***Creating annotated 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.

You’ve already defined the Performance interface as the subject of your aspect’s pointcuts. Now let’s use AspectJ annotations to create an aspect.

***Defining an aspect***

**package** com.aop;

**import** org.aspectj.lang.annotation.AfterReturning;

**import** org.aspectj.lang.annotation.AfterThrowing;

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

**import** org.aspectj.lang.annotation.Before;

@Aspect

**public** **class** Audiance {

@Before("execution(\*\* com.aop.Performance.perform(..))")

**public** **void** silenceCellPhones() {

System.***out***.println("Silencing Cell Phones");

}

@Before("execution( \*\* com.aop.Performance.perform(..))")

**public** **void** takeSeats() {

System.***out***.println("Taking Seats");

}

@AfterReturning("execution(\*\* com.aop.Performance.perform(..))")

**public** **void** applause() {

System.***out***.println("Clap Clap Clap");

}

@AfterThrowing("execution(\*\* com.aop.Performance.perform(..))")

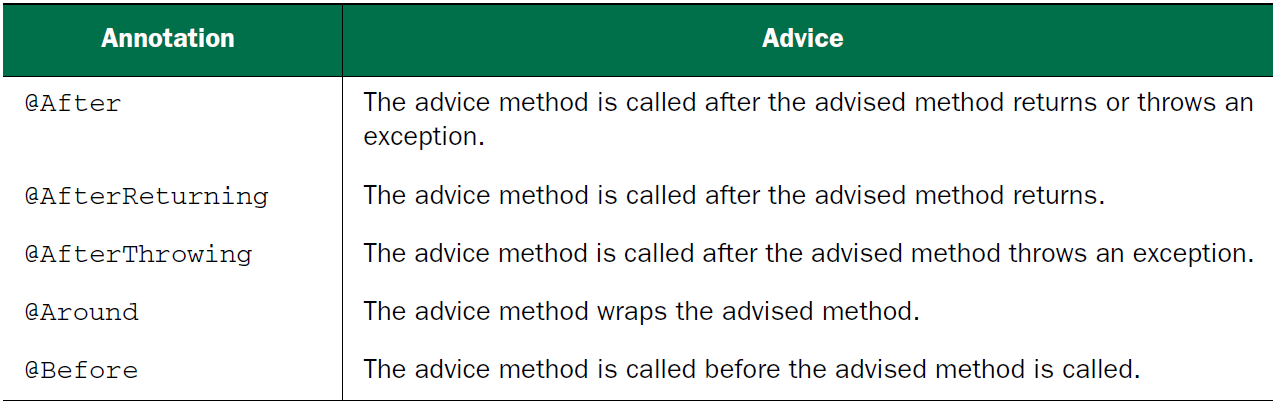
**public** **void** demandRefund() {

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

}

}

* Notice how the Audience class is annotated with @Aspect. This annotation indicates that Audience isn’t just any POJO—it’s an aspect. And throughout the Audience class are methods that are annotated to define the specifics of the aspect.
* Audience has four methods that define things an audience might do as it observes a performance. Before the performance, the audience should take their seats (takeSeats()) and silence their cell phones (silenceCellPhones()).
* If the performance goes well, the audience should applaud (applause()). But if the performance fails to meet the audience’s expectations, then the audience should demand a refund (demandRefund()).
* Those methods are annotated with advice annotations to indicate when those methods should be called. AspectJ provides five annotations for defining advice.



* The Audience class makes use of three out of the five advice annotations.
* The takeSeats()and silenceCellPhones() methods are both annotated with @Before, indicating that they should be called before a performance is performed.
* The applause() method is annotated with @AfterReturning so that it will be called after a performance returns successfully.
* And the @AfterThrowing annotation is placed on demandRefund() so that it will be called if any exceptions are thrown during a performance.

**Note:**

* You’ve probably noticed that all of these annotations are given a pointcut expression as a value.
* And you may have noticed that it’s the same pointcut expression on all four methods.
* They could each be given a different pointcut expression, but this particular pointcut suits your needs for all the advice methods.
* Taking a closer look at the pointcut expression given to the advice annotations, you’ll see that it triggers on the execution of the perform() method on a Performance.
* It’s a shame that you had to repeat that same pointcut expression four times. Duplication like this doesn’t feel right. It’d be nice if you could define the pointcut once and then reference it every time you need it.
* Fortunately, there’s a way: the @Pointcut annotation defines a reusable pointcut within a @AspectJ aspect. The next listing shows the Audience aspect, updated to use @Pointcut.

**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** Audiance {

@Pointcut("execution (\*\* com.aop.Performance.perform(..))")

**public** **void** performance() {}

@Before("performance()")

**public** **void** silenceCellPhones() {

System.***out***.println("Silencing Cell Phones");

}

@Before("performance()")

**public** **void** takeSeats() {

System.***out***.println("Taking Seats");

}

@AfterReturning("performance()")

**public** **void** applause() {

System.***out***.println("Clap Clap Clap");

}

@AfterThrowing("performance()")

**public** **void** demandRefund() {

System.***out***.println("Slap Slap Slap...Give My Money Back");

}

}

* In Audience, the performance() method is annotated with @Pointcut. The value given to the @Pointcut annotation is a pointcut expression, just like the ones you used previously with the advice annotations. By annotating performance() with @Pointcut in this way, you essentially extend the pointcut expression language so that you can use performance() in your pointcut expressions anywhere you’d otherwise use the longer expression. As you can see, you replace the longer expression in all the advice annotations with performance().
* The 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.
* Note that aside from the annotations and the no-op performance() method, the Audience class is essentially a POJO. Its methods can be called just like methods on any other Java class. Its methods can be individually unit-tested just as in any other Java class. Audience is just another Java class that happens to be annotated to be used as an aspect.
* And, just like any other Java class, it can be wired as a bean in Spring:

@Bean

**public** Audiance audiance() {

**return** **new** Audiance();

}

* If you were to stop here, Audience would only be a bean in the Spring container. Even though it’s annotated with AspectJ annotations, it wouldn’t be treated as an aspect without something that interpreted those annotations and created the proxies that turn it into an aspect.
* If you’re using JavaConfig, you can turn on auto-proxying by applying the

@EnableAspectJAutoProxy annotation at the class level of the configuration class.

@Configuration

@EnableAspectJAutoProxy

**public** **class** Config {

@Bean

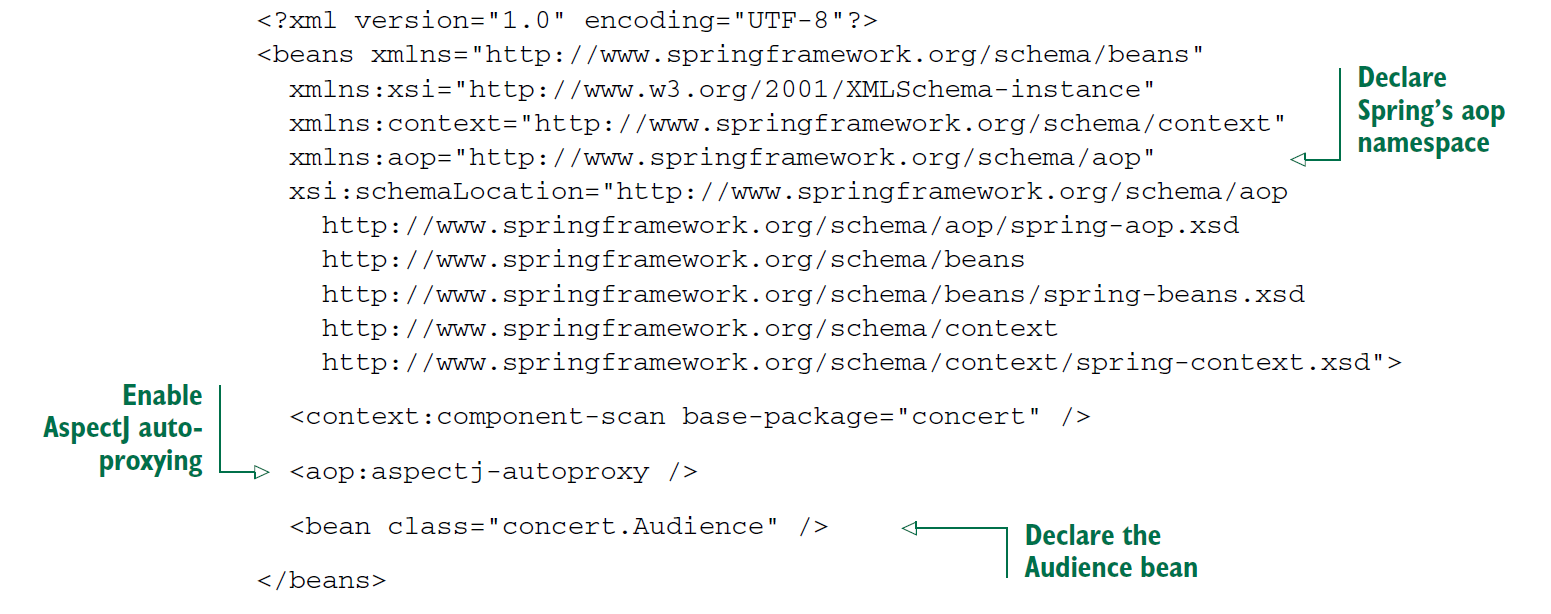
**public** Audiance audiance() {

**return** **new** Audiance();

}

}

* If, however, you’re using XML to wire your beans in Spring, then you need to use the <aop:aspectj-autoproxy> element from Spring’s aop namespace. The XML configuration in the following listing shows how this is done.

****

* Whether you use JavaConfig or XML, AspectJ auto-proxying uses the @Aspect annotated bean to create a proxy around any other beans for which the aspect’s pointcuts are a match.
* It’s important to understand that Spring’s AspectJ auto-proxying only uses @AspectJ annotations as a guide for creating proxy-based aspects. Under the covers, it’s still Spring’s proxy-based aspects. This is significant because it means that although you’re using @AspectJ annotations, you’re still limited to proxying method invocations.

***Creating around advice***

* Around advice is the most powerful advice type. It allows you to write logic that completely wraps the advised method. It’s essentially like writing both before advice and after advice in a single advice method.

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

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

**import** org.aspectj.lang.annotation.Pointcut;

@Aspect

**public** **class** Audiance {

@Pointcut("execution (\*\* com.aop.Performance.perform(..))")

**public** **void** performance() {}

@Around("performance()")

**public** **void** watchPerformace(ProceedingJoinPoint jp) {

**try** {

System.***out***.println("Silencing Cell Phones");

System.***out***.println("Taking Seats");

jp.proceed();

System.***out***.println("Clap Clap Clap");

} **catch** (Throwable e) {

System.***out***.println("Slap Slap Slap...Give My Money Back");

}

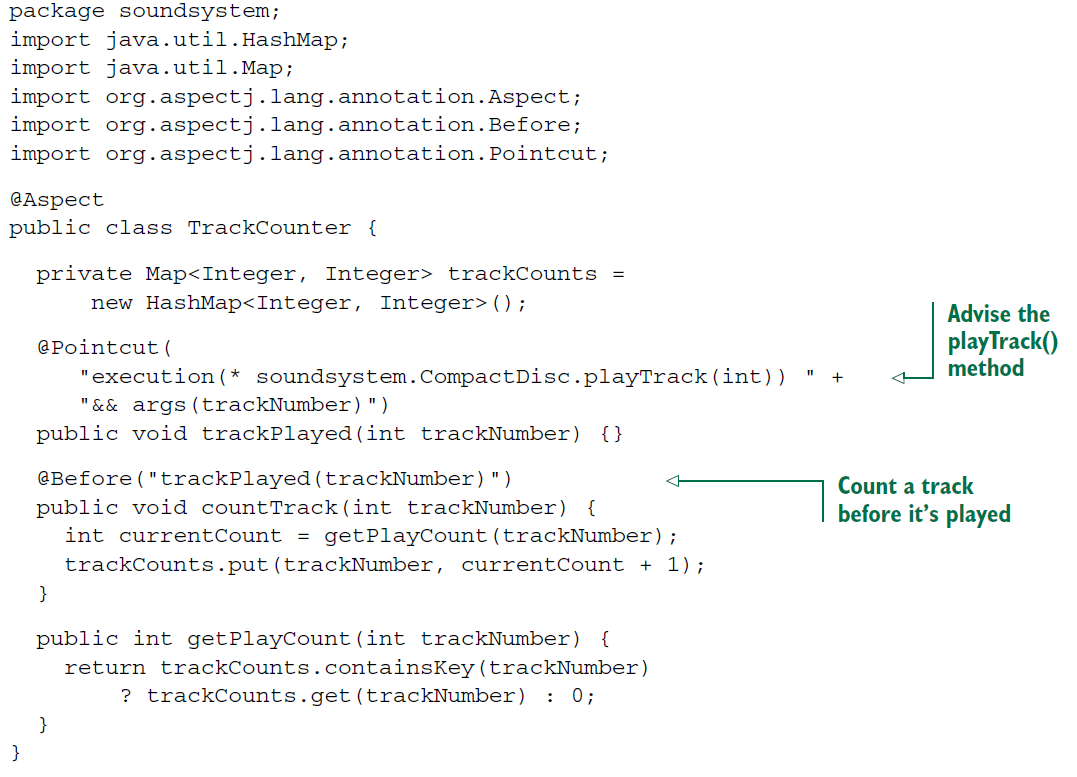
}

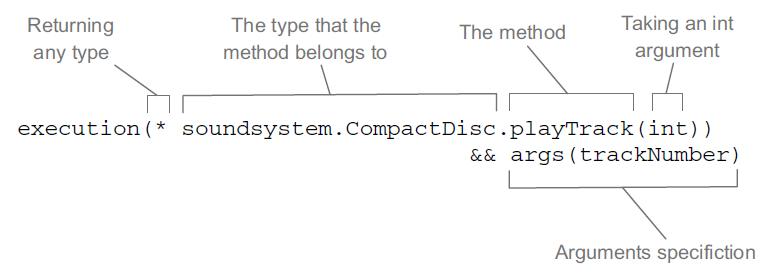
}

* Here the @Around annotation indicates that the watchPerformance() method is to be applied as around advice to the performance() pointcut.
* In this advice, the audience will silence their cell phones and take their seats before the performance and will applaud after the performance. And just like before, if an exception is thrown during the performance, the audience will ask for their money back.
* As you can see, the effect of this advice is identical to what you did earlier with before and after advice. But here it’s all in a single advice method, whereas before it was spread across four distinct advice methods.
* 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 because it’s how you can invoke the advised method from within your advice. The advice method will do everything it needs to do; and when it’s ready to pass control to the advised method, it will 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.

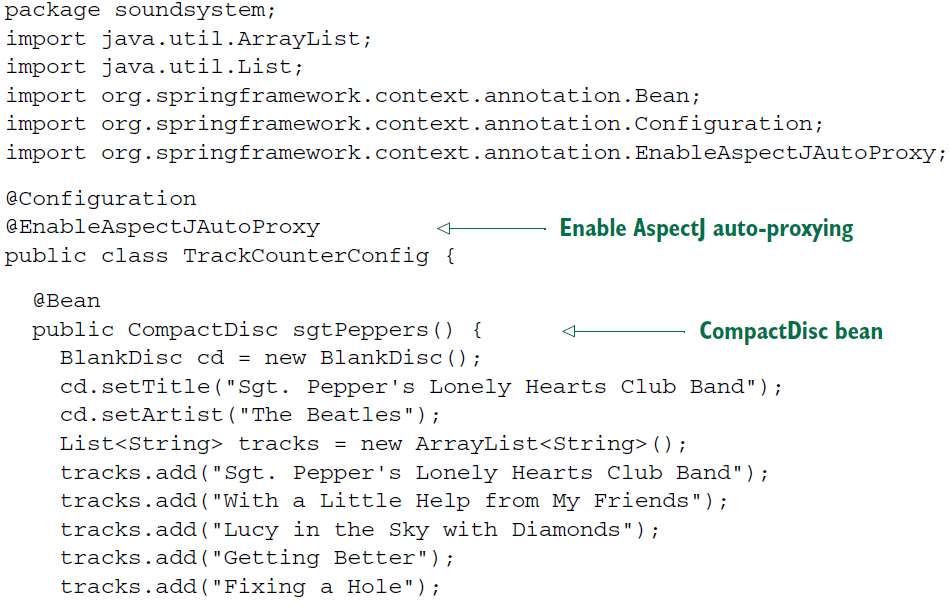
***Handling parameters in advice***

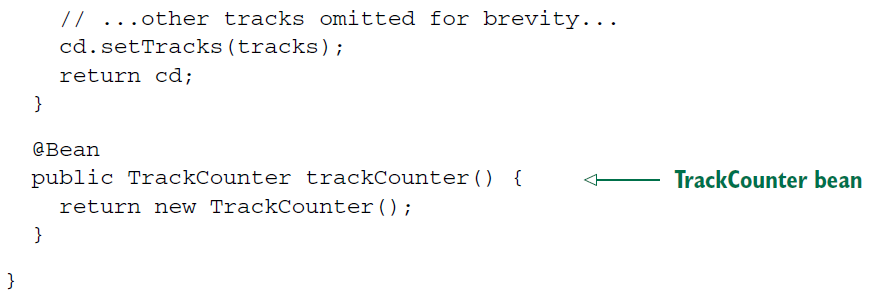
* What if your aspect was to advise a method that does take parameters? Could the aspect access the parameters that are passed into the method and use them?
* To illustrate, let’s revisit the BlankDisc class, the play()method cycles through all the tracks and calls playTrack() for each track. But you could call the playTrack() method directly to play an individual track.
* Suppose you want to keep a count of how many times each track is played. One way to do this is to change the playTrack() method to directly keep track of that count each time it’s called. But track-counting logic is a separate concern from playing a track and therefore doesn’t belong in the playTrack() method. This looks like a job for an aspect.
* To keep a running count of how many times a track is played, let’s create Track-Counter, an aspect that advises playTrack(). The following listing shows just such an aspect.

****

****

* As with the other aspects you’ve created so far, this aspect uses @Pointcut to define a named pointcut and @Before to declare a method as supplying before advice.
* What’s different here, however, is that the pointcut also declares parameters to be supplied to the advice method.
* The thing to focus on in the figure is the args(trackNumber) qualifier in the pointcut expression.
* This indicates that any int argument that is passed into the execution of playTrack() should also be passed into the advice.
* The parameter name, trackNumber, also matches the parameter in the pointcut method signature.
* That carries over into the advice method where the @Before annotation is defined with the named pointcut, trackPlayed(trackNumber). The parameter in the pointcut aligns with the parameter of the same name in the pointcut method, completing the path of the parameter from the named pointcut to the advice method.
* Now you can configure BlankDisc and TrackCounter as beans in the Spring configuration and enable AspectJ auto-proxying, as shown next:

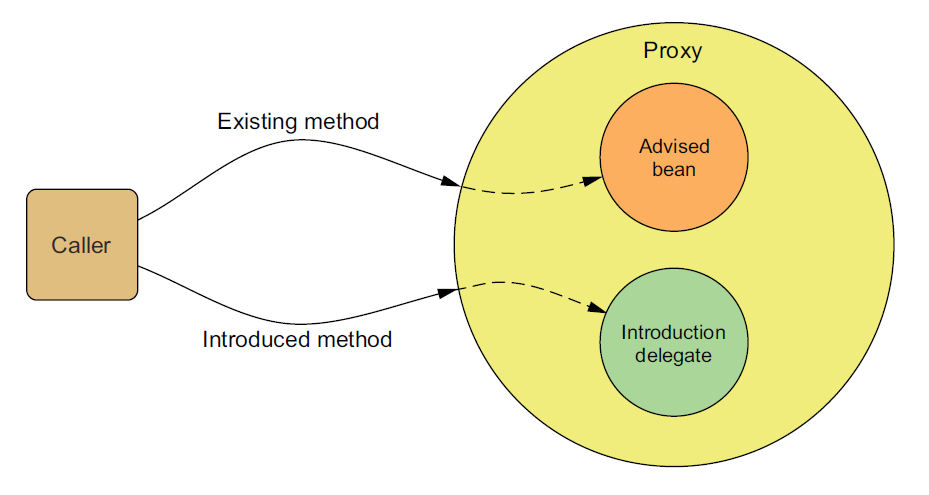
****

****

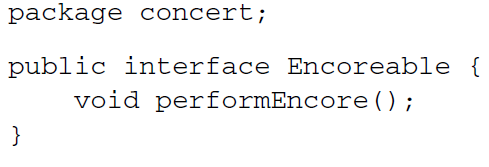
***Annotating introductions***

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 or classes. Unfortunately, Java isn’t that dynamic. Once a class has been compiled, there’s little you can do to append new functionality to it.

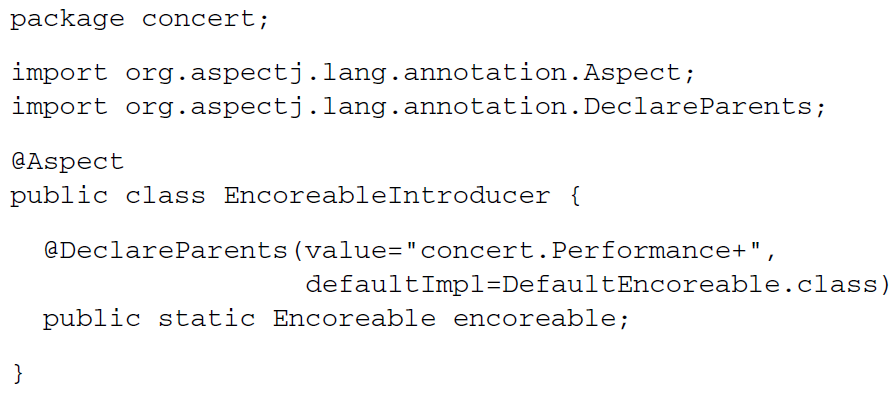
* But if you think about it, isn’t that what you’ve been doing in this chapter with aspects? Sure, you haven’t added any new methods to objects, but you’re adding new functionality around the methods that the objects already have. If an aspect can wrap existing methods with additional functionality, why not add new methods to the object?
* In fact, using an AOP concept known as *introduction*, aspects can attach new methods to Spring beans.
* Recall that in Spring, aspects are proxies that implement the same interfaces as the beans they wrap. What if, in addition to implementing those interfaces, the proxy is also 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.

****

* Notice 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 you one bean whose implementation is split across multiple classes.
* Putting this idea to work, let’s say you want to introduce the following Encoreable interface to any implementation of Performance:

****

* You need a way to apply this interface to your Performance implementations. You could visit all implementations of Performance and change them so that they also implement Encoreable. But from a design standpoint, that may not be the best move. Not all Performances will necessarily be Encoreable.
* Moreover, it may not be possible to change all implementations of Performance, especially if you’re working with third-party implementations and don’t have the source code.
* Fortunately, AOP introductions can help you without compromising design choices or requiring invasive changes to the existing implementations. To pull it off, you create a new aspect:

****

* As you can see, EncoreableIntroducer is an aspect. But unlike the aspects you’ve created so far, it doesn’t provide before, after, or around advice. Instead, it introduces the Encoreable interface to Performance beans using the @DeclareParents annotation.
* The @DeclareParents annotation is made up of three parts:
* The value attribute identifies the kinds of beans that should be introduced with the interface. In this case, that’s anything that implements the Performance interface. (The plus sign at the end specifies any subtype of Performance, as opposed to Performance itself.)
* The defaultImpl attribute identifies the class that will provide the implementation for the introduction. Here you’re saying that DefaultEncoreable will provide that implementation.
* The static property that is annotated by @DeclareParents specifies the interface that’s to be introduced. In this case, you’re introducing the Encoreable interface.
* As with any aspect, you need to declare EncoreableIntroducer as a bean in the Spring application context:

**aop.png**

* Spring auto-proxying will take it from there. When Spring discovers a bean annotated with @Aspect, it will 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.
* Annotations and auto-proxying provide a convenient programming model for creating aspects in Spring. It’s simple and involves only minimal Spring configuration.
* But annotation-oriented aspect declaration has one clear disadvantage: you must be
* able to annotate the advice class. And to do that, you must have the source code.