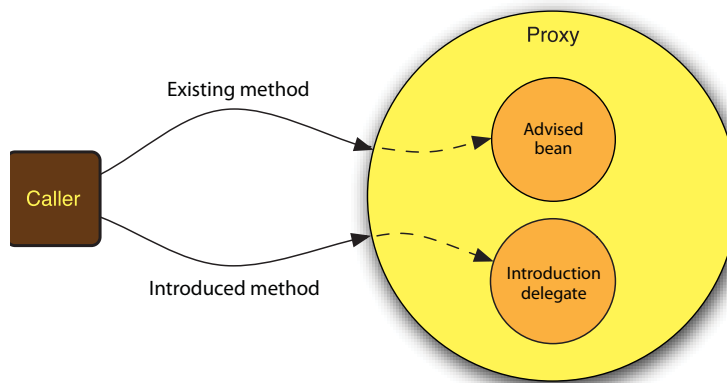


Figure 4.7 With Spring AOP, you can introduce new methods to a bean. A proxy intercepts the calls and delegates to a different object that implements the method.

Recall that in Spring, aspects are just proxies that implement the same interface(s) as the beans that they wrap. What if, in addition to implementing those interfaces, the proxy were to also be exposed through some new interface? Then any bean that's advised by the aspect will appear to implement the new interface, even if its underlying implementation class doesn't. Figure 4.7 illustrates how this works.

What you'll notice from figure 4.7 is that when a method on the introduced interface is called, the proxy delegates the call to some other object that provides the implementation of the new interface. Effectively this gives us one bean whose implementation is split across multiple classes.

Putting this idea to work, let's say that you want to introduce the following Contestant interface to all of the performers in our example:

```
package com.springinaction.springidol;

public interface Contestant {
    void receiveAward();
}
```

I suppose that we could visit all implementations of `Performer` and change them so that they also implement `Contestant`. But, from a design standpoint, that may not be the most prudent move (because `Contestants` and `Performers` aren't necessarily mutually inclusive concepts). Moreover, it may not even be possible to change all of the implementations of `Performer`, especially if we're working with third-party implementations and don't have the source code.

Thankfully, AOP introductions can help us out here without compromising design choices or requiring invasive changes to the existing implementations. To pull it off, you must use the `<aop:declare-parents>` element:

```
<aop:aspect>
  <aop:declare-parents
    types-matching="com.springinaction.springidol.Performer+"
    implement-interface="com.springinaction.springidol.Contestant"
    default-impl="com.springinaction.springidol.GraciousContestant"
  />
</aop:aspect>
```

As its name implies, `<aop:declare-parents>` declares that the beans it advises will have new parents in its object hierarchy. Specifically, in this case we're saying that the beans whose type matches the `Performer` interface (per the `types-matching` attribute) should have `Contestant` in their parentage (per the `implement-interface` attribute). The final matter to settle is where the implementation of the `Contestant`'s methods will come from.

There are two ways to identify the implementation of the introduced interface. In this case, we're using the `default-impl` attribute to explicitly identify the implementation by its fully-qualified class name. Alternatively, we could've identified it using the `delegate-ref` attribute:

```
<aop:declare-parents
  types-matching="com.springinaction.springidol.Performer+"
  implement-interface="com.springinaction.springidol.Contestant"
  delegate-ref="contestantDelegate"
/>
```

The `delegate-ref` attribute refers to a Spring bean as the introduction delegate. This assumes that a bean with an ID of `contestantDelegate` exists in the Spring context:

```
<bean id="contestantDelegate"
  class="com.springinaction.springidol.GraciousContestant" />
```

The difference between directly identifying the delegate using `default-impl` and indirectly using `delegate-ref` is that the latter will be a Spring bean that itself may be injected, advised, or otherwise configured through Spring.

4.4 **Annotating aspects**

A key feature introduced in AspectJ 5 is the ability to use annotations to create aspects. Prior to AspectJ 5, writing AspectJ aspects involved learning a Java language extension. But AspectJ's annotation-oriented model makes it simple to turn any class into an aspect by sprinkling a few annotations around. This new feature is commonly referred to as *@AspectJ*.

Looking back at our `Audience` class, we see that `Audience` contained all of the functionality needed for an audience, but none of the details to make it an aspect. That left us having to declare advice and pointcuts in XML.

But with *@AspectJ* annotations, we can revisit our `Audience` class and turn it into an aspect without the need for any additional classes or bean declarations. The following shows the new `Audience` class, now annotated to be an aspect.

Listing 4.6 Annotating Audience to be an aspect

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
```

```

@Aspect
public class Audience {
    @Pointcut(
        "execution(* com.springinaction.springidol.Performer.perform(..))")
    public void performance() {
    }

    @Before("performance()")
    public void takeSeats() {
        System.out.println("The audience is taking their seats.");
    }

    @Before("performance()")
    public void turnOffCellPhones() {
        System.out.println("The audience is turning off their cellphones");
    }

    @AfterReturning("performance()")
    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }

    @AfterThrowing("performance()")
    public void demandRefund() {
        System.out.println("Boo! We want our money back!");
    }
}

```

Define
pointcut

Before
performance

Before
performance

After
performance

After bad
performance

The new Audience class is now annotated with `@Aspect`. This annotation indicates that Audience isn't just any POJO but is an aspect.

The `@Pointcut` annotation is used to define a reusable pointcut within an `@AspectJ` aspect. The value given to the `@Pointcut` annotation is an `AspectJ` pointcut expression—here indicating that the pointcut should match the `perform()` method of a `Performer`. The name of the pointcut is derived from the name of the method to which the annotation is applied. Therefore, the name of this pointcut is `performance()`. The actual body of the `performance()` method is irrelevant and in fact should be empty. The method itself is just a marker, giving the `@Pointcut` annotation something to attach itself to.

Each of the audience's methods has been annotated with advice annotations. The `@Before` annotation has been applied to both `takeSeats()` and `turnOffCellPhones()` to indicate that these two methods are before advice. The `@AfterReturning` annotation indicates that the `applaud()` method is an after-returning advice method. And the `@AfterThrowing` annotation is placed on `demandRefund()` so that it'll be called if any exceptions are thrown during the performance.

The name of the `performance()` pointcut is given as the value parameter to all of the advice annotations. This tells each advice method where it should be applied.

Note that aside from the annotations and the no-op `performance()` method, the Audience class is functionally unchanged. This means that it's still a simple Java object and can be used as such. It can also still be wired in Spring as follows:

```

<bean id="audience"
      class="com.springinaction.springidol.Audience" />

```

Because the `Audience` class contains everything that's needed to define its own pointcuts and advice, there's no more need for pointcut and advice declarations in the XML configuration. There's one last thing to do to make Spring apply `Audience` as an aspect. You must declare an autoproxy bean in the Spring context that knows how to turn `@AspectJ`-annotated beans into proxy advice.

For that purpose, Spring comes with an autoproxy creator class called `AnnotationAwareAspectJAutoProxyCreator`. You could register an `AnnotationAwareAspectJAutoProxyCreator` as a `<bean>` in the Spring context, but that would require a lot of typing (believe me... I've typed it a few times before). Instead, to simplify that rather long name, Spring also provides a custom configuration element in the `aop` namespace that's much easier to remember:

```
<aop:aspectj-autoproxy />
```

`<aop:aspectj-autoproxy/>` will create an `AnnotationAwareAspectJAutoProxyCreator` in the Spring context and will automatically proxy beans whose methods match the pointcuts defined with `@Pointcut` annotations in `@Aspect`-annotated beans.

To use the `<aop:aspectj-autoproxy>` configuration element, you'll need to remember to include the `aop` namespace in your Spring configuration file:

```
<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-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

You should be aware that `<aop:aspectj-autoproxy>` only uses `@AspectJ`'s annotations as a guide for creating proxy-based aspects. Under the covers, it's still Spring-style aspects. This is significant because it means that although you're using `@AspectJ`'s annotations, you're still limited to proxying method invocations. If you want to be able to exploit the full power of AspectJ, you'll have to use the AspectJ runtime and not rely on Spring to create proxy-based aspects.

It's also worth mentioning at this point that both the `<aop:aspect>` element and the `@AspectJ` annotations are effective ways to turn a POJO into an aspect. But `<aop:aspect>` has one distinct advantage over `@AspectJ` in that you don't need the source code of the class that's to provide the aspect's functionality. With `@AspectJ`, you must annotate the class and methods, which requires having the source code. But `<aop:aspect>` can reference any bean.

Now let's see how to create around advice using `@AspectJ` annotations.

4.4.1 **Annotating around advice**

Just as with Spring's XML-based AOP, you're not limited to before and after advice types when using `@AspectJ` annotations. You may also choose to create around advice. For that, you must use the `@Around` annotation, as in the following example:


```

@Around("performance()")
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
        System.out.println("The audience is taking their seats.");
        System.out.println("The audience is turning off their cellphones");

        long start = System.currentTimeMillis();
        joinpoint.proceed();
        long end = System.currentTimeMillis();

        System.out.println("CLAP CLAP CLAP CLAP CLAP");

        System.out.println("The performance took " + (end - start)
            + " milliseconds.");
    } catch (Throwable t) {
        System.out.println("Boo! We want our money back!");
    }
}

```

Here the `@Around` annotation indicates that the `watchPerformance()` method is to be applied as around advice to the `performance()` pointcut. This should look oddly familiar, as it's the same `watchPerformance()` method that we saw before. The only difference is that it's now annotated with `@Around`.

As you may recall from before, around advice methods must remember to explicitly invoke `proceed()` so that the proxied method will be invoked. But simply annotating a method with `@Around` isn't enough to provide a `proceed()` method to call. Therefore, methods that are to be around advice must take a `ProceedingJoinPoint` object as an argument and then call the `proceed()` method on that object.

4.4.2 Passing arguments to annotated advice

Supplying parameters to advice using `@AspectJ` annotation isn't much different than how we did it with Spring's XML-based aspect declaration. In fact, for the most part, the XML elements we used earlier translate almost straight into equivalent `@AspectJ` annotations, as you can see in the new `Magician` class.

Listing 4.7 Using `@AspectJ` annotations to turn a `Magician` into an aspect

```

package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Magician implements MindReader {
    private String thoughts;

    @Pointcut("execution(* com.springinaction.springidol.*
        + "Thinker.thinkOfSomething(String)) && args(thoughts)")
    public void thinking(String thoughts) {
    }

    @Before("thinking(thoughts)")
    public void interceptThoughts(String thoughts) {

```

Declare
parameterized
pointcut

Pass
parameters
into advice

```

        System.out.println("Intercepting volunteer's thoughts : " + thoughts);
        this.thoughts = thoughts;
    }

    public String getThoughts() {
        return thoughts;
    }
}
/

```

The `<aop:pointcut>` element has become the `@Pointcut` annotation and the `<aop:before>` element has become the `@Before` annotation. The only significant change here is that `@AspectJ` can lean on Java syntax to determine the details of the parameters passed into the advice. Therefore, there's no need for an annotation-based equivalent to the `<aop:before>` element's `arg-names`.

4.4.3 Annotating introductions

Earlier, I showed you how to use `<aop:declare-parents>` to introduce an interface onto an existing bean without changing the bean's source code. Now let's have another look at that example, but this time using annotation-based AOP.

The annotation equivalent of `<aop:declare-parents>` is `@AspectJ`'s `@DeclareParents`. `@DeclareParents` works almost exactly like its XML counterpart when used inside of an `@Aspect`-annotated class. The following shows how to use `@DeclareParents`.

Listing 4.8 Introducing the `Contestant` interface using `@AspectJ` annotations

```

package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class ContestantIntroducer {

    @DeclareParents(
        value = "com.springinaction.springidol.Performer+",
        defaultImpl = GraciousContestant.class)
    public static Contestant contestant;
}

```

Mix in
Contestant
interface

As you can see, `ContestantIntroducer` is an aspect. But unlike the aspects we've created so far, it doesn't provide before, after, or around advice. Instead, it introduces the `Contestant` interface onto `Performer` beans. Like `<aop:declare-parents>`, `@DeclareParents` annotation is made up of three parts:

- The `value` attribute is equivalent to `<aop:declare-parents>`'s `types-matching` attribute. It identifies the kinds of beans that should be introduced with the interface.

- The `defaultImpl` attribute is equivalent to `<aop:declare-parents>`'s `default-impl` attribute. It identifies the class that will provide the implementation for the introduction.
- The static property that is annotated by `@DeclareParents` specifies the interface that is to be introduced.

As with any aspect, you'll need to declare `ContestantIntroducer` as a bean in the Spring application context:

```
<bean class="com.springinaction.springidol.ContestantIntroducer" />
```

`<aop:aspectj-autoproxy>` will take it from there. When it discovers a bean annotated with `@Aspect`, it'll automatically create a proxy that delegates calls to either the proxied bean or to the introduction implementation, depending on whether the method called belongs to the proxied bean or to the introduced interface.

One thing you'll notice is that `@DeclareParents` doesn't have an equivalent to `<aop:declare-parents>`'s `delegate-ref` attribute. That's because `@DeclareParents` is an `@AspectJ` annotation. `@AspectJ` is a project that's separate from Spring and thus its annotations aren't bean-aware. The implications here are that if you want to delegate to a bean that's configured with Spring, then `@DeclareParents` may not fit the bill and you'll have to resort to using `<aop:declare-parents>`.

Spring AOP enables separation of cross-cutting concerns from an application's business logic. But as we've seen, Spring aspects are still proxy-based and are limited to advising method invocations. If you need more than just method proxy support, you'll want to consider using AspectJ. In the next section, you'll see how traditional AspectJ aspects can be used within a Spring application.

4.5 Injecting AspectJ aspects

Although Spring AOP is sufficient for many applications of aspects, it's a weak AOP solution when contrasted with AspectJ. AspectJ offers many types of pointcuts that aren't possible with Spring AOP.

Constructor pointcuts, for example, are convenient when you need to apply advice upon the creation of an object. Unlike constructors in some other object-oriented languages, Java constructors are different from normal methods. This makes Spring's proxy-based AOP woefully inadequate for advising creation of an object.

For the most part, AspectJ aspects are independent of Spring. Although they can be woven into any Java-based application, including Spring applications, there's little involvement on Spring's part in applying AspectJ aspects.

But any well-designed and meaningful aspect will likely depend on other classes to assist in its work. If an aspect depends on one or more classes when executing its advice, you can instantiate those collaborating objects with the aspect itself. Or, better yet, you can use Spring's dependency injection to inject beans into AspectJ aspects.

To illustrate, let's create a new aspect for the *Spring Idol* competition. A talent competition needs a judge. So, let's create a judge aspect in AspectJ. `JudgeAspect` is such an aspect.

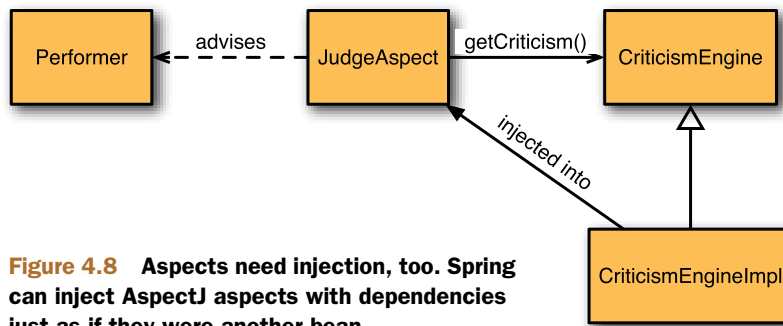


Figure 4.8 Aspects need injection, too. Spring can inject AspectJ aspects with dependencies just as if they were another bean.

Listing 4.9 An AspectJ implementation of a talent competition judge

```

package com.springinaction.springidol;

public aspect JudgeAspect {
    public JudgeAspect() {}

    pointcut performance() : execution(* perform(..));

    after() returning() : performance() {
        System.out.println(criticismEngine.getCriticism());
    }

    // injected
    private CriticismEngine criticismEngine;
    public void setCriticismEngine(CriticismEngine criticismEngine) {
        this.criticismEngine = criticismEngine;
    }
}

```

The chief responsibility for `JudgeAspect` is to make commentary on a performance after the performance has completed. The `performance()` pointcut in listing 4.9 matches the `perform()` method. When it's married with the `after()returning()` advice, you get an aspect that reacts to the completion of a performance.

What makes listing 4.9 interesting is that the judge doesn't make commentary on its own. Instead, `JudgeAspect` collaborates with a `CriticismEngine` object, calling its `getCriticism()` method, to produce critical commentary after a performance. To avoid unnecessary coupling between `JudgeAspect` and the `CriticismEngine`, the `JudgeAspect` is given a reference to a `CriticismEngine` through setter injection. This relationship is illustrated in figure 4.8.

`CriticismEngine` itself is an interface that declares a simple `getCriticism()` method. Here's the implementation of `CriticismEngine`.

Listing 4.10 An implementation of the `CriticismEngine` used by `JudgeAspect`

```

package com.springinaction.springidol;

public class CriticismEngineImpl implements CriticismEngine {
    public CriticismEngineImpl() {}
}

```

```

public String getCriticism() {
    int i = (int) (Math.random() * criticismPool.length);

    return criticismPool[i];
}

// injected
private String[] criticismPool;
public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
}
}

```

CriticismEngineImpl implements the CriticismEngine interface by randomly choosing a critical comment from a pool of injected criticisms. This class can be declared as a Spring <bean> using the following XML:

```

<bean id="criticismEngine"
      class="com.springinaction.springidol.CriticismEngineImpl">
  <property name="criticisms">
    <list>
      <value>I'm not being rude, but that was appalling.</value>
      <value>You may be the least talented
        person in this show.</value>
      <value>Do everyone a favor and keep your day job.</value>
    </list>
  </property>
</bean>

```

So far, so good. You now have a CriticismEngine implementation to give to JudgeAspect. All that's left is to wire CriticismEngineImpl into JudgeAspect.

Before I show you how to do the injection, you should know that AspectJ aspects can be woven into your application without involving Spring at all. But if you want to use Spring's dependency injection to inject collaborators into an AspectJ aspect, you'll need to declare the aspect as a <bean> in Spring's configuration. The following <bean> declaration injects the criticismEngine bean into JudgeAspect:

```

<bean class="com.springinaction.springidol.JudgeAspect"
      factory-method="aspectOf">
  <property name="criticismEngine" ref="criticismEngine" />
</bean>

```

For the most part, this <bean> declaration isn't much different from any other <bean> you may find in Spring. But the big difference is the use of the factory-method attribute. Normally Spring beans are instantiated by the Spring container, but AspectJ aspects are created by the AspectJ runtime. By the time Spring gets a chance to inject the CriticismEngine into JudgeAspect, JudgeAspect has already been instantiated.

Since Spring isn't responsible for the creation of JudgeAspect, it isn't possible to simply declare JudgeAspect as a bean in Spring. Instead, we need a way for Spring to get a handle to the JudgeAspect instance that has already been created by AspectJ so that we can inject it with a CriticismEngine. Conveniently, all AspectJ aspects provide

a static `aspectOf()` method that returns the singleton instance of the aspect. So to get an instance of the aspect, you must use `factory-method` to invoke the `aspectOf()` method instead of trying to call `JudgeAspect`'s constructor.

In short, Spring doesn't use the `<bean>` declaration from earlier to create an instance of the `JudgeAspect`—it has already been created by the `AspectJ` runtime. Instead, Spring retrieves a reference to the aspect through the `aspectOf()` factory method and then performs dependency injection on it as prescribed by the `<bean>` element.

4.6 **Summary**

AOP is a powerful complement to object-oriented programming. With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules. You can then declare exactly where and how this behavior is applied. This reduces code duplication and lets your classes focus on their main functionality.

Spring provides an AOP framework that lets you insert aspects around method executions. You've learned how you can weave advice before, after, and around a method invocation, as well as add custom behavior for handling exceptions.

You have several choices in how you can use aspects in your Spring applications. Wiring advice and pointcuts in Spring is much easier with the addition of `@AspectJ` annotation support and a simplified configuration schema.

Finally, there are times when Spring AOP isn't powerful enough and you must turn to `AspectJ` for more powerful aspects. For those situations, we looked at how to use Spring to inject dependencies into `AspectJ` aspects.

At this point, we've covered the basics of the Spring Framework. You've seen how to configure the Spring container and how to apply aspects to Spring-managed objects. As you've seen, these core techniques offer great opportunity to create applications composed of loosely coupled objects. In the next chapter, we'll look at how loose coupling through DI and AOP foster developer-driven testing and see how to keep your Spring code covered by tests.