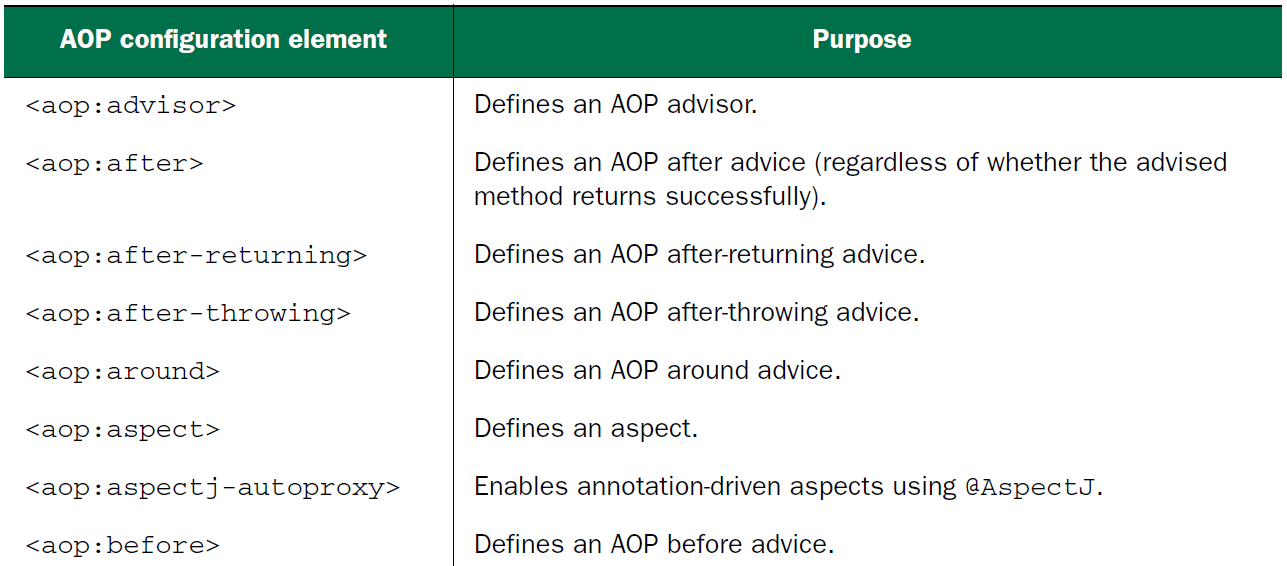
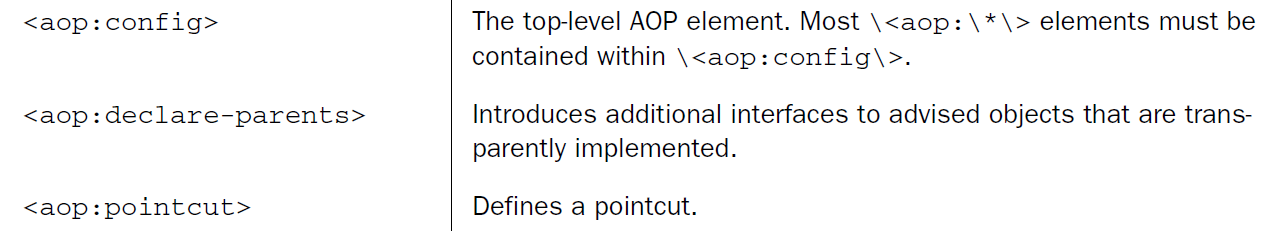
***Declaring aspects in XML***

When you don’t have the source code, or if you don’t want to place AspectJ annotations in your code, Spring offers another option for aspects. If you need to declare aspects without annotating the advice class, then you must turn to

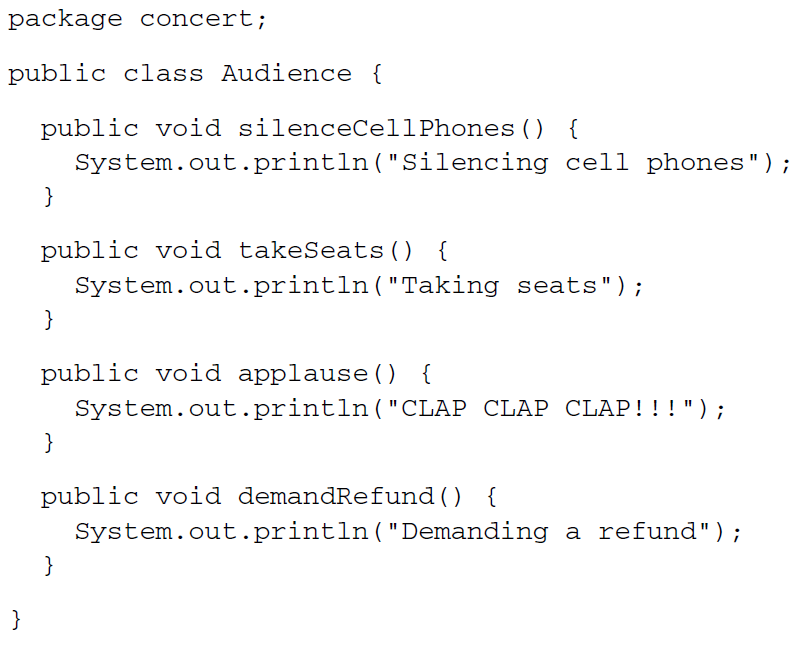
XML configuration.

Spring’s aop namespace offers several elements that are useful for declaring aspects in XML, as described in table:



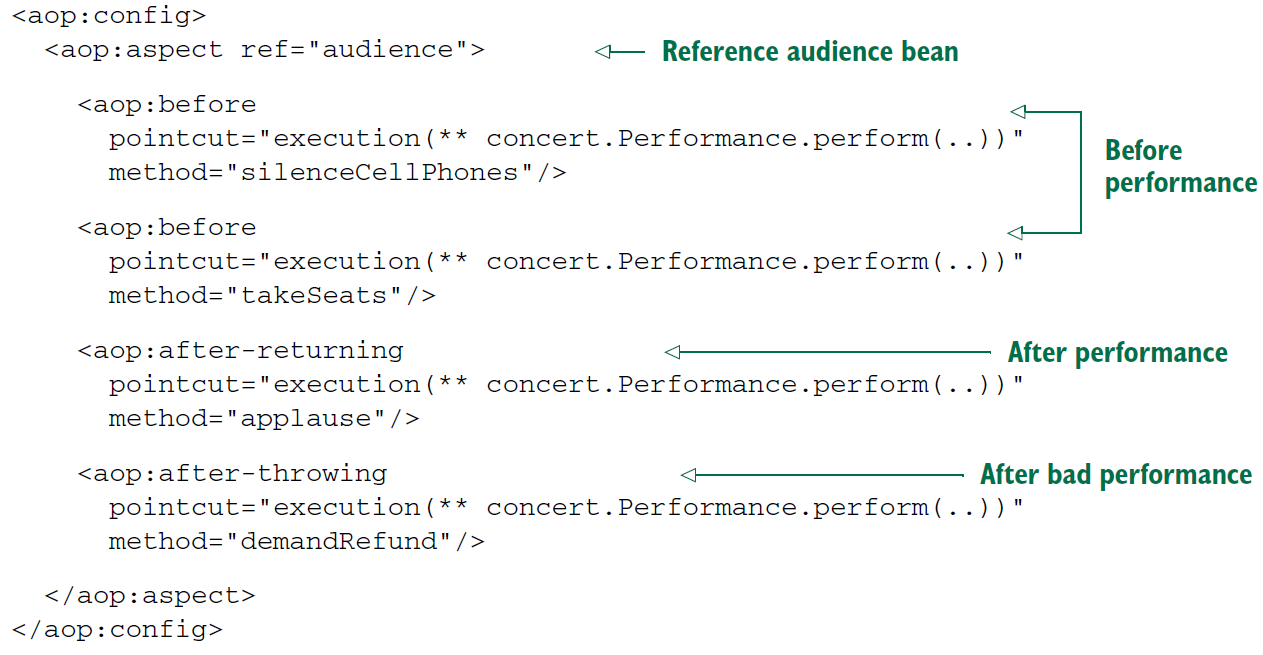


* We’ve already seen the <aop:aspectj-autoproxy> element and how it can enable auto-proxying of AspectJ-annotated advice classes. The other elements in the aop namespace let you declare aspects directly in your Spring configuration without using annotations.
* For example, let’s have another look at the Audience class. This time, let’s remove all of those AspectJ annotations:

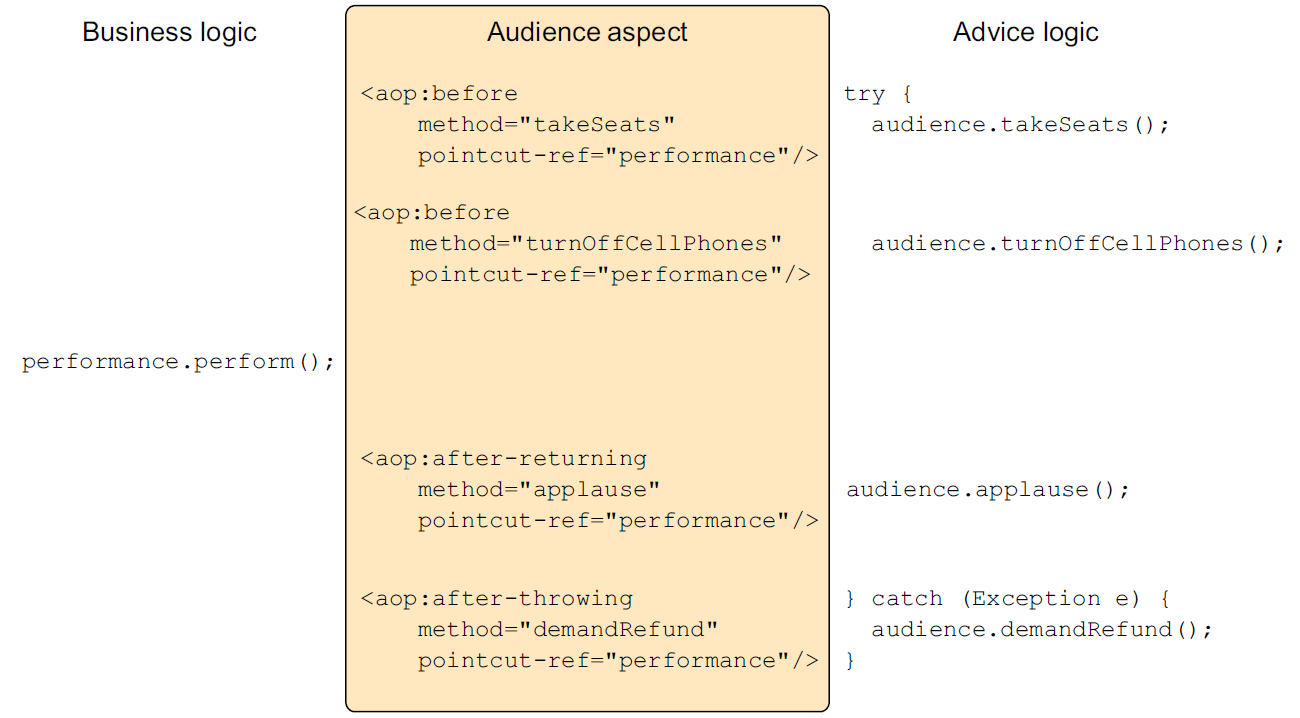


* As you can see, without the AspectJ annotations, there’s nothing remarkable about the Audience class. It’s a basic Java class with a handful of methods. And you can register it as a bean in the Spring application context like any other class.

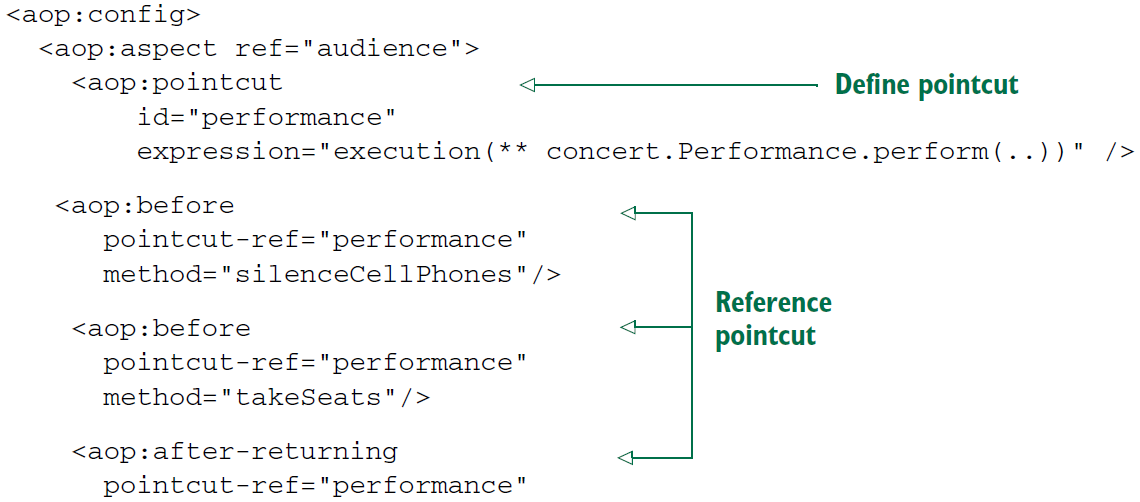
***Declaring before and after advice***

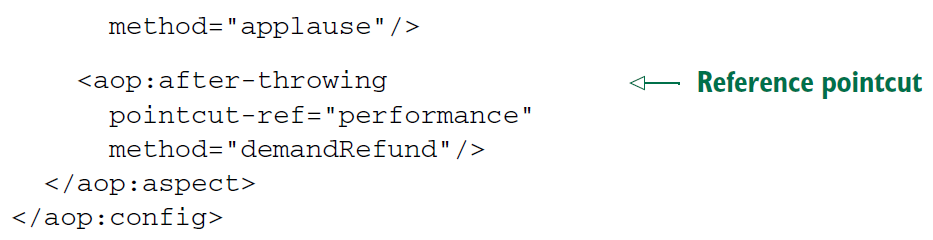


* The first thing to notice about the Spring AOP configuration elements is that most of them must be used in the context of the <aop:config> element.
* There are a few exceptions to this rule, but when it comes to declaring beans as aspects, you’ll aways start with <aop:config>.
* In <aop:config>, you may declare one or more advisers, aspects, or pointcuts.
* You declare 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.
* It’s worth noting that the referenced advice bean can be any type that provides methods to be called at the designated pointcuts. 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.
* The aspect has four different bits of advice. The two <aop:before> elements define before advice that will call the takeSeats() and silenceCellPhones() methods (declared by the method attribute) of the Audience bean before any methods matching the pointcut are executed.
* The <aop:after-returning> element defines after-returning advice to call the applause() 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.



* When you found the same kind of duplication in your AspectJ-annotated advice, you eliminated it by using the @Pointcut annotation. For XML-based aspect declarations, however, you’ll need to use the <aop:pointcut> element. The following XML shows how to extract the common pointcut expression into a single pointcut declaration that can be used across all advice elements.

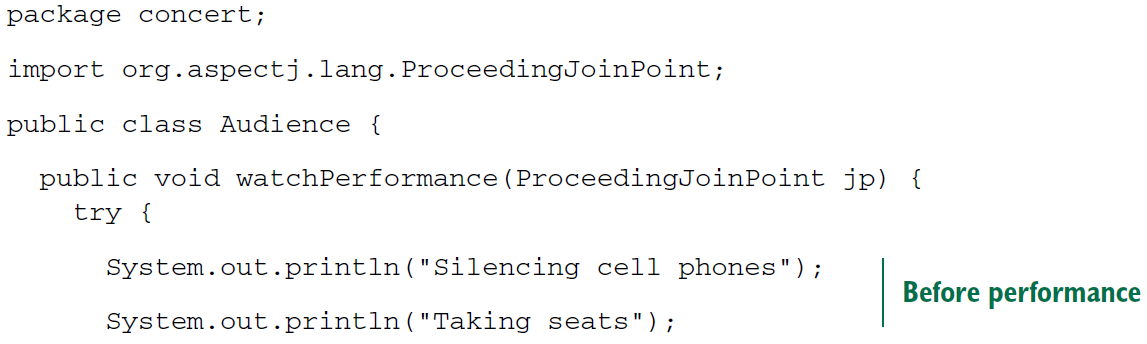


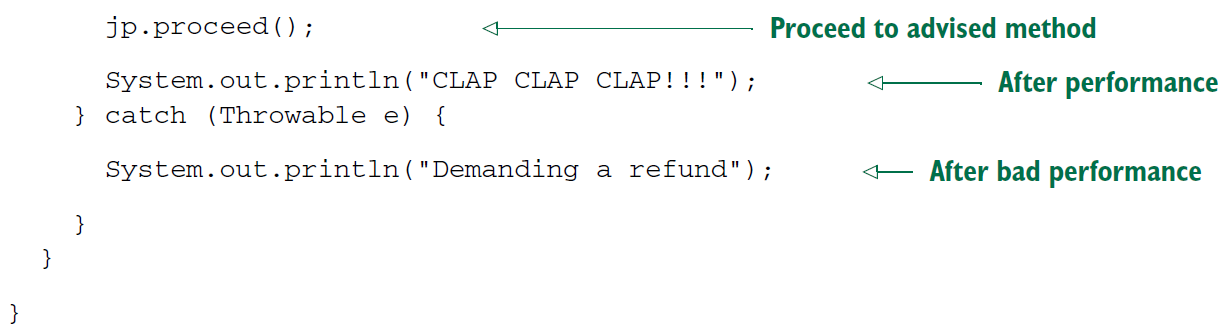


* Now the pointcut is defined in a single location and is referenced across multiple advice elements. The <aop:pointcut> element defines the pointcut to have an id of performance. Meanwhile, all the advice elements have been changed to reference the named pointcut with the pointcut-ref attribute.
* The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

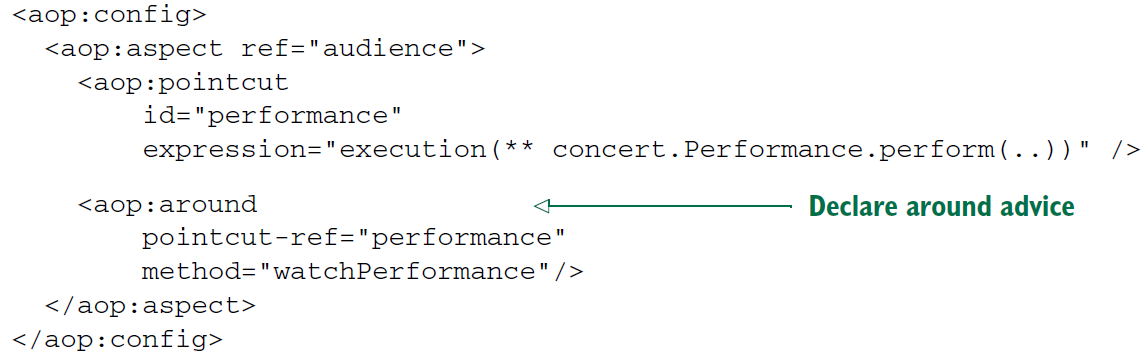
***Declaring around advice***

* 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 after advice. But you’d have to store the start time in a member variable. Because 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 you could with distinct before and after advice, but you can do it in a single method. Because 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 annotation-free Audience class with a single watchPerformance() method.





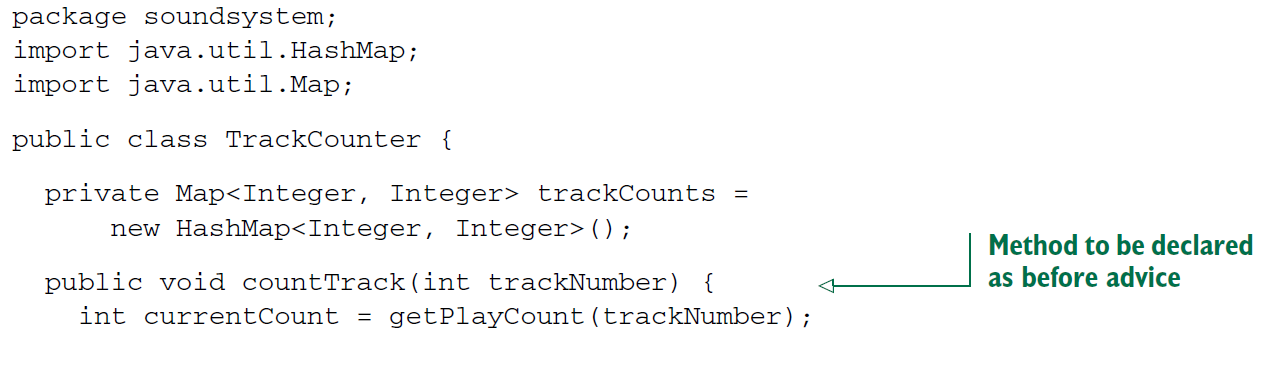
* In the case of the audience aspect, the watchPerformance() method contains all the functionality of the previous four advice methods. But all of it is contained in this single method, and this method is responsible for its own exception handling.
* Declaring around advice isn’t dramatically different from declaring other types of advice. All you need to do is use the <aop:around> element, as shown:

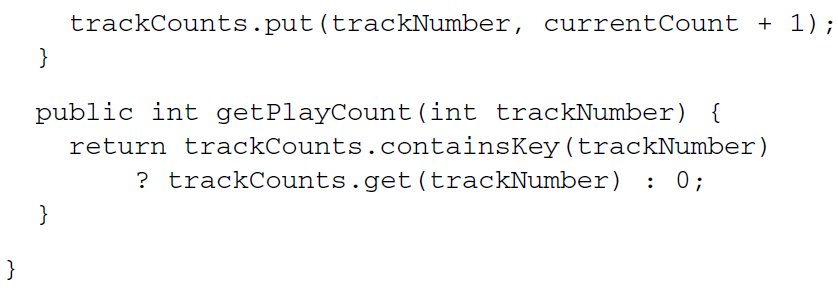


* As with the other advice XML elements, <aop:around> is given a pointcut and the name of an advice method. Here you’re using the same pointcut as before, but you set the method attribute to point to the new watchPerformance() method.

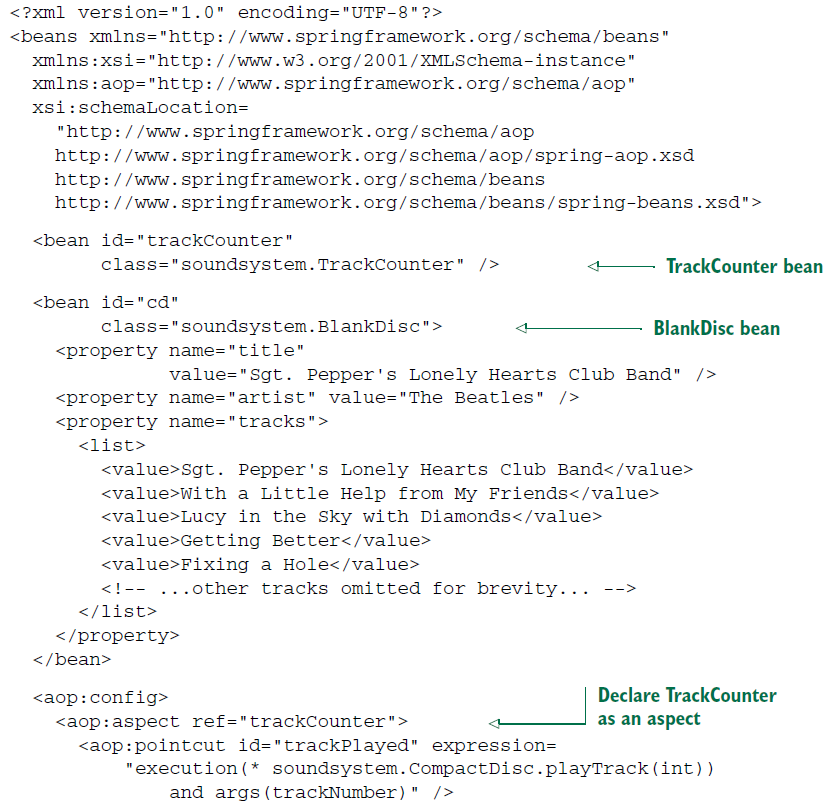
***Passing parameters to advice***

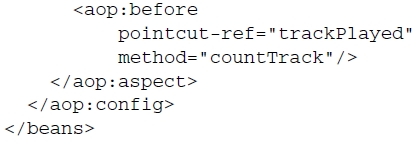
* Now that you’re configuring your aspects in XML, let’s see how you can accomplish the same thing. First, let’s strip all the @AspectJ annotations out of the TrackCounter.





* Without the AspectJ annotations, TrackCounter seems kind of bare. And as it stands now, TrackCounter won’t count any tracks unless you explicitly call the count-Track()method. But with a little XML Spring configuration, you can reinstate TrackCounter’s status as an aspect.
* The following listing shows the complete Spring configuration that declares both the TrackCounter bean and the BlankDisc bean and enables TrackCounter as an aspect:



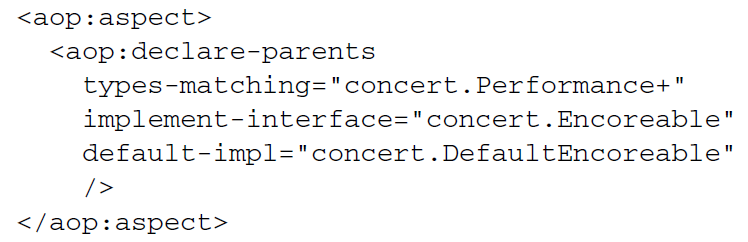


* As you can see, you’re using the same XML elements from the aop namespace as before; they declare a POJO to be treated as an aspect. The only significant difference is that your pointcut expression now includes a parameter to be passed into the advice method. If you compare this expression with the one from listing 4.6, you’ll see that they’re almost identical. The only real difference is that here you use the and keyword instead of && (because ampersands are interpreted as the beginning of an entity in XML).

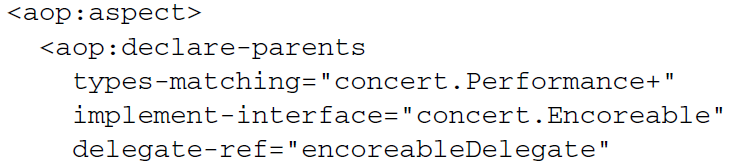
***Introducing new functionality with aspects***

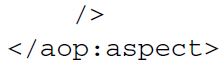
Earlier, we have seen how to use AspectJ’s @DeclareParents annotation to magically introduce a new method into an advised bean. But AOP introductions aren’t exclusive to AspectJ. Using the <aop:declare-parents> element from Spring’s aop namespace, you can do similar magic in XML.

The following snippet of XML is equivalent to the AspectJ-based introduction you created earlier:

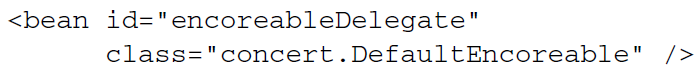


* As its name implies, <aop:declare-parents> declares that the beans it advises will have new parents in its object hierarchy. Specifically, in this case you’re saying that the beans whose type matches the Performance interface (per the types-matching attribute) should have Encoreable in their parentage (per the implement-interface attribute). The final matter to settle is where the implementation of the Encoreable’s methods will come from.
* There are two ways to identify the implementation of the introduced interface. In this case, you’re using the default-impl attribute to explicitly identify the implementation by its fully qualified class name. Alternatively, you could identify it using the delegate-ref attribute:





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



* 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.