***Core Spring***

* Spring does a lot of things. But underneath all of the fantastic functionality it adds to enterprise development, its primary features are dependency injection (DI) and aspect-oriented programming (AOP).
* Spring is an open source framework, originally created by Rod Johnson.
* Spring was created to address the complexity of enterprise application development and makes it possible to use simple JavaBeans to achieve things that were previously only possible with EJB.
* Any java application can benefit from spring in terms of simplicity, testability, and loose coupling.
* **Spring simplifies Java development.**
* Spring uses four key strategies:
  + Lightweight and minimally invasive development with POJOs or Java Beans
  + Loose coupling through DI and interface orientation
  + Declarative programming through aspects and common conventions
  + Eliminating boilerplate code with aspects and templates.
* Spring avoids (as much as possible) littering your application code with its API.
* Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class. Instead, the classes in a Spring-bases application often have no indication that they’re being used by Spring. At worst a class may be annotated with one of Springs’s annotation, but it’s otherwise a POJO.

***Injecting Dependencies***

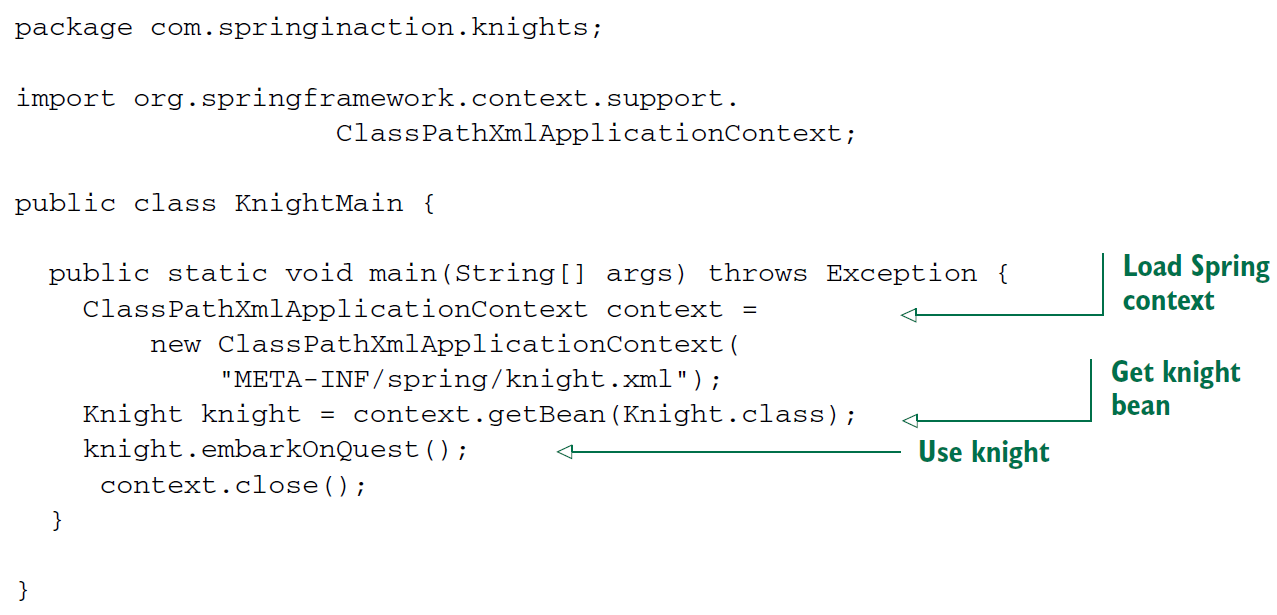
* The phrase *dependency injection* may sound intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, DI isn’t nearly as complex as it sounds. By applying DI in your projects, you’ll find that your code will become significantly simpler, easier to understand, and easier to test.

**How DI works**

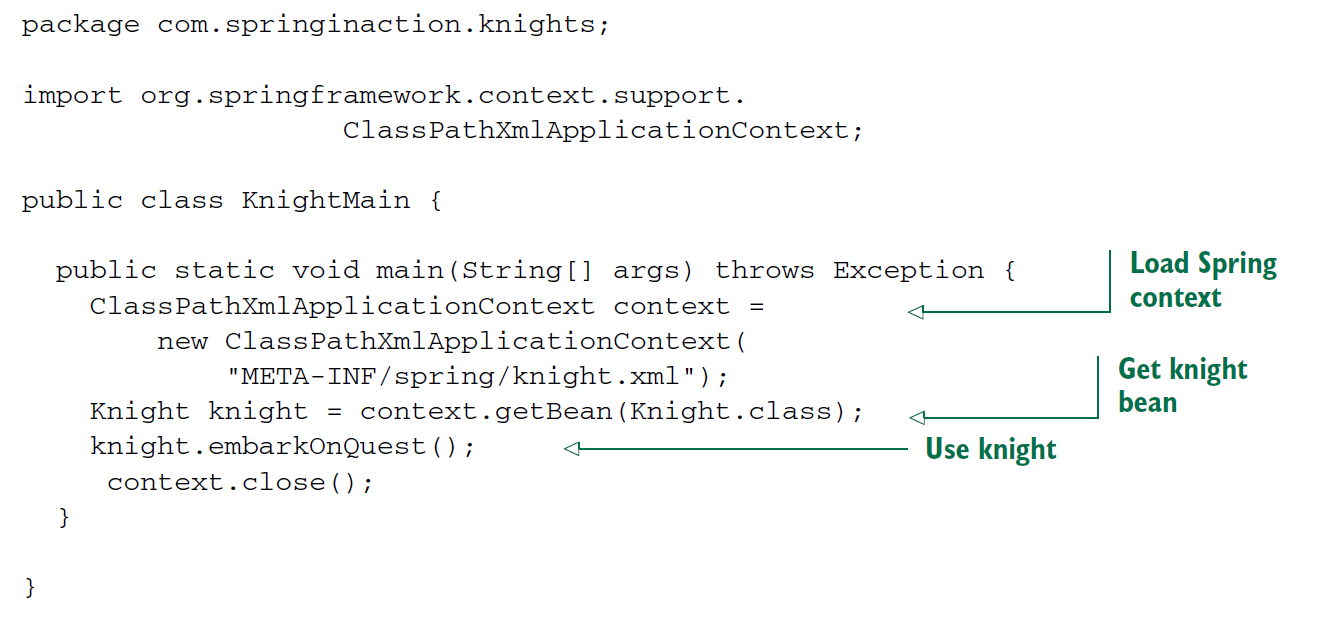
* Any nontrivial application (pretty much anything more complex than a Hello World example) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code.
* Coupling object is a two-headed beast. On the one hand, tightly coupled code is difficult to test, difficult to reuse, and difficult to understand, and it typically exhibits buggy behavior (fixing one bug results in the creation of one or more new bugs)
* On the other hand, a certain amount of coupling is necessary – Completely uncoupled code doesn’t do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but should be carefully managed.
* With **DI,** objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren’t expected to create or obtain their dependencies.
* Dependencies are injected into the objects that need them.
* Key benefit of DI- loose coupling. If an object only knows about its dependencies by their interface (not by their implementation or how they’re instantiated), then the dependency can be swapped out with a different implementation without the depending object knowing the difference.

**Wiring**

* The act of creating associations between application components is commonly referred to as *wiring.*
* In spring, there are many ways to wire components together, but a common approach has always been via XML.
* Application context loads bean definitions and wires them together. The Spring application context is fully responsible for the creation of and wiring of the objects that make up the application.
* Spring comes with several implementations of its application context, each primarily differing only in how it loads its configuration.
* Whenever we want to load the configuration from a xml file, an appropriate choice for application context might be ***ClassPathXmlApplicationContext****.* This Spring context implementation loads the Spring context from one or more XML files located in the application’s classpath. The main() method in the following listing uses *ClassPahtXmlApplicationContext* to load knights.xml and to get a reference to the *Knight* object.

****

* Whether you **use XML-based or Java-based configuration**, the benefits of DI are the same. Although BraveKnight depends on a Quest, it doesn’t know what type of Quest it will be given or where that Quest will come from. Like Wise, SlayDragonQuest depends on a PrintStream, but it isn’t coded with knowledge of how that PrintStream comes to be. **Only Spring, thorough its configuration, knows how all the pieces come together.**  This makes it possible to change those dependencies with no changes to the depending classes.
* This example has shown approach to wiring beans in Spring. Don’t concern yourself too much with the details right now. We’ll dig more in Spring configuration later.
* We will look at ways that beans cab be wired in Spring, including a way to let Spring automatically discover beans and create the relationships between them.

****

* For Java-based configuration, Spring offers ***AnnotationConfigApplicationContext****.*
* Here the main() creates the Spring application context based on the knights.xml file. Then it uses the application context as factory to retrieve the bean whose ID is *knight.* With a reference to Knight object, it calls the embarkOnQuest() method to have the knight embark on the Quest he was given. Note that this class knows nothing about which type of Quest your hero has. For that matterm it’s blissfully unaeare of the fact that it’s dealing with BraveKnight. Only the knight.xml file knows for sure what the implementations are.

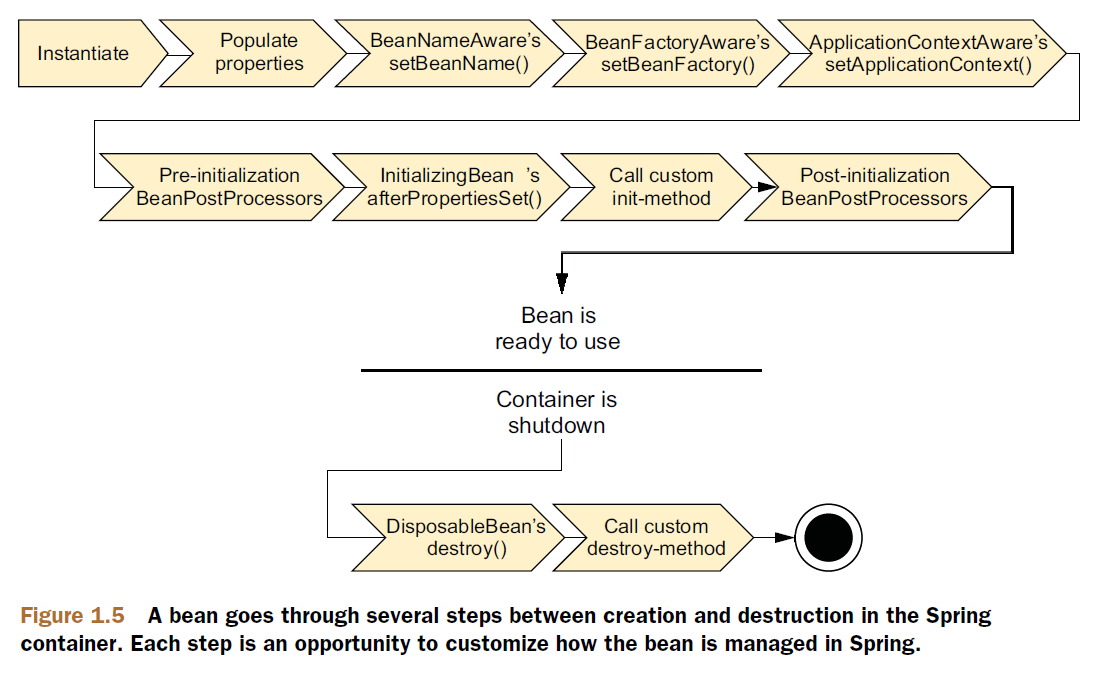
**Spring Container**

* In a spring-based application, your application objects live in the Spring *container.*
* The container creates the objects, wires them together, configures them, and manages their complete lifecycle from cradle to grave (or new to finalize() ).
* The container is the core of the Spring Framework. Spring’s container used DI (dependencies Injection) to manage the components that make up an application. This includes creating associations between collaborating components. As such, these objects are cleaner and easier to understand, they support reuse, and they’re easy to unit test.
* There’s no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. *Bean factories* (defined by the *org.springframework.beans.factory.BeanFactory* Interface) are the simplest of containers, providing basic support for DI. *Application contexts* (defined by the *org.springframework.context.ApplicationContext* interface ) build on the notion of a bean factory by providing application-framework services, such as the ability to publish application events to interested event listeners.
* Although it’s possible to work with spring using either bean factories or application contexts, bean factories are often too low-level for most applications. Therefore, application context are preffered over bean factories.
* **Working with an application context:** 
  + Spring comes with several flavors of application context. Here are a few that you’ll most likely encounter:
    - *AnnotaionConfigApplicationContext-* Loads a Spring application context from one or more Java-based Configuration classes.
    - *AnnotationConfigWebApplicationContext –* Loads a Spring web application context from one or more Java-based configuration classes.
    - *ClassPathXmlApplicationContext –* Loads a context definition from one or more XML files located in the classpath, treating context-definition files as class path resources.
    - *FileSystemXmlApplicationContext –* Loads a contex definition from one or more XML files in the filesystem.
    - *XmlWebApplicationContext –* Loads context definitions from one or more XML files contained in a web application.
    - The difference between using *FileSystemXmlApplicationContext* and *ClassPathXmlApplicationContext* is that *FileSystemXmlApplicationContext* looks for knight.xml in a specific location within the filesystem, whereas ***ClassPathXmlApplicationContext***looks for knight.xml anywhere in the classpath (including JAR files).
    - Alternatively, if you’d rather load your application context from a Java configuration, you can use *AnnotationConfiApplicationContext.*
    - Instead of specifying an XML file from which to load the Spring application context, *AnnotatinConfigApplicationContext has been given a configuration class from which to load beans.*

***A bean’s life:***

In a traditional Java application, the lifecycle of a bean is simple. Java’s *new* keyword is used to instantiate the bean, and it’s ready to use. Once the bean is no longer in use, it’s eligible for garbage collection and eventually goes to the big bit bucket in the sky.

* In contrast, the lifecycle of a bean in a Spring bean is more elaborate. There are many lifecycle phases for a spring bean and you may want to take advantage of some of the opportunities that spring offers to customize how a bean is created.



* Let’s break it down into more details:
  + Spring instantiates the bean.
  + Spring injects values and bean references into the bean’s properties.
  + If the bean implements *BeanNameAware*, Spring passes the bean’s ID to the *setBeanName*() method.
  + If the bean implements *BeanFactoryAware*, Spring calls the setBeanFactory*()* method, passing in the bean factory itself.
  + If the bean implements *ApplicationContextAware,* Spring calls the *setApplicationContext()* method, passing in a reference to the enclosing application context.
  + If the bean implements *BeanPostProcessor,* Spring calls its *postProcessAfterInitialization()* method.
  + At this point, the bean is ready to be used by the application and remains in the application context until the application context is destroyed.
  + If the bean implements the *DisposableBean* interface, Spring calls its destroy() method. Likewise, if the bean was declared with a *destroyed-method,*  the specified method is called.

**Wiring Beans in Detail**

The Spring container is responsible for creating the beans in your application and coordinating the relationships between those objects via DI. But it’s your responsibility as a Developer to tell Spring which beans to create and how to wire them together.

* When it comes to bean wiring specification, Spring is incredibly flexible, offering three primary wiring mechanisms:
* Explicit configuration in XML
* Explicit configuration in Java
* Implicit bean discovery and automatic wiring

The choice is largely a matter of personal taste, and you’re welcome to choose the approach that feels best for you.

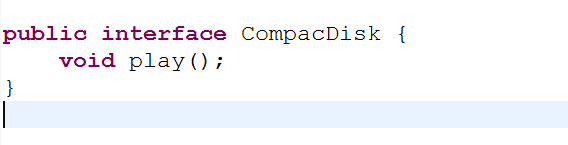
* It’s great that you have many choices about how to wire beans in Spring, but at some point you must select one.
* Spring’s configuration styles are mix-and-match, so you could choose XML to wire up some beans, Use Spring’s Java-based configuration (JavaConfig) for other beans, and let other beans be automatically discovered by Spring.
* Recommendation is to lean on automatic configuration as much as you can. The less configuration you have to do explicitly, the better. When you must explicitly configure beans (such as when you’re configuring beans for which you don’t maintain the source code), favor the type safe and more powerful JavaConfig over XML. Finally, fall back on XML only in situations where there’s convenient XML namespace you want to use that has no equivalent in JavaConfig.

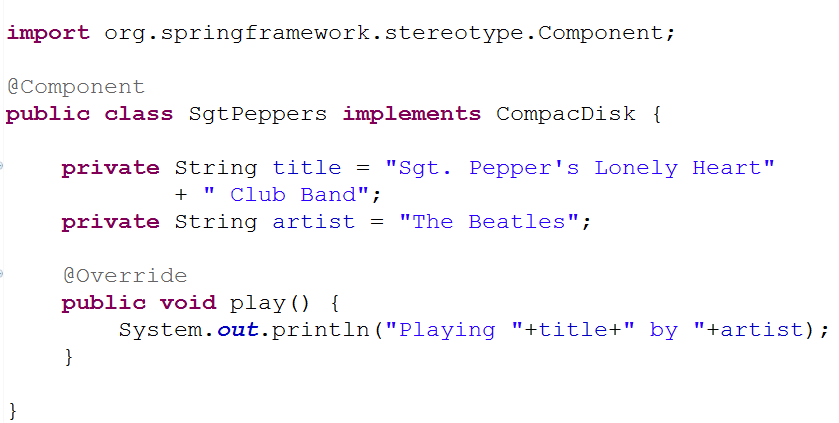
**Automatic Wiring**

Why bother explicitly wiring beans together if Spring can be configured to automatically do it for you…

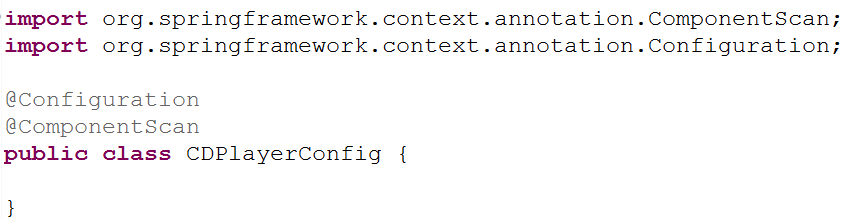
* Spring do automatic wiring from two angles:
* *Component scanning –* Spring automatically discovers beans to be created in the application context.
* *Autowiring –* Spring automatically satisfies bean dependencies.

**Creating discoverable beans**

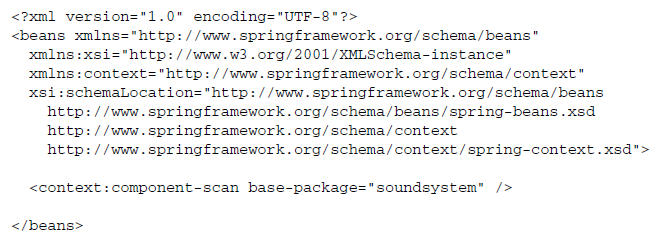




* As with the CompactDisc interface, the specifies of SgtPeppers aren’t important to this discussion. What you should take note of it, is that SgtPeppers is annotated with **@Component**.
* This simple annotation identifies this class as a component class and serves as a component class and serves as a clue to Spring that a bean should be created for the class.
* There’s no need to explicitly configure a SgtPeppers bean; Spring will do it for you because this class is annotated with @Component.
* Component scanning isn’t turned on by default, however. You’ll still need to write an explicit configuration to tell Spring to seek out classes annotated with @Component and to create beans from them.
* The configuration class in the following listing shows the minimal configuration to make this possible.



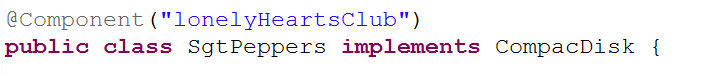
* The CDPlayerConfig class defines a Spring wiring specification, expressed in Java.
* For Now, observe that CDPlayerConfig doesn’t explicitly define any beans itself. Instead, it’s annotated with @ComponentScan to enable component scanning in Spring.
* @ComponentScan will default to scanning the same package as the configuration class and spring will scan that package and any subpackages underneath it, looking for classes that are annotated with @Component.
* It should find the CompacDisk class and automatically create a bean for it in the Spring.
* If you’d rather turn on component Scanning via XML configuration, then you can use the **<context:component-scan>** element from Spring’s context namespace. Here is a minimal XML configuration to enable component scanning.



* Even though XML is an option for enabling component scanning, we will focus more on using the preferred Java-based configuration for the remainder of this discussion.
* In XML the *<context:component-scan>* element has attributes and sub-elements that mirror the attributes you’ll use when working with *@Component-Scan*

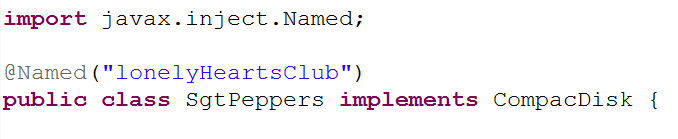
***Naming a component-scanned bean***

* All beans in a Spring application context are given an ID.
* In previous example, although you didn’t explicitly give the SgtPeppers bean an ID, it was given one derived from its name. Specifically, the bean was given an ID of sgtPeppers by lowercasing the first letter of the class name.
* If you’d rather give the bean a different ID, all you have to do is pass the desired ID as value to the *@Component* annotation. For example< if you wanted to identify the bean as *lonelyHeartClub*, the you’d annotate the *SgtPeppers* class with *@Component* like this:



eclipse_2017-06-27_12-18-20.png

* Another way to name a bean is not to use the *@Component* annotation at all. Instead, you can use the @Named annotation from the Java Dependency Injection specification (JSR-300) to provide a bean ID:

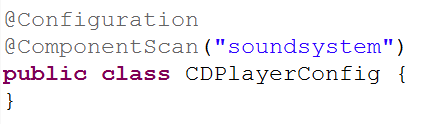


eclipse_2017-06-27_12-18-20.png

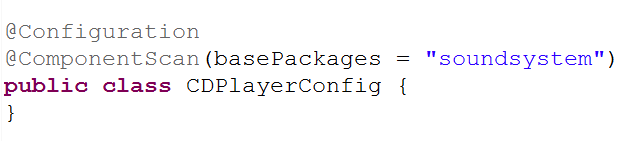
* Spring supports the @Named annotation as an alternative to @Component. There are a few subtle differences, but in most common cases they’re interchangeable.
* The author have a strong preference for the *@Component* annotation, largely because *@Named* is.. well … poorly named. It doesn’t describe what it does as well as *@Component.* Therefore, the author won’t use *@Named* any further in this book or its examples.

***Setting a base package for component scanning***

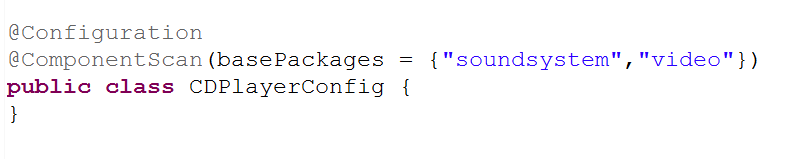
* So far we’ve used *@ComponentScan* with no attributes. That means it will default to the configuration class’s package as its base package to scan for components.
* But what if you want to scan a different package? Or what if you want to scan multiple base packages?
* One common reason for explicitly setting the base package is so that you can keep all of you configuration code in a package of its own, separate from the rest of you application’s code. In that case, the default base package won’t do.
* To specify a different base package, you only need to specify the package in *@ComponentScan’s* value attribute.



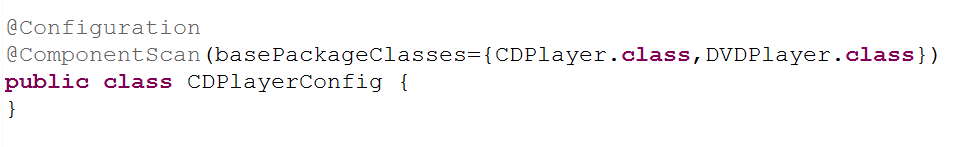
* Or, if you’d rather it be clear that you’re setting the base package, you can do so with the *basePackage* attribure:



* You probably noticed that *basePackages* is plural. If you’re wondering whether that means you can specify multiple base packages, you can. All you need to do is set *basePackages* to an array to packages to be scanned:



* The base packages are expressed as *String* values. But it’s not very type-safe. If you were to refactor the package names, the specified base packages would be wrong.
* Rather than specify the packages as simple *String* values, *@ComponentScan* also offers you the option of specifying them via classes or interfaces that are in the packages:

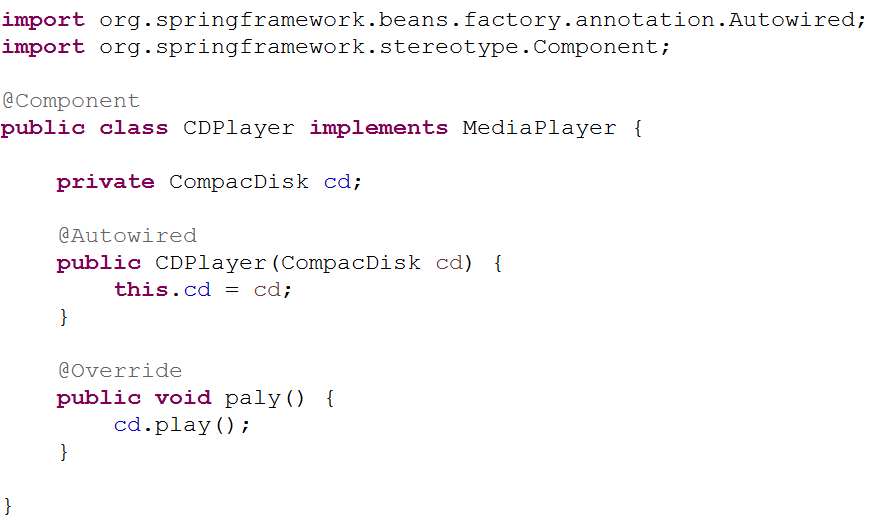


* The basePackages attribute has been replaced with *basePackageClasses*. And instead of identifying the packages with *String* names, the array given to *basePackageClasses* includes classes. Whatever packages those classes are in will used as the package for component scanning.
* You might consider creating an empty marker interface in the packages to be scanned. With a marker interface, you can still have a refactor-friendly reference, but without references to any actual application code (that could later be refactored out of the package you intended to component-scan).

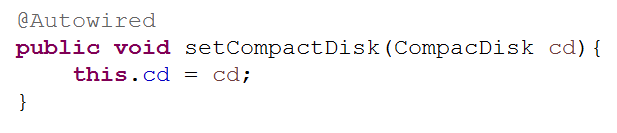
***Annotating beans to be automatically wired***

­­­Autowiring is a means of letting Spring automatically satisfy a bean's dependencies by finding other beans in the application context that are a match to the bean's needs. To indicate that autowiring should be performed, you can use Spring’s *@Autowired* annotation.

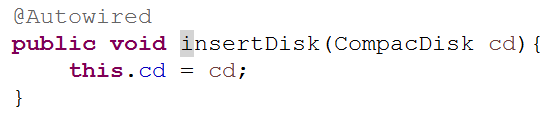
For example, consider the *CDPlayer* class in the following listing. Its constructor is annotated with @Autowired, indicating that when Spring creates the *CDPlayer* bean, it should instantiate it via that constructor and pass in a bean that is assignable to *CompactDisc.*



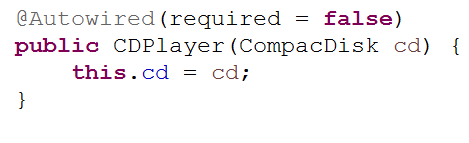
* The *@Autowired*  annotation’s use isn’t limited to constructors. It can also be used on a property’s setter method. For example, if CDPlayer had a *setCompactDisk()* method, you might annotate it for autowiring like this:



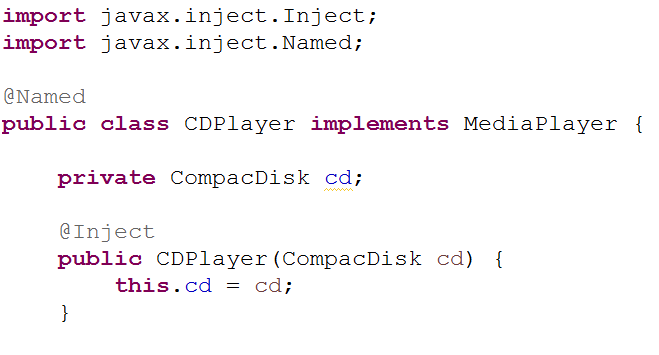
* After Spring has instantiated the bean, it will try to satisfy the dependencies expressed through methods such as the *setCopactDisc()* method that are annotated with *@Autowired.*
* There’s nothing special about setter methods. @*Autowired* can also beapplied on any method on the classs. Pretending that CDPlater has an *insertDisc()* method, @*Autowired* would work equally well there as on *setCompactDisk():*



* Whether it’s a constructor, a setter method, or any other method, Spring will attempt to satisfy the dependency expressed in the method’s parameters. **Assuming that one and only one bean matches, that bean will be wired in.**
* If there are no matching beans, Spring will throw an exception as the application context is being created. To avoid that exception, you can set the *required* attribute on @*Autowired* to *false*:

****

* When *required* is *false*, Spring will attempt to perform autowiring; but if there are no matching beans, it will leave the bean unwired. you should be careful setting *required*  to *false*, however. Leaving the property unwired could lead to *NullPointerExceptions* if you don’t check for *null* in your code.
* In the event that multiple beans can satisfy the dependency, Spring will throw an exception indicating ambiguity in selecting a bean for autowiring. We’ll talk more about managing ambiguity in autowiring later.
* @*Autowired* is a Spring-specific annotation. If it troubles you to be scattering Spring-specific annotations throughout you code for autowiring, you might consider using the *@Inject* annotation instead:



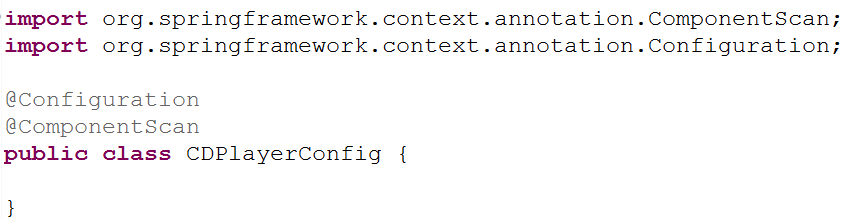
* *@Inject* comes from Java Dependency Injection specification, the same specification that gave us *@Named.* Spring supports the @*Inject* annotation for autowring alongside its own *@Autowired.* Although there are some subtle differences between *@Inject* and @*Autowired,* they’re interchangeable in many cases.

***Wiring beans with Java***

* Although automatic Spring configuration with component scanning and automatic wiring is preferable in many cases, there are times when automatic configuration isn’t an option and you must configure Spring explicitly.
* For instance, let’s say that you want to wire components from some third-party library into your application. Because you don’t have the source code for that library, there’s no opportunity to annotate its classes with *@Component* and *@Autowired* . Therefore, automatic configuration isn’t an option.
* In that case, you must turn to explicit configuration. You have two choices for explicit configuration: Java and XML.
* JavaConfig is the preferred option for explicit configuration because it’s more powerful, type-safe, and refactor-friendly. That’s because it’s just Java Code, like any other Java Code in your application.
* It’s important to recognize that JavaConfig code isn’t just any other Java code. It’s conceptually set apart from the business logic and domain code in your application. Even though it’s expressed in the same language as those components, JavaConfig is configuration code.
* This means it shouldn’t contain any business logic, nor should JavaConfig invade any code where business logic resides.
* JavaConfig is often set apart in a separate package from the rest of an application’s logic so there’s no confusion as to its purpose.

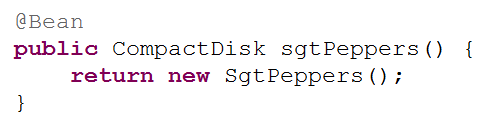
***Creating a configuration class***

Let’s revisit *CDPlayerConfig* from that example:

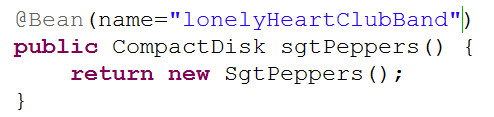
****

* The key to creating a JavaConfig class is to annotate it with *@Configuration*. The *@Configuration* annotation identifies this as a configuration class, and it’s expected to contain details on beans that are to be created in the Spring application context.
* If you remove the *@Component* annotation gone, the *CDPlayerConfig* is ineffective. If you were to run *CDPlayerTest* now, the test would fail with a *BeanCreationExcepton.* The test expects to be injected with *CDPlayer* and *CompactDisk,* but those beans are never created because they’re never discovered by component scanning.
* To make the test happy again, you could put @*ComponentScan* back in.
* Let’s see how you can wire the *CDPlayer* and *CompactDisc* beans in *JavaConfig.*

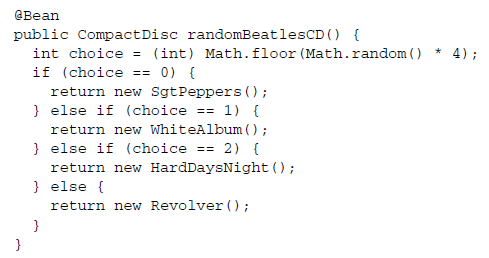
***Declaring a simple bean***

To declare a bean in JavaConfig, you write a method that creates an instance of the desired type and annotate it with *@Bean.* For example, the following declares the *CompactDisc* bean*:*

* The @Bean annotation tells Spring that this method will return an object that should be registered as a bean in the Spring application context. The body of the method contains logic that ultimately results in the creation of the bean instance.
* By default, the bean will be given an ID that is the same as the *@Bean-annotated method’s name. In this case, the bean* will be named *compactDisc.*
* If you’d rather it have a different name, you can either rename the method or prescribe a different name with the *name* attribute:

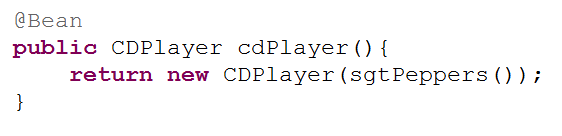


* No matter how you name the bean, this bean declaration is about as simple as they come. The body of the method returns a new instance of *SgtPeppers*. But because it’s expressed in Java, it has every capability afforded it by the Java language to do almost anything to arrive at the *CompactDisc* that is returned.
* Unleashing your imagination a bit, you might do something crazy like randomly selecting a *CompactDisc* from a selection of choices:

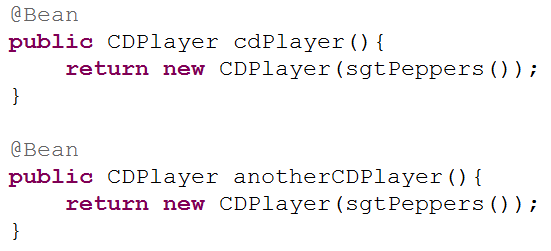


***Injecting with JavaConfig***

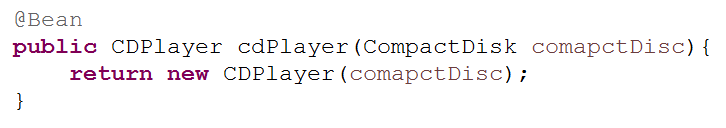
* You must declare the *CDPlayer* bean, which depends on a *CompactDisc.* How can you wire that up in JavaConfig?
* The simplest way to wire up beans in JavaCongig is to refer to the reference bean’s method. For example, here’s how you might declare the CDPlayer bean:



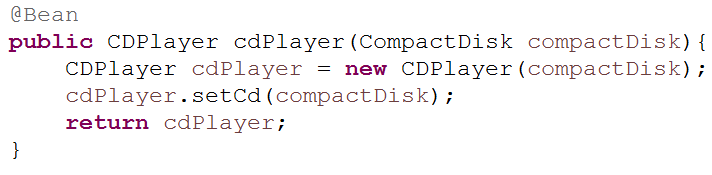
* The *cdPlayer*() method, like the *sgtPeppers*() method, is annotated with *@Bean* to indicate that it will produce an instance of a bean to be registered in the Spring application context. The ID of the bean will be *cdPlayer*, the same as the method’s name.
* The body of the *cdPlayer()* method differs subtly from that of the *sgtPeppers()* method. Rather than construct an instance via its default method, the *CDPlayer* instance is created by calling its constructor that takes a *CompactDisc.*
* It appears that the *CompactDisc*  is provides by calling *sgtPeppers,* but that’s not exactly true. Because the *sgtPeppers()* method is annotated with @*Bean,* Spring will intercept any calls to it and ensure that the bean produced by that method is returned rather than allowing it to be invoked again.
* For example, suppose you were to introduce another *CDPlayer* bean that is just like the first:



* If the calls to *sgtPeppers()* was treated like any other call to Java method, then each *CDPlayer()* would be given its own instance of *SgtPeppers.* That would make sense if we were talking about real CD players and compact discs. If you have two CD Players, there’s no physical way for a single compact disc to simultaneously be inserted into two CD players.
* In software, however, there’s no reason you couldn’t inject the same instance of *SgtPeppers* into as many other beans you want. **By default, all beans in Spring are singletons,** and there’s no reason you need to create a duplicate instance for the second *CDPlayer* bean. So Spring intercepts the call to *sgtPeppers()* and makes sure that what is returned is the Spring bean that was created when Spring itself called *sgtPeppers()* to create the *CompactDisc* bean. Therefore, both *CDPlayer* beans will be given the same instance of *SgtPeppers.*
* Referring to a bean by calling its method can be confusing. There’s another way that might be easier to digest:



* Here, the *cdPlayer()* method asks for a *CompactDisc* as a parameter. When Spring calls *cdPlayer()* to create the *CDPlayer* bean, it autowires a *CompactDisc* into the configuration method. Then the body of the method can use it however it sees fit.
* With this technique, the *cdPlayer()* method can still inject the *CompactDisc* into the *CDPlayer’s* constructor without explicitly referring to the *CompactDisc’s @Bean* method.
* The second approach to referring to other beans is usually the best choice because it doesn’t depend on the *CompactDisc* bean being declared in the same configuration class. In fact, there’s nothing that says the *CompactDisc* bean even needs to be declared in JavaConfig; it could have been discovered by component scanning or declared in XML..
* You could break up your configuration into a healthy mix of configuration classes, XML files, and automatically scanned and wired beans.
* No matter how the *CompactDisc* was created, Spring will be happy to hand it to this configuration method to create the CDPlayer bean.
* If you wanted to inject a *CompactDisc* vie a setter method, it might look like that:



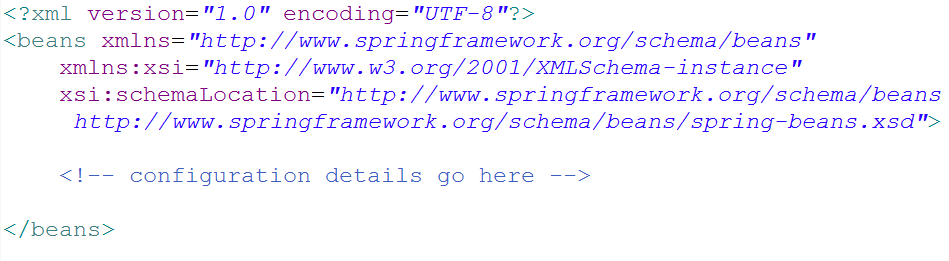
***Wiring beans with XML***

So far, you’ve seen how to let Spring automatically discover and wire beans. And you’ve seen how to step in and explicitly wire beans using JavaConfig. But, there’s another option for bean wiring that, although less desirable, has a long history with Spring.

* Since the beginning of Spring, XML has been the primary way of expressing configuration. Countless lines of XML have been created in the name of Spring. And for many, Spring has become synonymous with XML configuration.
* Now that Spring has strong support for automatic configuration and Java-based configuration, XML should not be your first choice.
* Because so much XML-based Spring configuration has already been written, it’s important to understand how to use XML with Spring. This section will only help you work with existing XML configuration, and that you’ll lean on automatic configuration and JavaConfig for any new Spring work you do.

**Creating an XML configuration specification**

* For XML configuration, that means creating an XML file rooted with a *<beans>* element.
* The simplest possible Spring XML configuration looks like this:



* As you can see that this basic XML configuration is already much more complex than an equivalent JavaConfig class. Whereas JavaConfig’s *@Configuration* annotation was all you needed to get started, the XML elements for configuring Spring are defined in several XML schema (XSD) files that must be declared in the preamble of the XML configuration file.

**CREATING XML CONFIGURATION WITH SPRING TOOL SUITE** *An easy way to create and manage Spring XML configuration files is to use Spring Tool Suite* (<https://spring.io/tools/sts>). Select File > New > *Spring XML Configuration File from Spring Tool Suite’s menu to create a Spring XML configuration file, and select from one of the available configuration namespaces.*

* The most basic XML elements for wiring beans are contained in the *spring-beans* schema, which is declared as the root namespace of this XML file.
* The *<beans>* element, the root element of any Spring configuration file, is one of the elements in this schema.
* In the above XML schema, you have a perfectly valid Spring XML configuration. It’s also a perfectly useless configuration, because it dosen’t (yet) declare any beans. To give it some life, let’s re-create the CD example, this time using XML configuration instead of JavaConfig or automatic configuration.

**Declaring a simple <bean>**

* To declare a bean in Spring’s XML-based configuration, you’re going to use another element from the *spring-beans* schema: the *<bean>* element. The *<bean>* element is the XML analogue to JavaConfig’s *@Bean annotation.*
* You can uset it to declare the *CompactDisc* bean like this:

eclipse_2017-06-30_11-01-03.png

* Here you declare a very simple bean. The class used to create this bean is specified in the *class* attribute and is expressed as the fully qualified class name.
* **For lack of an explicitly given ID, the bean will be named according to fully qualified class name. In this case, the bean’s ID will be** *soudnsytem.SgtPeppers#0.*
* The #0 is an enumeration used to differentiate this bean from any other bean of the same type. If you were to declare another *SgtPeppers* bean without explicitly identifying it, it would automatically be given an ID of soundsystem.SgtPeppers#1.
* Even though it’s convenient to have a bean named automatically for you, the generated names will be less useful if you need to refer to them later. Therefore, it’s usually a good idea to give each bean a name of your own choosing via the *id* attribute:

eclipse_2017-06-30_11-01-03.png

* **Reducing Verbosity:** *To cut down on XML verbosity, only explicitly name a bean if you’ll need to refer to it by name (such as if you were to inject a reference to it into another bean).*
* Before we go any further, let’s take a moment to examine some of the characteristics of this simple bean declaration.
* The first thing to notice is that you aren’t directly responsible for creating an instance of *SgtPeppers* as you were when using JavaConfig. When Spring sees this *<bean> element, it will create a SgtPeppers* bean for you by calling its default constructor.
* Bean creation is much more passive with XML configuration. But it’s also less powerful than JavaConfig, where you can do almost anything imaginable to arrive at the bean instance.
* Another notable thing about this simple *<bean>* declaration is that you express the type of the bean as a String set to the *class* attribute.
* Who’s say that the value given to *class* even refers to a real class? Spring’s XML configuration doesn’t benefit from compile-time verification of the Java types being referred to. And even if it does refer to an actual type, what will happen if you rename the class?

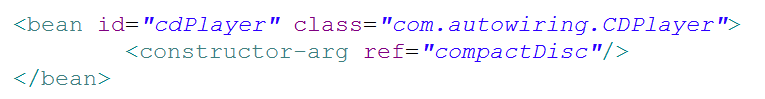
**CHECK XML VALIDITY WITH AN IDE:** *Using a Spring-aware IDE such as Spring Tool Suite can help a lot to ensure the validity of your Spring XML configuration.*

***Initializing a bean with constructor injection***

* There’s only one way to declare a bean in Spring XML configuration: use the *<bean>* element, and specify a *class* attribute. Spring takes it from there.
* But when it comes to declaring DI in XML, there are several options and styles. With specific regard to constructor injection, you have two basic options to choose from:
* The ***<constructor-arg>***element
* Using the c-namespace introduced in Spring 3.0
* The difference between these two choices is largely one of verbosity. The *<constructor-arg>* element is generally more verbose than using the c-namespace and results in XML that is more difficult to read.
* On the other hand, *<constructor-arg>* can do a few things that the c-namespace can’t.

**INJECTING CONSTRUCTORS WITH BEAN REFERENCES**

* As currently defined, the *CDPlayer* bean has a constructor that accepts a *CompactDisc.* This makes it a perfect candidate for injection with a bean reference.
* Because you’ve already declared a *SgtPeppers* bean, and because the *SgtPeppers* class implements the *CompactDisc* interface, you have a bean to inject into a *CDPlayer* bean. All you need to do is declare a *SgtPeppers* bean and because the *SgtPeppers* class implements the *CompactDisc* interface, you have a bean to inject into a *CDPlayer bean.* All you need to do is declare the *CDPlayer* bean in XML and reference the *SgtPeppers* bean by its ID:

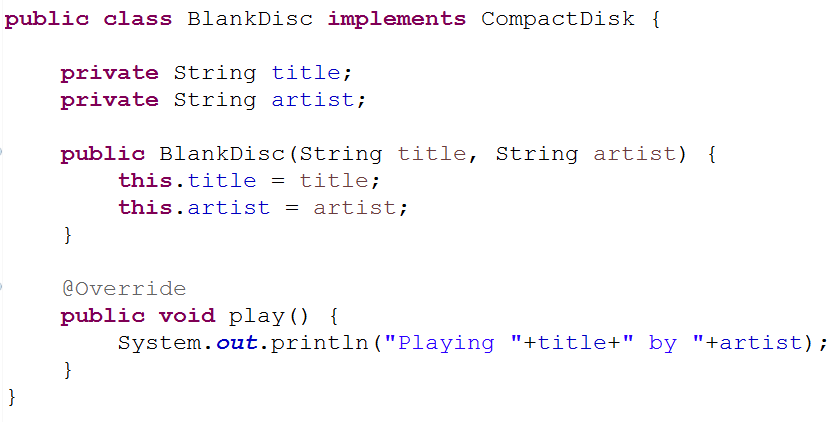


When Spring encounters this *<bean>* element, it will create an instance of *CDPlayer.*

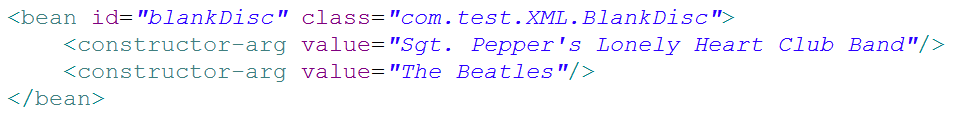
* The *<constructor-arg>* element tells it to pass a reference to the bean whose ID is *compactDisc* to the *CDPlayer’s* constructor.

**INJECTING CONSTRUCTORS WITH LITERAL VALUES**

* Sometimes you need to do is configure an object with a literal value, To illustrate, suppose you were to create a new implementation of *CompactDisc,* as show here:

****

* Unlike *SgtPeppers*, which was hard-caded with a title and artis, this implementation of *CompactDisc* is considerably more flexible.

****

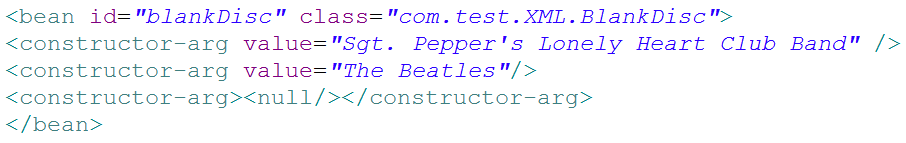
* Once again, the *<constructor-arg>* element is used to inject into constructor arguments. But this time, instead to using the *ref* attribute to reference another bean, you use the *value attribute* to indicate that the given value is to be taken literally and injected into the constructor.

**WIRING COLLECTIONS**

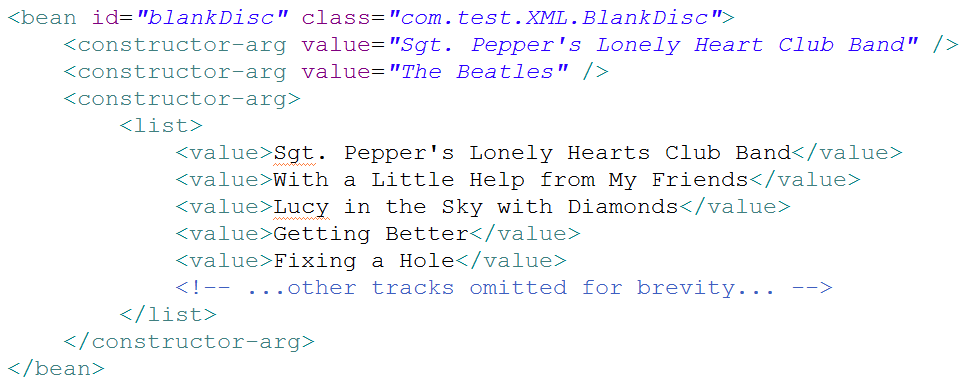
If CompactDisc is to truly model a real-world CD, then it must also have the notion of a list of tracks. Consider the following new *BlankDisc* shown here:



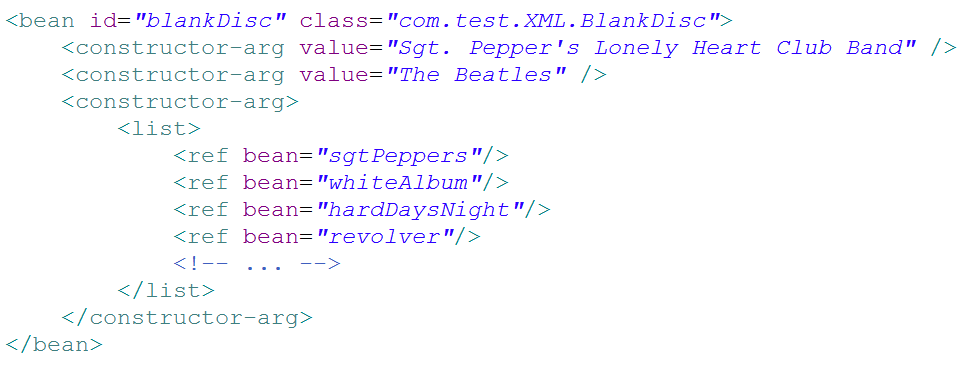
* This change has implication for how you configure the bean in Spring. You must provide a list of tracks when declaring the bean.
* The simplest thing you could do is leave the list null. Because it’s a constructor argument, you must specify it, but you can still pass *null* like this:



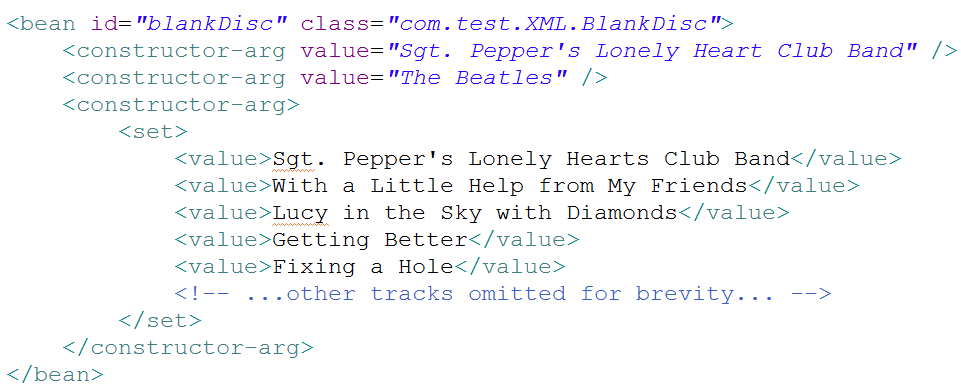
* The *<null/>* element does as you’d expect: it passes null into the constructor. It’s a dirty fix, but it will work at injection time. You’ll get a *NullPointerException* when the play() method is called, so it’s far from ideal.
* A better fix would be to supply a list of track names. For that you have a couple of options. First, you could specify it as a list, using the *<list>* element:



* The *<list>* element is a child of *<constructor-arg>* and indicates that a list of values is to be passed into the constructor. The *<value>* element is used to specify each element of the list.
* Similarly, a list of bean references could be wired using the *<ref>* element instead of <value>, For example, suppose you have a *Discography* class with the following constructor:



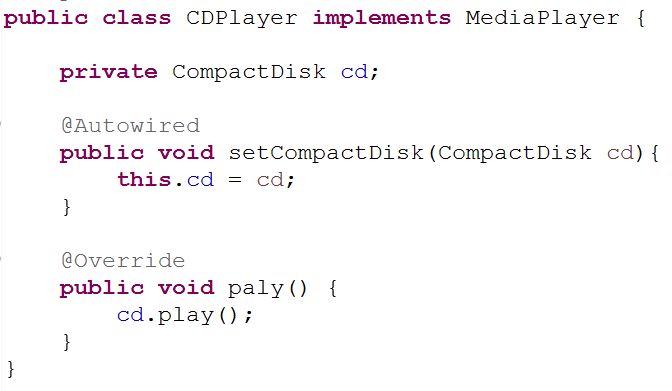
* It makes sense to use *<list>* when wiring a constructor argument of type *java.util.List.* Even so, you could also use the *<set>* element in the same way:



* There’s little difference between *<set> and <list>.* The main difference is that when Spring creates the collection to be wired, it will create it as either a *java.util.Set* or *java.util.List*. If it’s a *Set,* then any duplicate values will be discarded and the ordering may not be honored. But in either case, either a *<set> or a <list>* can be wired into a *List, a Set, or even an* array.

***Setting properties***

Up to this point, the *CDPlayer* and *BlankDisc* classes have been configured entirely through constructor injection and don’t have any property setter methods. In contrast, let’s examine how property injection works in Spring XML. Suppose that your new property-injected *CDPlayer* looks like this:

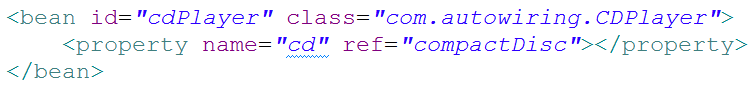


**CHOOSING BETWEEN CONSTRUCTOR INJECTION AND PROPERTY INJECTION.** As a general rule, constructor injection for hard dependencies and property injection for any optional dependencies.

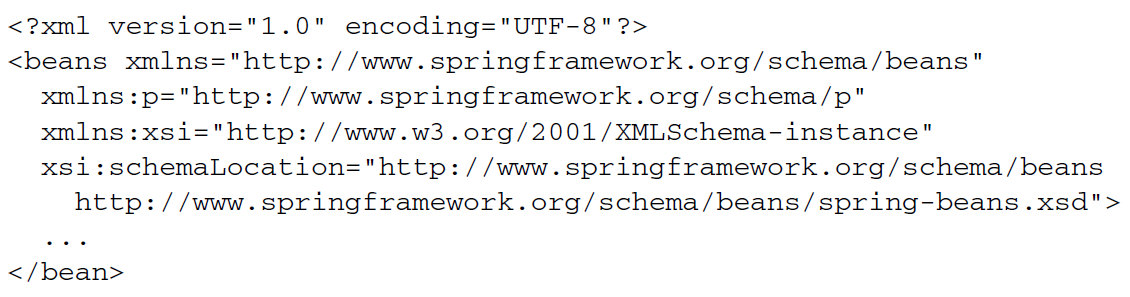
* Now that *CDPlayer* doesn’t have any constructor (aside from the implicit default constructor) ,it also doesn’t have any hard dependencies. Therefore, you could declare it as a Spring bean like this:

eclipse_2017-06-30_11-01-03.png

* Spring will have no problem creating that bean. Your *CDPlayerTest* would fail with a *NullPointerException,* however, because you never injected *CDPlayer’s compactDisc* property. But you can fix that with the following change to the XML:

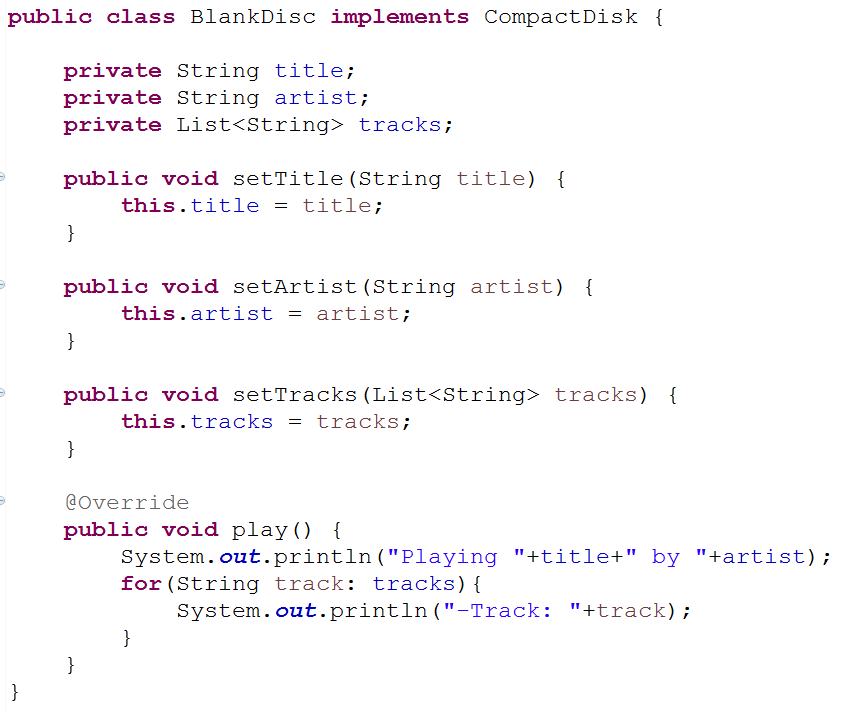


* The *<property>* element does for property setter methods what the <*constructor-arg>* element does for constructors. In this case, it references (with the ref attribute) the bean whose ID is *compactDisc* to be injected into the *compactDisc* property (via the *setCompactDisc()* method).
* Just as Spring offers the c-namespace as an alternative to the *<constructor-arg>* element, Spring also offers a succinct p-namespace as an alternative to the *<property>* element. To enable the p-namespace, you must declare it among the other namespaces in the XML files:



**INJECTING PROPERTIES WITH LITERAL VALUES**

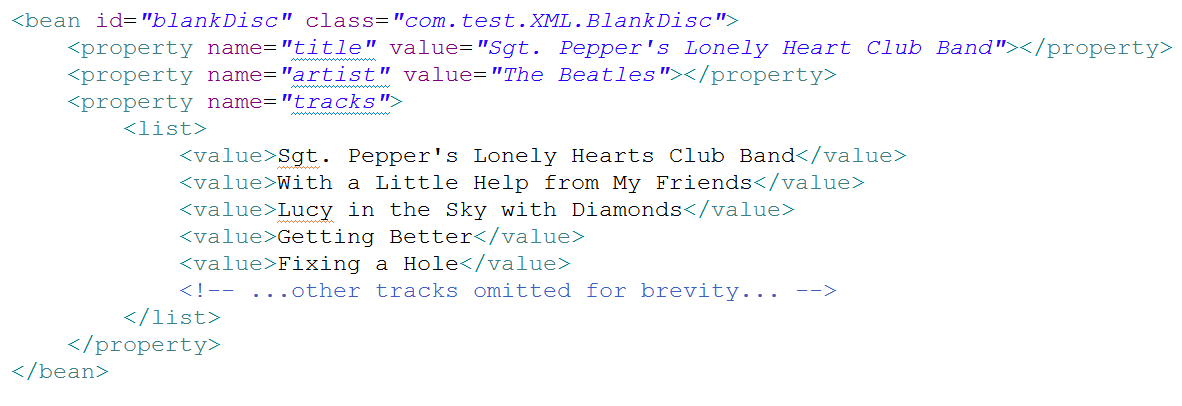
* Properties can be injected with literal values in much the same way as constructor arguments. As an example, let’s revisit the *BlankDisk* bean. This time, however, *BlankDisks* will be configured entirely by property injection, not constructor injection. The new *BlankDisc* class looks like this:

****

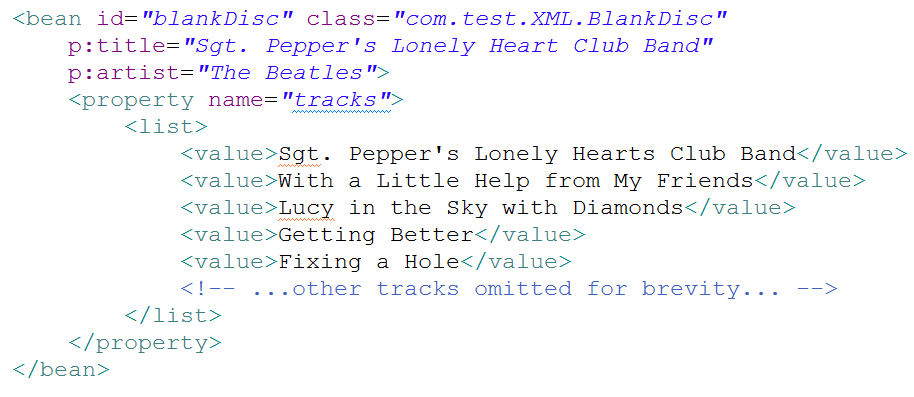
* Now you are no longer obligated to wire any of these properties. You could create a *BlankDisc* bean in its most blank form as follows:

eclipse_2017-06-30_11-01-03.png

* Of course, wiring the bean without setting those properties wouldn’t play out well at runtime. The *play()* method would claim that it’s playing null by null just before a *NullPointerException* is thrown because there are no tracks. Therefore, you probably should wire up those properties. You can do that using the *value attribute* of the *<property>* element:

**

* Aside from using the *<property> element’s value* attribute to set the *title* and *artist properties, notice how you set* the *tracks* property with a nested *<list>* element, the same as before when wiring the tracks through *<constructor-arg> .*
* Optionally, you can accomplish the same thing using p-namespace attributes:



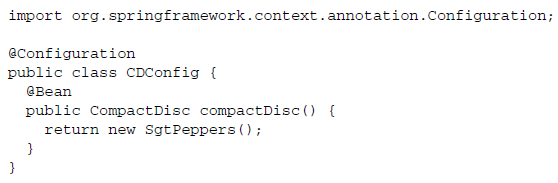
***Importing and mixing configurations***

* You’re free to mix component scanning and autowiring with JavaConfig and /or XML configuration.
* The first thing to know about mixing configuration styles is that when it comes to autowiring, it doesn’t matter where the bean to be wired comes from. Autowiring considers all beans in the Spring container, regardless of whether they were declared in JavaConfig or XML or picked up by component scanning.
* That leaves you with how to reference beans when doing explicit configuration, either with XML configuration or with Java configuration. Let’s start by seeing how to reference XML-configured beans from JavaConfig.

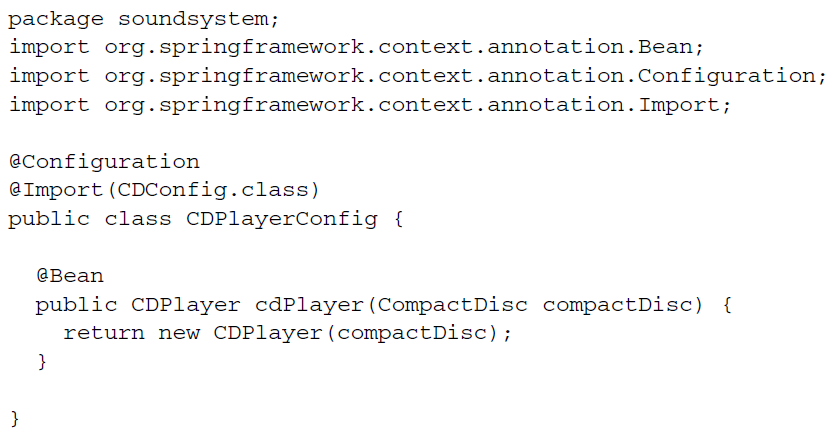
***Referencing XML configuration in JavaConfig***

* Let’s break out the *BlankDisc* bean from *CDPlayerConfig* into its own *CDConfig* Class, like this:

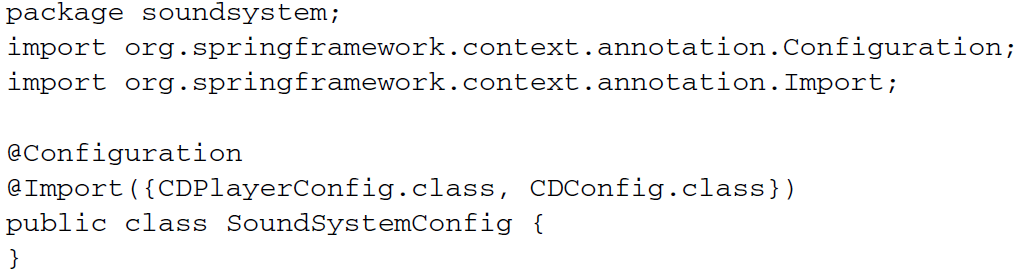
**eclipse_2017-06-30_11-01-03.png**

****

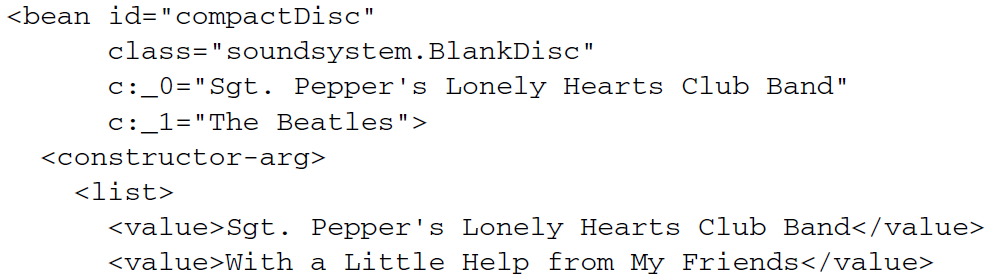
* Now that the *compactDisc()* method is gone from *CDPlayerConfig*, you need a way to bring the two configuration classes together. One way is to import *CDConfig* from *CDPlayerConfig* using the *@Import* annotation:

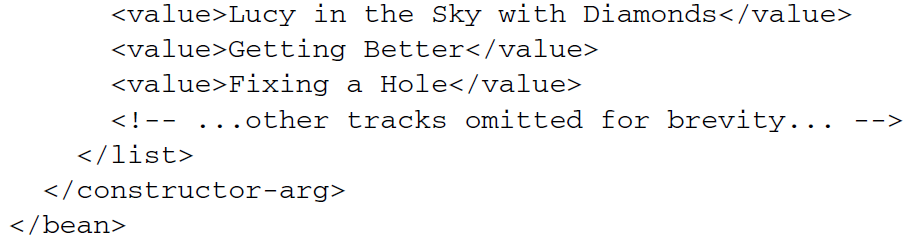
****

* Or, better yet, you can leave @*Import* out of *CDPlayerConfig* and instead create a higher-level *SoundSystemConfig* that uses @*Import* to bring configurations together:

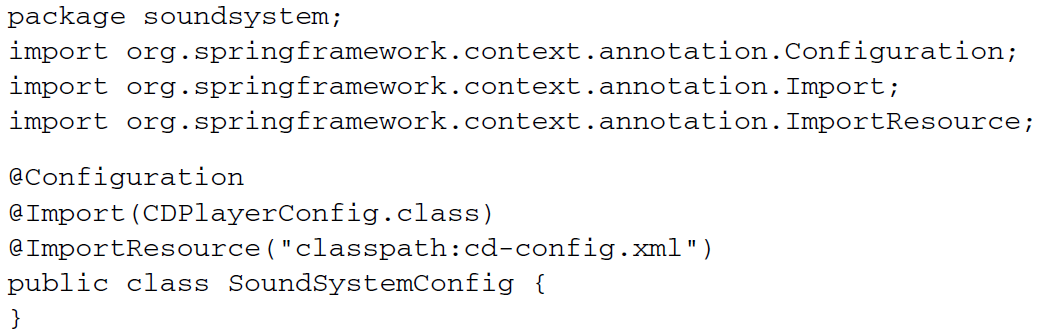
****

* Either way, you’ve separated the configuration of *CDPlayer* from the configuration of *BlankDisc.* Now let’s suppose that you want to configure the *BlankDisc* bean in XML like this:

****

****

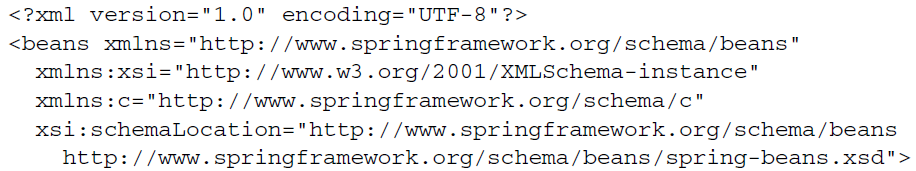
* With  *BlankDisc* being declared in XML, how can you have Spring load it in along with the rest of your Java-based configuration?
* The answer lies with the @*ImportResource* annotation. Assuming that the *BlankDisc* bean is declared in a file named cd-config.xml that can be found at the root of the classpath, you can change *SoundSystemConfig* to use @*ImportResource* like this:

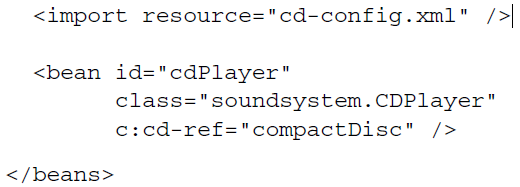
****

* Both beans – *CDPlayer* configured in *JavaConfig* *and BlankDisc* configured in XML – will be loaded into the Spring container. And because *CDPlayer’s* @*Bean* method accepts a *CompactDisc* as a parameter, the *BlankDisc* bean will be wired into it, even though it’s configured in XML.

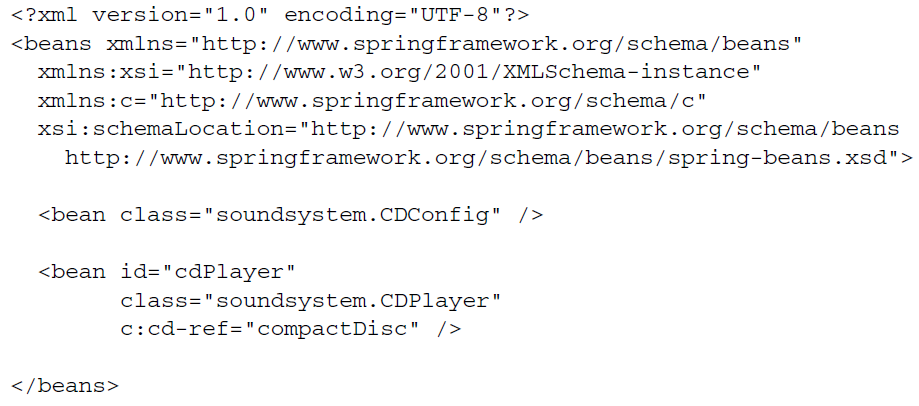
***Referencing JavaConfig in XML configuration***

* In XML, you can use the *<import>* element to split up the XML configuration.
* For example, suppose you were to split out the *BlankDisc* bean into its own configuration file called cd-config.xml, as you did when working with @*ImportResource.* You can reference that file from the XML configuration file using *<imort>:*

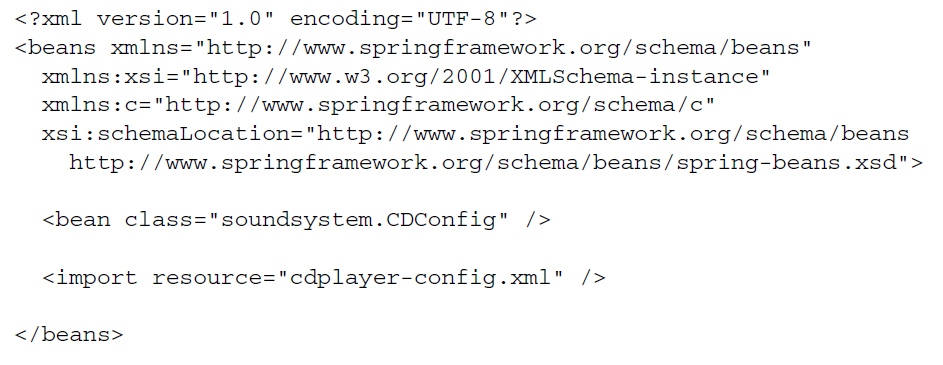
****

****

* Now, suppose that instead of configuring *BlankDisc* in XML, you want to configure it in XML while leaving the *CDPlayer* configuration in *JavaConfig.* How can your XML-based configuration reference a *JavaConfig* class?
* The *<import>* element only work to import other XML configuration files, and there isn’t an XML element whose job it is to import JavaConfig class into an XML configuration, you declare it as bean like this:

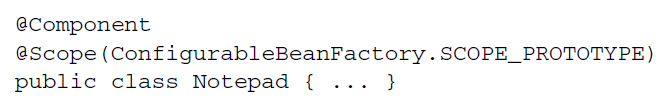
****

* And just like that, the two configurations – one expressed in XML and one expressed in Java – have been brought together. Similarly, you might consider a higher level configuration file that doesn’t declare any beans but that brings two or more configuration together. For example, you could leave the *CDConfig* bean out of the previous XML configuration and instead have a third configuration file that joins them:

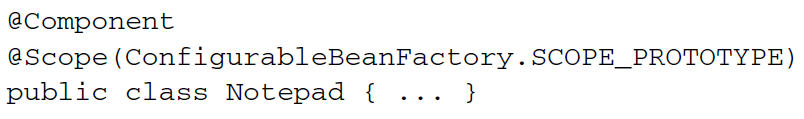
****

***Scoping beans***

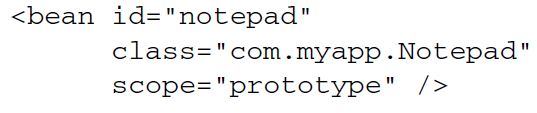
* By default, all beans created in the Spring Container are created as **singletons**. That is to say, no matter how many times a given bean is injected into other beans, it’s always the same instance that is injected each time.
* Most of the time, singleton beans are ideal. The cost of instantiating and garbage-collecting instances of objects that are only used for small tasks can’t be justified when an object is stateless and can be reused over and over again in an application.
* But sometimes you may find yourself working with a mutable class that does maintain some state and therefore isn’t safe for reuse. In that case, declaring the class as a singleton bean probably isn’t a good idea because that object can be tainted and create unexpected problems reused later.
* Spring defines several scopes under which a bean can be created, including the following:
* **Singleton**–One instance of the bean is created for the entire application.
* ***Prototype***–One instance of the bean is created every time the bean is injected into or retrieved from the Spring application context.
* ***Session***– In a web application, one instance of the bean is created for each session.
* ***Request***– In a Web application, one instance of the bean is created for each request.
* **Singleton scope is the default scope**, but as we’ve discussed, it isn’t ideal for mutable types. To select an alternative type, you can use the *@Scope* annotation, either in conjunction with the *@Component* annotation or with the *@Bean* annotation.
* For example, if you’re relying on component-scanning to discover and declare a bean, then you can annotate the bean class with *@Scope* to make it a prototype bean:



* Here, you specify prototype scope by using the *SCOPE\_PROTOTYPE* constant from the *ConfigurableBeanFactory* class. You could also use *@Scope(“prototype”),* but using the *SCOPE\_PROTOTYPE* is safer and less prone to mistakes.
* Alternatively, if you’re configuring the *Notepad* bean as a prototypein Java configuration, you can use *@Scope* along with *@Bean* to specify the desired scoping:



* And, in the event that you’re configuring the bean in XML, you can set the scope using the *scope* attribute of the *<bean>* element:



* Regardless of how you specify prototype scope, an instance of the bean will be created each and every time it’s injected into or retrieved from the Spring application context. Consequently, everyone gets their own instance of *Notepad.*

***Wiring with the Spring Expression Language***

* Spring 3 introduced Spring Expression Language (SpEL), a powerful yet succinct way of wiring values into a bean’s properties or constructor arguments using expressions that are evaluated at runtime.
* Things a SpEL can do:
* The ability to reference beans by their IDs
* Invoking methods and accessing properties on objects
* Mathematical, relational, and logical operations on values
* Regular expression matching
* Collection manipulation
* SpEL can also be used for purposes other than dependency injection.

**A FEW SPEL EXAMPLES**

* Few basic examples from which you can draw inspiration for your own expressions.
* The First thing to know is that SpEL expressions are framed with #{ … }, much as property placeholders are framed with ${ … }. What follows is possibly one of the simplest SpEL expressions you can write:

AcroRd32_2017-07-29_14-32-19.png

* Stripping away the #{ … } markers, what’s left is the body of a SpEL expression, which is a numeric constant. It probably won’t surprise you much to learn that this expression evaluates to the numeric value of 1. Of course, you’re not likely to use such a simple expression in a real application. You’re more likely to use such a simple expression in a real application. You’re more likely to build up more interesting expressions, such as this one:

AcroRd32_2017-07-29_14-32-19.png

* Ultimately this expression evaluates to the current time in milliseconds at the moment when the expression is evaluated. The T( ) operator evaluates *java.lang.System* as a type so that the *staticcurrentTimeMillis()* method can be invoked.
* SpEL expressions can also refer to other beans or properties on those beans. For example, the following expression evaluates to the value of the *artist* property on a bean whose ID is *sgtPeppers:*

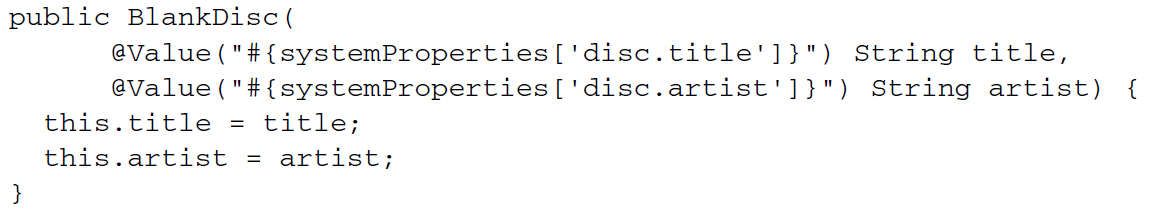
AcroRd32_2017-07-29_14-32-19.png

* You can also refer to system properties via the *systemProperties* object:

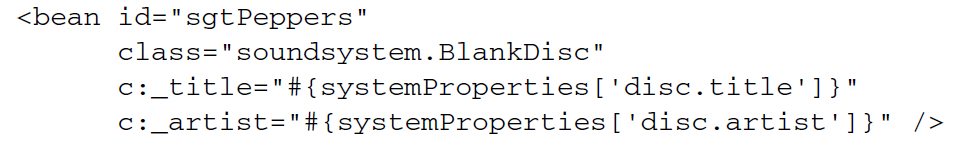
AcroRd32_2017-07-29_14-32-19.png

* These are just a few examples of SpEL. Let’s first, let’s consider how you might use these expressions during bean wiring.

When injecting properties and constructor arguments on beans that are created via component-scanning, you can use the *@Value* annotation, much as you saw earlier with property placeholders. Rather than use a placeholder expression, however, you use a SpEL expression. For example, here’s what the *BlankDisc* constructor might look like, drawing the album title and artist from system properties:



In XML configuration, you can pass in the SpEL expression to the *value* attribute of *<property> or <constructor-arg>,* or as the value given to a p-namespace or c-namespace entry. For example, here’s the XML declaration of the *BlankDisc* bean that has its constructor arguments set from a SpEL expression:



Now that we’ve looked at a few simple examples and how to inject values resolved from SpEL expressions, let’s go over some of the primitive expressions supported in SpEL.

**EXPRESSING LITERAL VALUES**

* You’ve already seen an example of using SpEL to express a literal integer value. But it can also be used for floating-point numbers, *String* values, and *Boolean* values.
* Here’s an example of a *SpEL* expression that is a floating-point value:

AcroRd32_2017-07-29_14-32-19.png

* Numbers can also be expressed in scientific notation. For example, the following expression evaluates to 98,700:

AcroRd32_2017-07-29_14-32-19.png

* A SpEL expression can also evaluate literal *String* values, such as:

AcroRd32_2017-07-29_14-32-19.png

* Finally, Boolean literals *true* and *false* are evaluated to their Boolean value. For Example,

AcroRd32_2017-07-29_14-32-19.png

**REFERENCING BEANS, PROPERTIES, AND METHODS**

* Another basic thing that a SpEL expression can do is reference another bean by its ID.
* For example, you could use SpEL to wire one bean into another bean’s property by using the bean ID as the SpEL expression (in this case, a bean whose ID it sgtPeppers):

AcroRd32_2017-07-29_14-32-19.png

* Now let’s say that you want to refer to the *artist* property of the *sgtPeppers.* What follows the period delimiter is a reference to the *artist* property.
* In addition to referencing a bean’s properties, you can also call methods on a bean. For example, suppose you have another bean whose ID is *artistSelector.* You can call that bean’s *selectArtist()* method in a SpEL expression like this:

AcroRd32_2017-07-29_14-32-19.png

* You can also call methods on the value returned from the invoked method. for example, if *selectArtist()* returns a String, you can call *toUpperCase()* to make the entire artist name uppercase lettering:

AcroRd32_2017-07-29_14-32-19.png

* This will work fine, as long as *selectArtist()* doesn’t return *null.* To guard against a *NullPointerException,* you can use the type-safe operator:

AcroRd32_2017-07-29_14-32-19.png

* Instead of a lonely dot (.) to access the *toUpperCase()* method now you’re using the *?.* operator. This operator makes sure the item to its left is’t *null* before accessing the thing on its right. So, if *selectArtist()* *returns* null, then SpEL won’t even try to invoke *toUpperCase().* The expression will to *null.*

**WORKING WITH TYPES IN EXPRESSIONS**

* The key to working with class-scoped methods and constants in SpEL is to use the *T()* operator. For example, to express Java’s *Math* class in SpEL, you need to use the *T()* operator like this:

T(java.lang.Math)

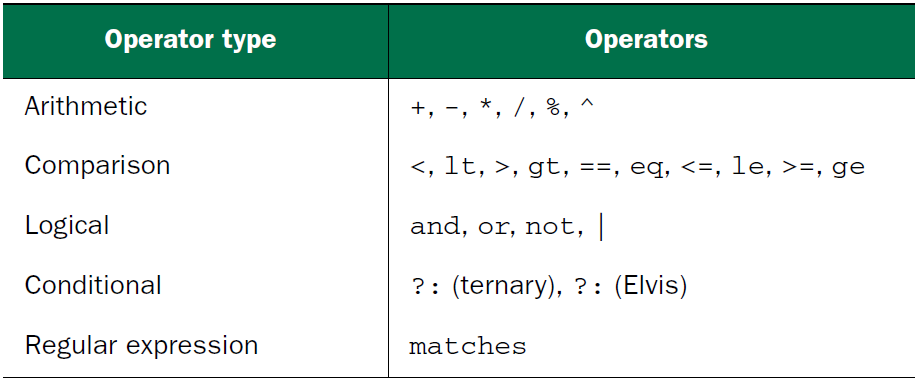
* The result of the *T()* operator, as shown here, is a *Class* object that represents *java.lang.Math.* You can even wire it into a bean property of type *Class,* if you want. But the real value of the *T( )* operator is that it gives you access to static methods and constants SpEL expression does the trick:

**AcroRd32_2017-07-29_14-32-19.png**

* Similarly, static methods can be invoked in the type resolved with the *T( )* operator. You’ve seen an example of using *T( )* to make a call to *System.currentTimeMillis().*
* Here’s another example that evaluates to a random value between 0 and 1:

**AcroRd32_2017-07-29_14-32-19.png**

* SpEL offers several operators that you can apply on values in SpEL expressions:

****

* As a simple example of using one of these operators ( \* ), but it also shows how you can compose simpler expression into a more complex expression.:

AcroRd32_2017-07-29_14-32-19.png

Here the value of pi is multiplied by 2, and the result is multiplied by the value of the *radius* property of a bean whose *ID* id *circle.* Essentially, it evaluates to the circumference of the circle defined in the *circle* bean.

* Similarly, you can use the carat symbol ( ^ ) in an expression to calculate a circle’s area:

AcroRd32_2017-07-29_14-32-19.png

The carat symbol is the power-of operator. In this case, it’s used to calculate the square of the circle’s radius.

* When working with *String* values, the + operator performs concatenation, just as in Java:

AcroRd32_2017-07-29_14-32-19.png

* SpEL also offers comparison operators for comparing values in an expression. Notice in previous table the comparison operators come in two forms: symbolic and textual. For the most part, the symbolic sperators are equivalent to their textual counterparts, and you’re welcome to use whichever one suits you best.
* For example, to compare two numbers for equality, you can use the double-equal ( == ) operator:

AcroRd32_2017-07-29_14-32-19.png

or you can use the textual *eq* operator:

AcroRd32_2017-07-29_14-32-19.png

* Either way, the result is the same. The expression evaluates to a Boolean: *ture* if *counter.total* is equal to 100 or *false* if it’s not.
* SpEL also offers a ternary operator that works much like Java’s ternary operator. For example, the following expression evaluates to the *String “Winner!”* if *scoreboard.score* > 1000 or “Loser” if not:

AcroRd32_2017-07-29_14-32-19.png

* A common use of the ternary operator is to check for a *null* value and offer a default value in place of the *null.* If *disc.title* is *null,* then the expression evaluates to “Rattle and Hum”.

AcroRd32_2017-07-29_14-32-19.png

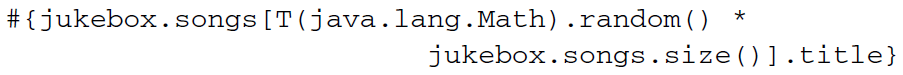
* This expression is commonly referred to as the *Elvis* operator. This strange name comes from using the operator as an emoticon, where the question mark appears to form the shape of Elvis Presley’s hair style.

**EVALUATING COLLECTIONS**

* Some of SpEL’s most amazing tricks involve working with collections and arrays. The most basic thing you can do is reference a single element from a list:

**AcroRd32_2017-07-29_14-32-19.png**

* This evaluates to the *title* property of the fifth (zero-based) element from the *songs* collection property on the bean whose ID is *jukebox*.
* To spice things up a bit, I suppose you could randomly select a song from the jukebox:

****

* As it turns out, the [ ] operator used to fetch an indexed element from a collection or array can also be used to fetch a single character from a *String.* For example,

**AcroRd32_2017-07-29_14-32-19.png**

* This references the fourth (zero-based) character in the *String,* or s.
* **SpEL** also offers a selection operator ( .?[ ] ) to filter a collection into a subset of the collection. As a demonstration, suppose you want a list of all songs in the jukebox where the *artist* property is *Aerosmith.* The following expression uses the selection operator to arrive at the list of available Aerosmith songs:

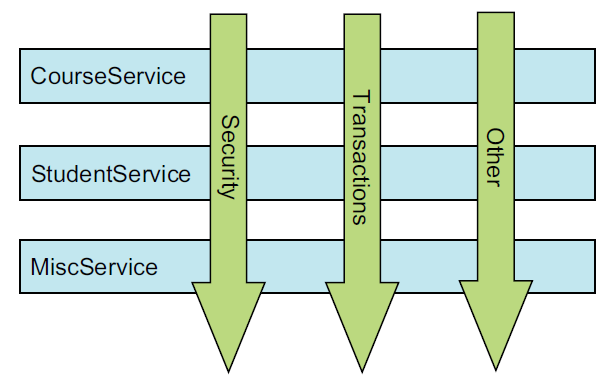
**AcroRd32_2017-08-02_20-08-14.png**

* **As you can see, the selection operator accepts another expression within its square brackets. As SpEL iterates over the list of songs, it evaluates that expression each entry in the songs collection.** If the expression evaluates to true, then the entry is carried over into the new collection. Otherwise it’s left out of the new collection. In this case, the inner expression checks to see if the song’s *artist* property equals *Aerosmith.*

*Aspect-oriented Spring*

***What is aspect-oriented programming?***

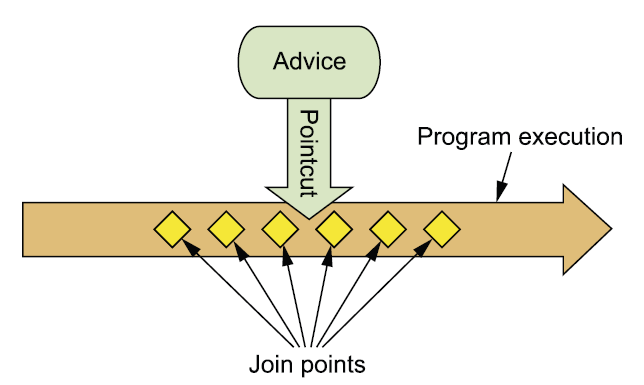
* Aspects help to modularize cross-cutting concerns. In short, a cross cutting concern can be described as any functionality that affects multiple points of an application. Security, for example, is a cross-cutting concern, in that many methods in an application can have security rules applies to them



* This figure represents a typical application that’s broken down into modules. Each module’s main concern is to provide services from its particular domain. But each module also requires similar ancillary functionality, 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 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 concern can now be modularized into special classes called ***aspects****.* This has two benefits. First, the logic for each concern is in one place, as opposed to being scattered all over the code base. Second, your service modules are cleaner because they only contain code for their primary concern (or core functionality), and secondary concerns have been moved to aspects.

***Defining AOP terminology***

* Like most technologies, AOP has its own jargon. Aspects are often described in terms of advice, pointcuts, and join points.

**

This figure illustrates how these concepts are tied together.

**ADVICE** When a meter reader shows up at you 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 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****.*

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 be applied only 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 before the advised method is invoked.
  + ***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 advided method is invoked.

**JOIN POINTS**

* An electric company services several houses, perhaps even an entire city. Each house has an electric meter that needs to be read, so each house is a potential target for the meter reader. The meter reader could potentially read all kinds of devices, but to do her job, she 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 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.

***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(..))")

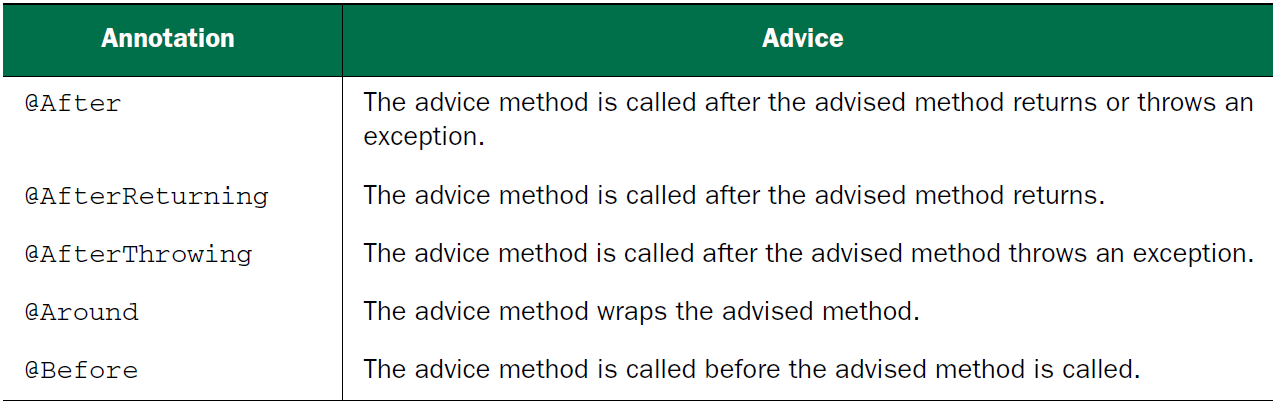
**public** **void** demandRefund() {

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

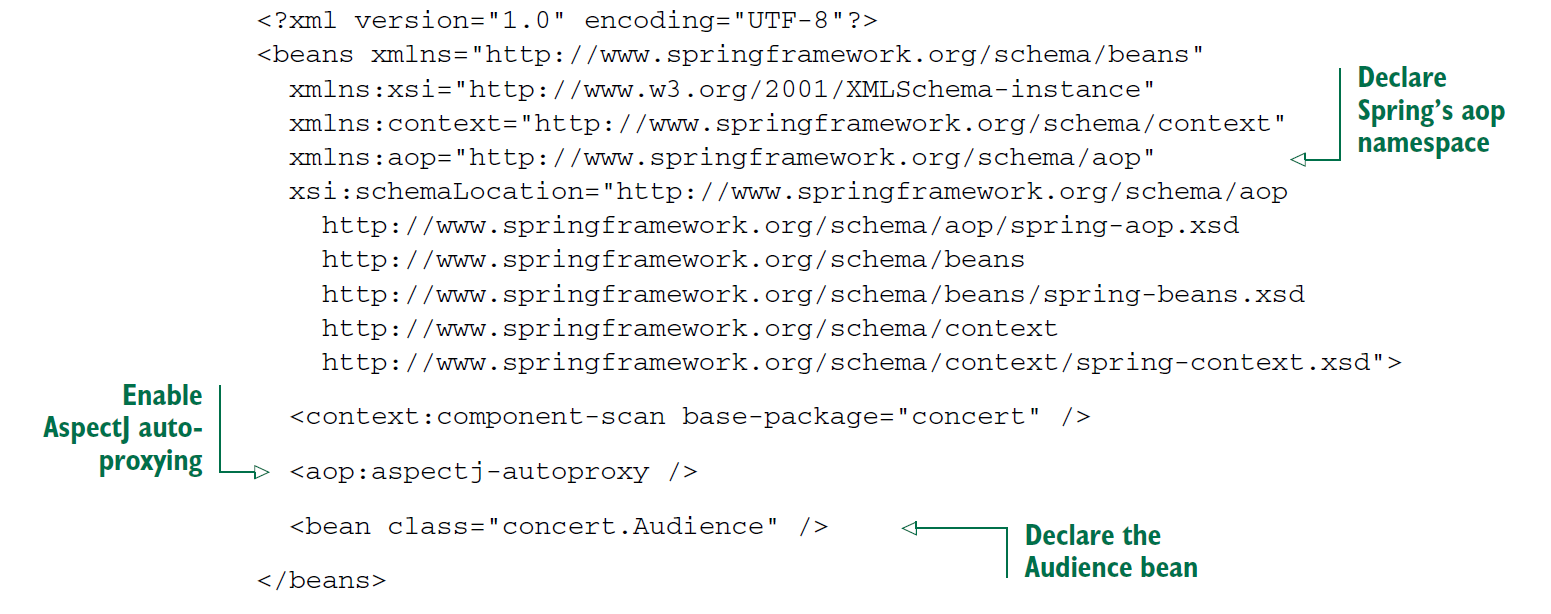
**public** Audiance audiance() {

**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");

}

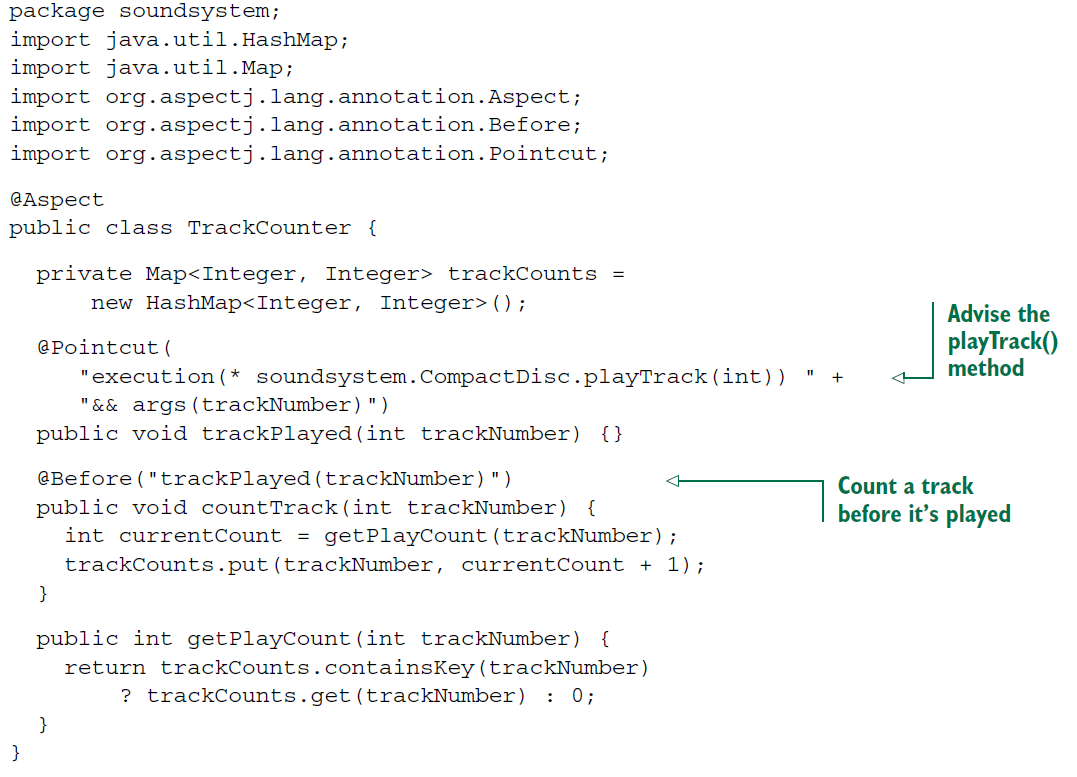
}

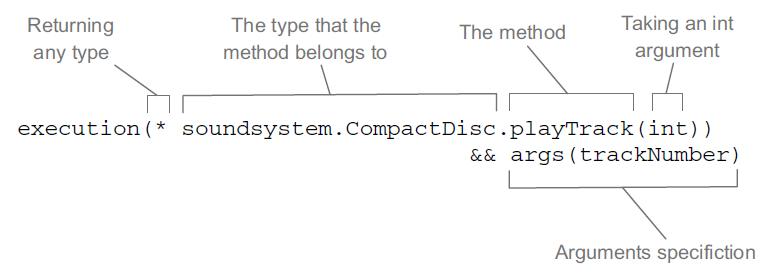
}

* 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.

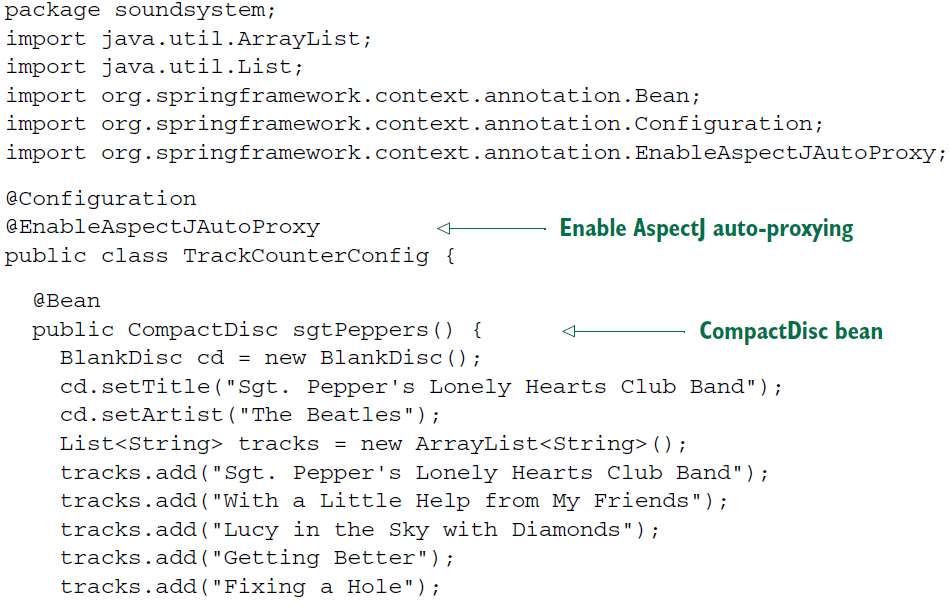
***Handling parameters in advice***

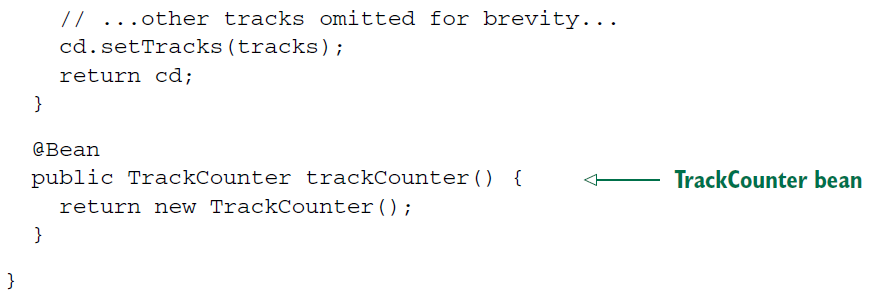
* What if your aspect was to advise a method that does take parameters? Could the aspect access the parameters that are passed into the method and use them?
* To illustrate, let’s revisit the BlankDisc class, the play()method cycles through all the tracks and calls playTrack() for each track. But you 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:

****

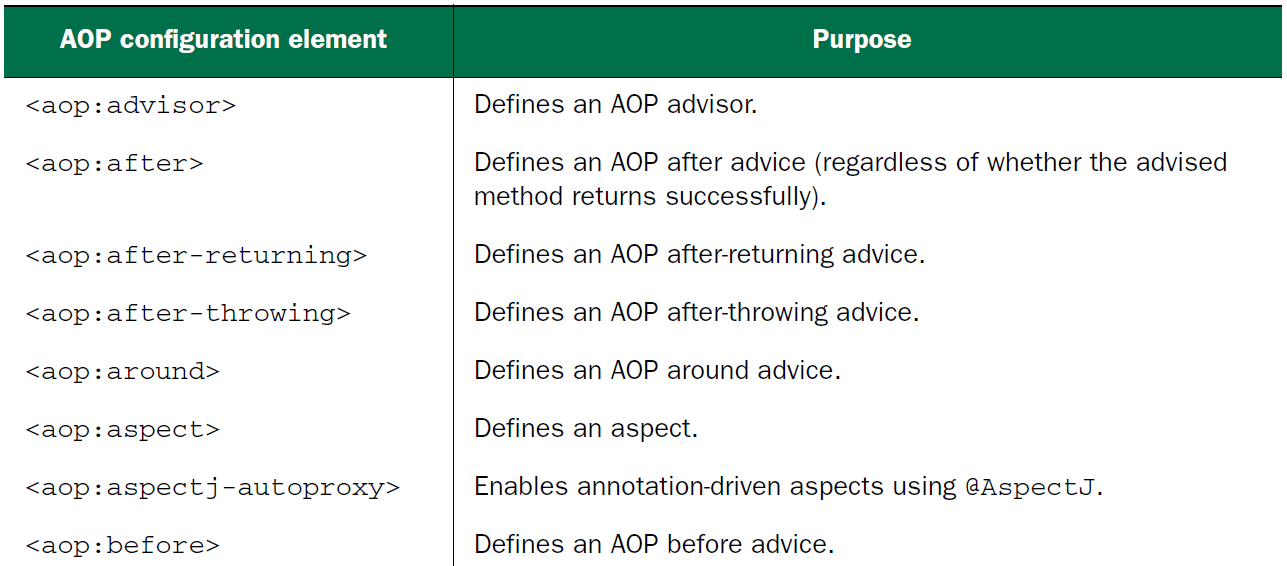
****

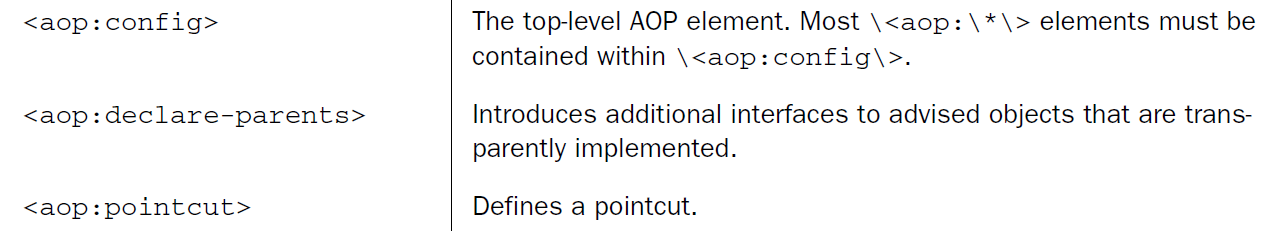
***Declaring aspects in XML***

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

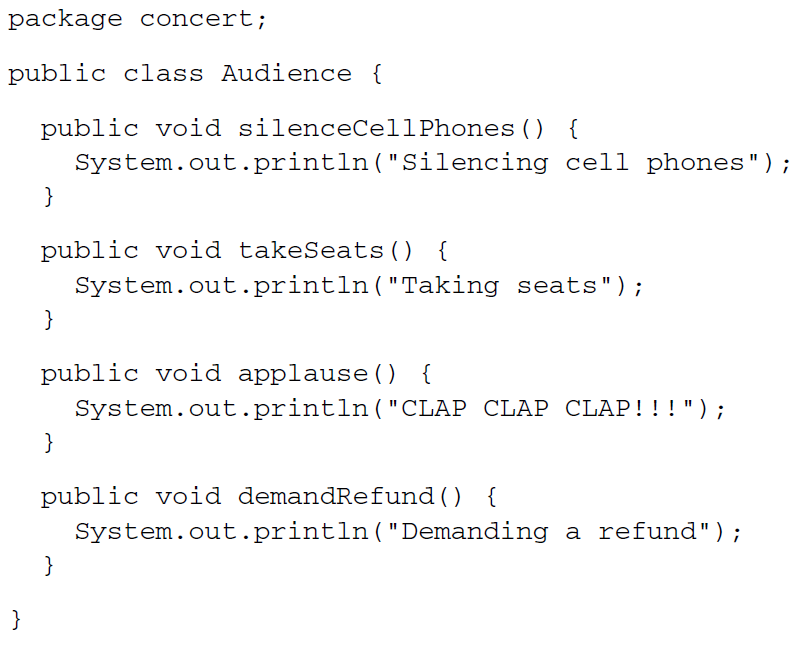
XML configuration.

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



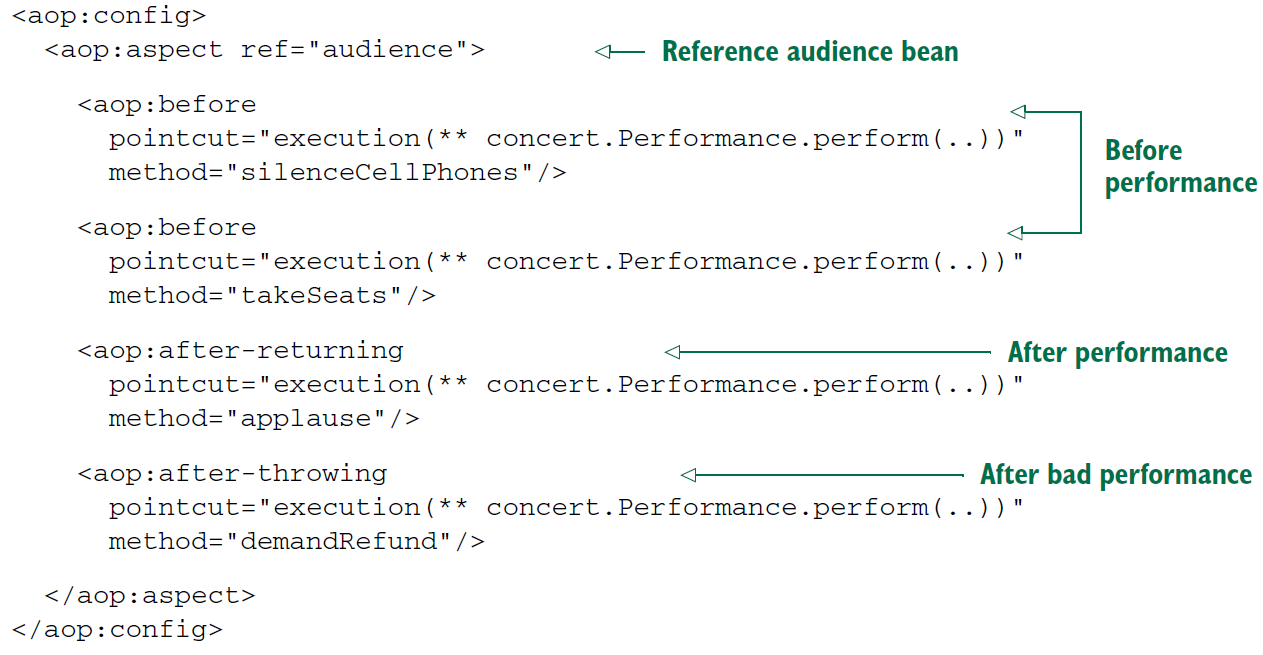


* We’ve already seen the <aop:aspectj-autoproxy> element and how it can enable auto-proxying of AspectJ-annotated advice classes. The other elements in the aop namespace let you declare aspects directly in your Spring configuration without using annotations.
* For example, let’s have another look at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

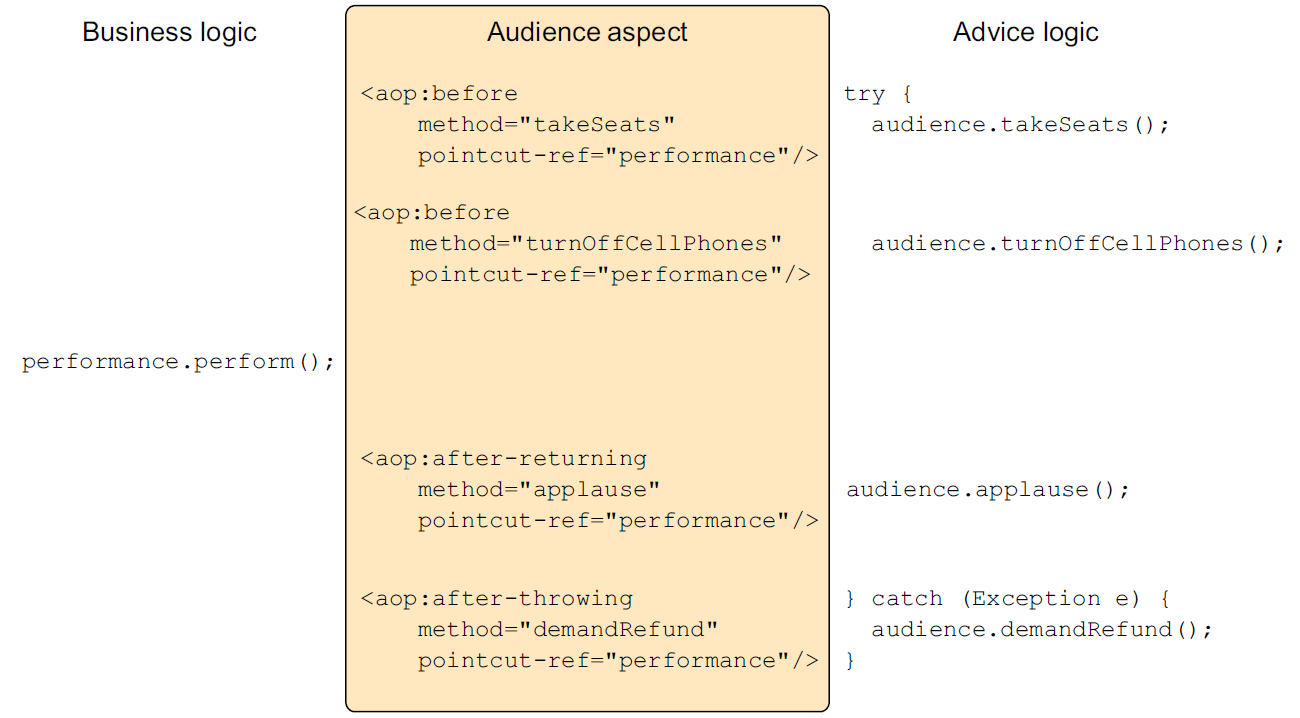


* As you can see, without the AspectJ annotations, there’s nothing remarkable about the Audience class. It’s a basic Java class with a handful of methods. And you can register it as a bean in the Spring application context like any other class.

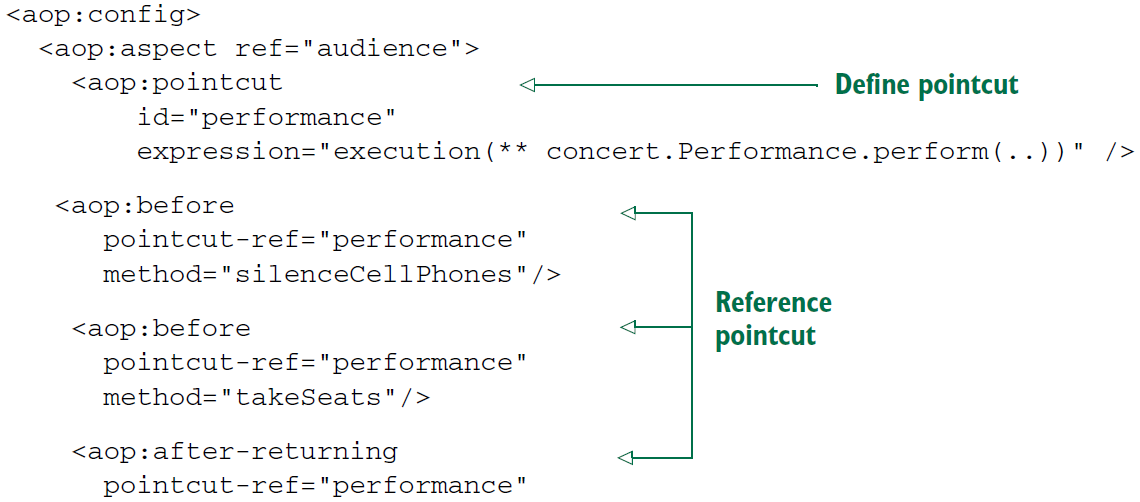
***Declaring before and after advice***

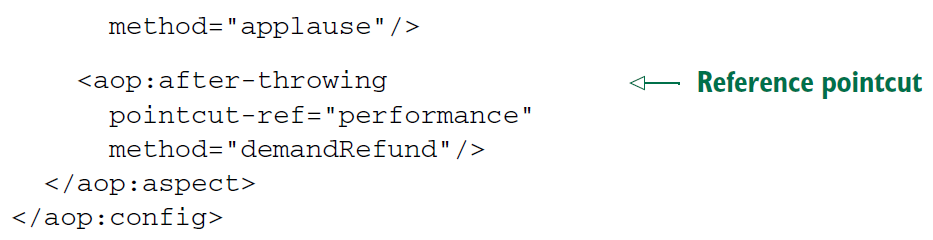


* The first thing to notice about the Spring AOP configuration elements is that most of them must be used in the context of the <aop:config> element.
* There are a few exceptions to this rule, but when it comes to declaring beans as aspects, you’ll aways start with <aop:config>.
* In <aop:config>, you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

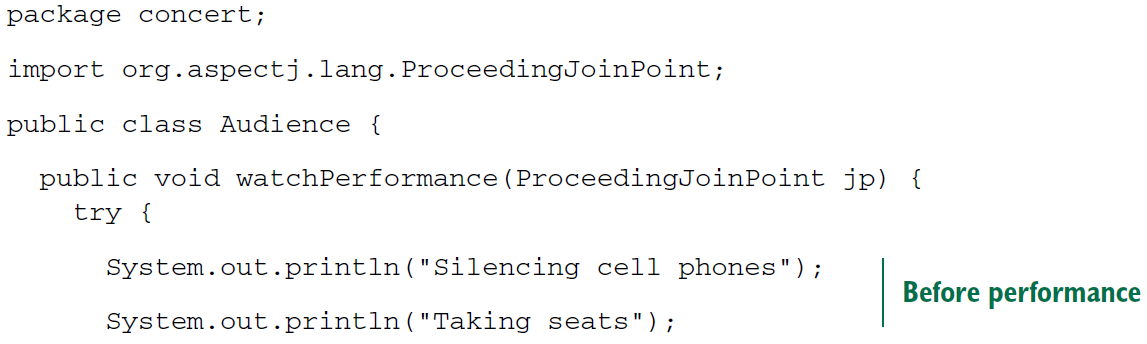


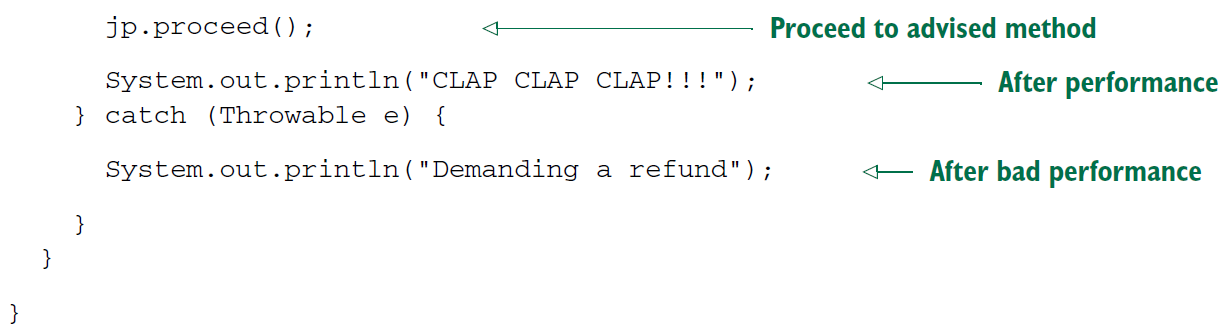


* Now the pointcut is defined in a single location and is referenced across multiple advice elements. The <aop:pointcut> element defines the pointcut to have an id of performance. Meanwhile, all the advice elements have been changed to reference the named pointcut with the pointcut-ref attribute.
* The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

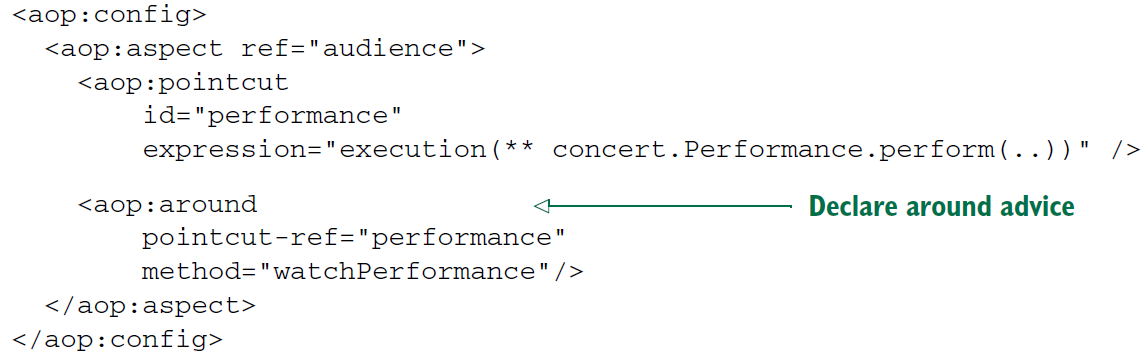
***Declaring around advice***

* 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.





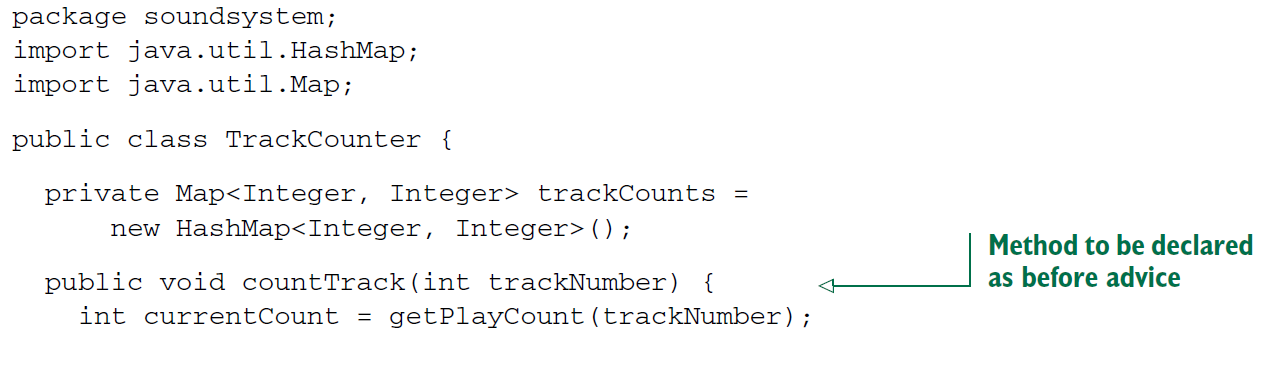
* In the case of the audience aspect, the watchPerformance() method contains all the functionality of the previous four advice methods. But all of it is contained in this single method, and this method is responsible for its own exception handling.
* Declaring around advice isn’t dramatically different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

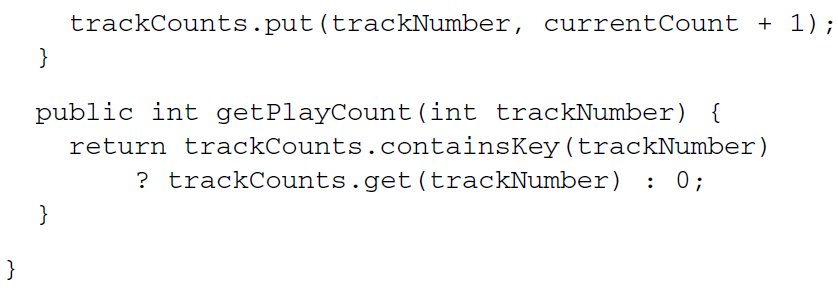


* As with the other advice XML elements, <aop:around> is given a pointcut and the name of an advice method. Here you’re using the same pointcut as before, but you set the method attribute to point to the new watchPerformance() method.

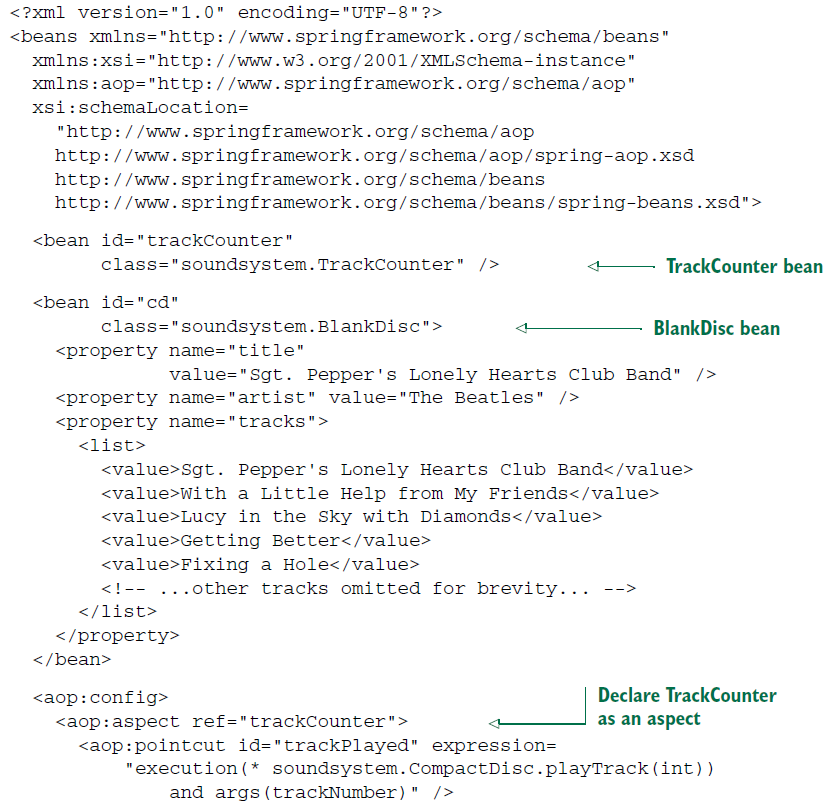
***Passing parameters to advice***

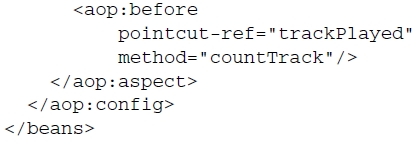
* Now that you’re configuring your aspects in XML, let’s see how you can accomplish the same thing. First, let’s strip all the @AspectJ annotations out of the TrackCounter.





* 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).