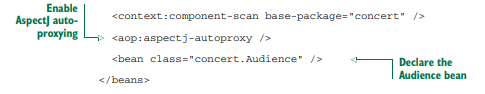
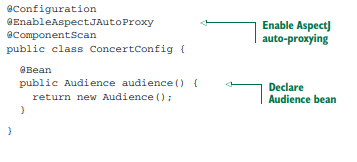
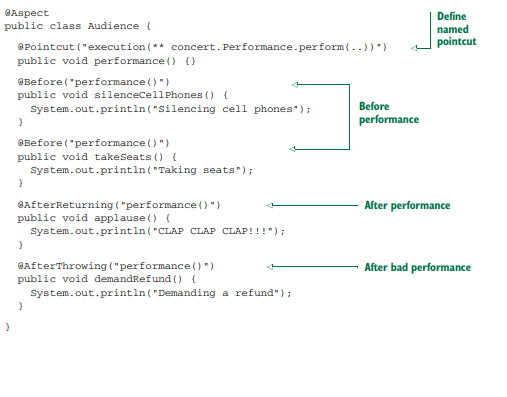


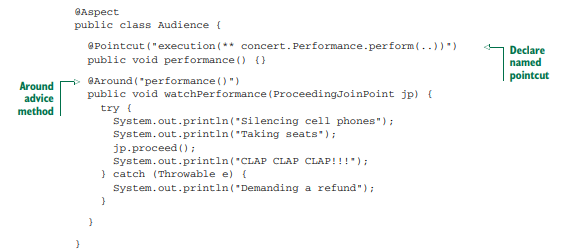
Notice how the Audience class is annotated with @Aspect. This annotation indicates that Audience isn’t just any POJO—it’s an aspect. And throughout the Audience class are methods that are annotated to define the specifics of the aspect.

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.

@Pointcut annotation defines a reusable pointcut within an @AspectJ aspect.

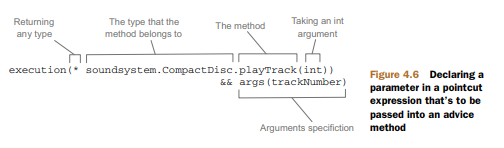
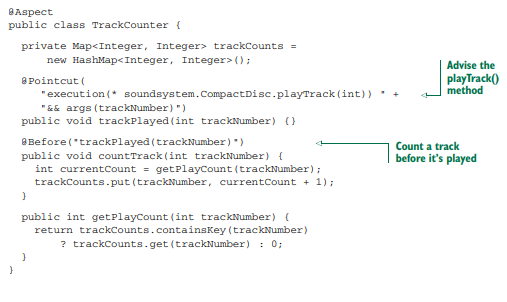


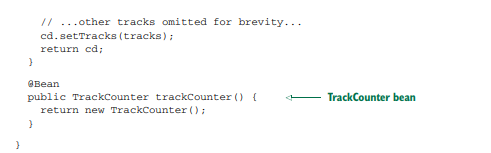
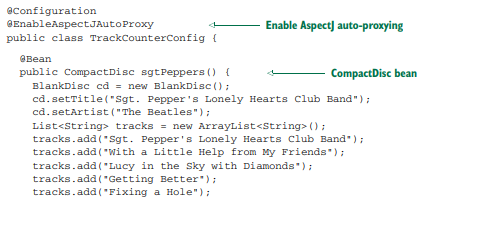
AspectJ auto-proxying uses the @Aspectannotated bean to create a proxy around any other beans for which the aspect’s pointcuts are a match. In this case, a proxy will be created for the Concert bean, with the advice methods in Audience being applied before and after the perform() method.



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.

Using parameterized advice to count how many times a track is played-





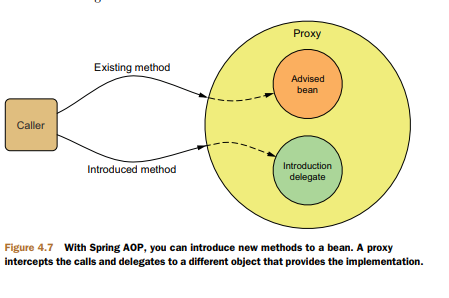
Test



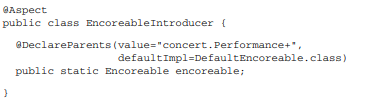
Annotating Introduction:

Using an AOP concept known as introduction, aspects can attach new methods to Spring beans.

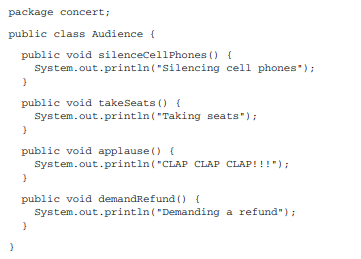
Recall that in Spring, aspects are proxies that implement the same interfaces as the beans they wrap. What if, in addition to implementing those interfaces, the proxy is also exposed through some new interface? Then any bean that’s advised by the aspect will appear to implement the new interface, even if its underlying implementation class doesn’t.

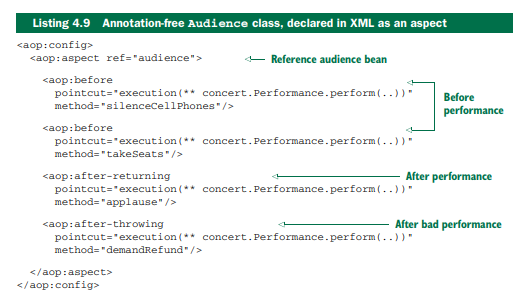


Notice that when a method on the introduced interface is called, the proxy delegates the call to some other object that provides the implementation of the new interface. Effectively, this gives you one bean whose implementation is split across multiple classes.

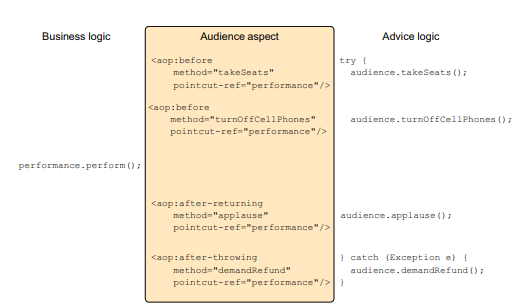


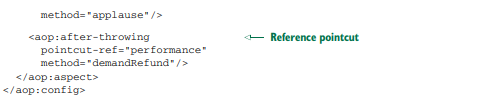
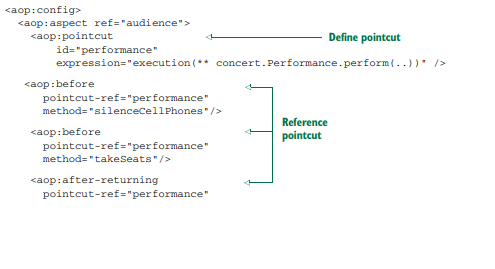
Declaring Aspects in XML:





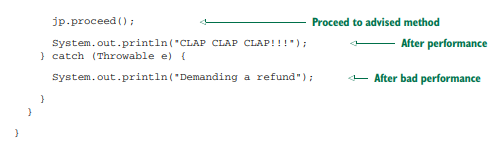
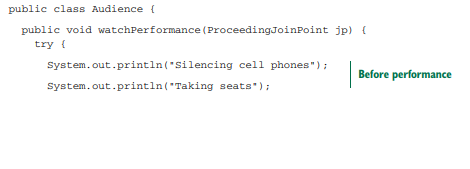
This makes Spring’s XML configuration for AOP a handy way to use types defined in third-party libraries as advice, even though you can’t annotate them with AspectJ aspects.

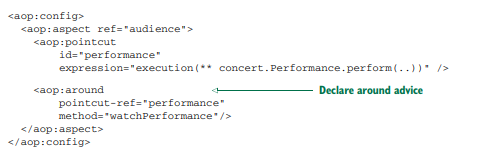




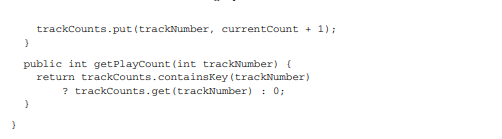
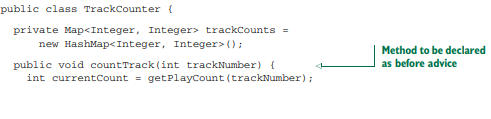
It’s tricky to share information between before advice and after advice without resorting to storing that information in member variables.

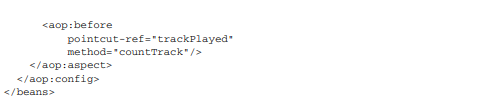
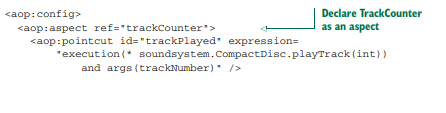
Around advice -the entire set of advice takes place in a single method, there’s no need to retain state in a member variable.



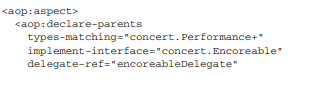
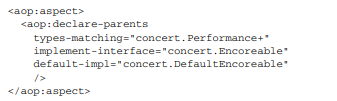


Passing parameters to advice:





Introductions with XML configuration:

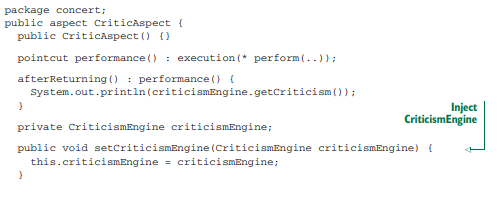


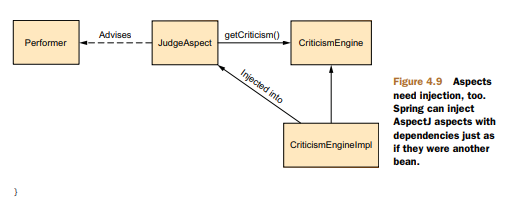


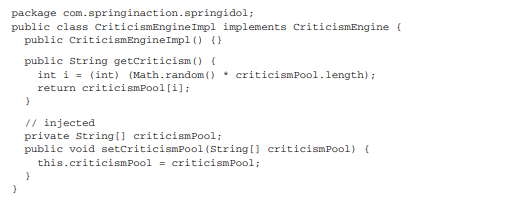
Injecting AspectJ aspects

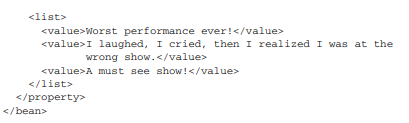
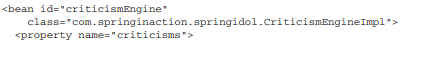
AspectJ offers many types of pointcuts that aren’t possible with Spring AOP. Constructor pointcuts, for example, are convenient when you need to apply advice on the creation of an object.

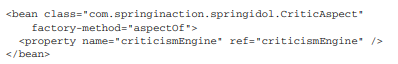
If an aspect depends on one or more classes when executing its advice, you can instantiate those collaborating objects with the aspect itself. Or, better yet, you can use Spring’s dependency injection to inject beans into AspectJ aspects.

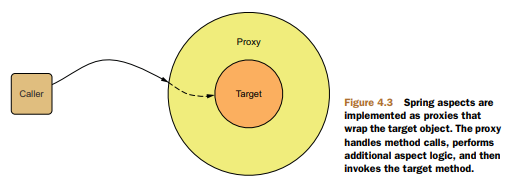












SPRING ADVISES OBJECTS AT RUNTIME In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. As illustrated in figure 4.3, the proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean. Between the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