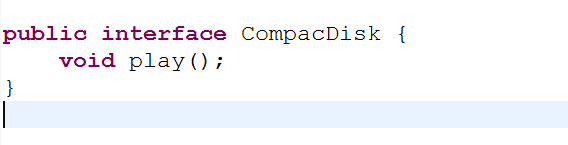
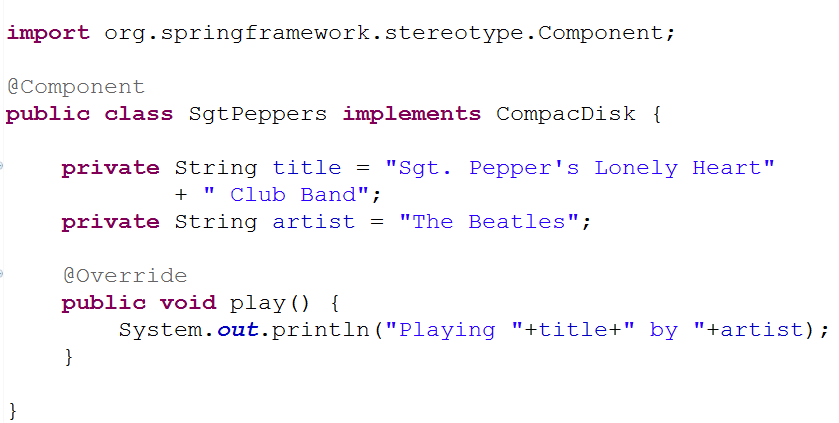
**Automatic Wiring**

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

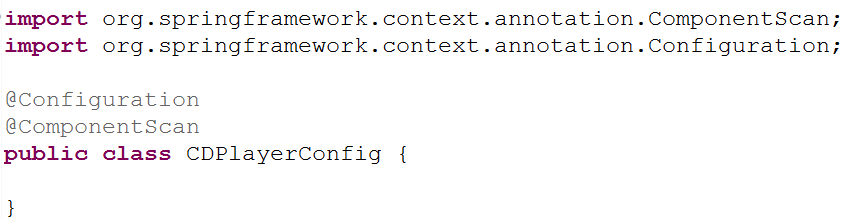
* Spring do automatic wiring from two angles:
* *Component scanning –* Spring automatically discovers beans to be created in the application context.
* *Autowiring –* Spring automatically satisfies bean dependencies.

**Creating discoverable beans**

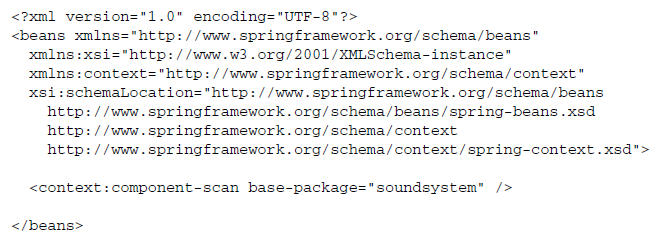




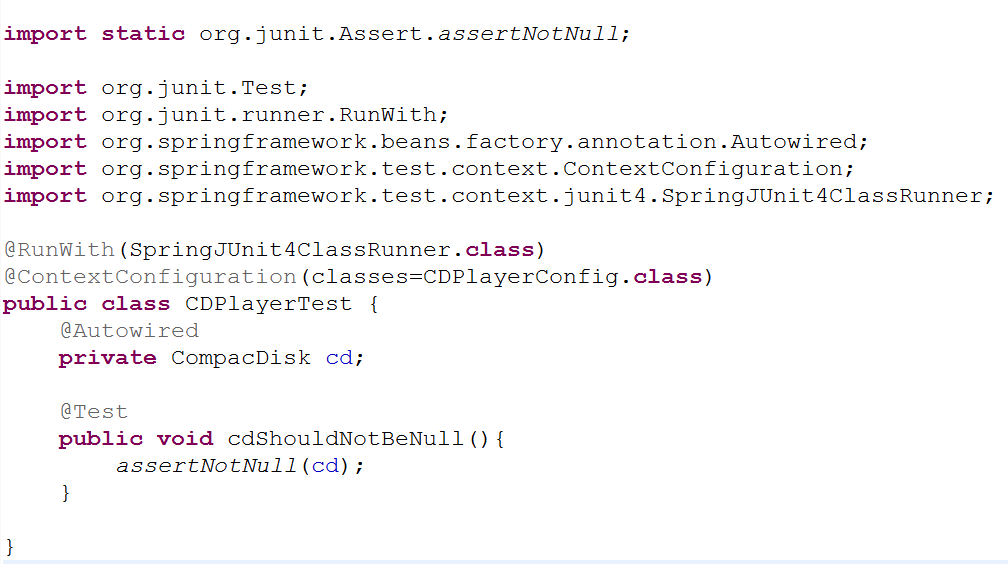
* As with the CompactDisc interface, the specifies of SgtPeppers aren’t important to this discussion. What you should take note of it, is that SgtPeppers is annotated with **@Component**.
* This simple annotation identifies this class as a component class and serves as a component class and serves as 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.



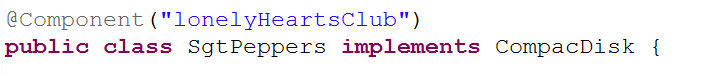
* Even though XML is an option for enabling component scanning, we will focus more on using the preferred Java-based configuration for the remainder of this discussion.
* In XML the *<context:component-scan>* element has attributes and sub-elements that mirror the attributes you’ll use when working with *@Component-Scan*
* To test that component scanning works, let’s write a SimpleJUnit test that creates a Spring application context and asserts that the *ConpactDisk* bean is, in fact, created. CDPlayerTest in the next listing does precisely that.



* CDPlayerTest takes advantage of Spring’s *SpringJUnit4ClasssRunner* to have a Spring application context automatically created when the test starts. And the @*ContextConfiguration* annotation tells it to load its configuration from the CDPlayerConfig class. Because that configuration class includes @*ComponentScan,* the resulting application context should include the *CompactDisc* bean.
* To prove that, the test has a property of type *CompactDisc* that is annotated with *@Autowired* to inject the *ComactDisc* bean into the test. ( we’ll talk more about it later). Finally, a simple test method asserts that the *cd* property isn’t null. If it’s not null, that means Spring was able to discover the *CompactDisc* classs, automatically create it as a bean in the Spring application context, and inject it into the test.

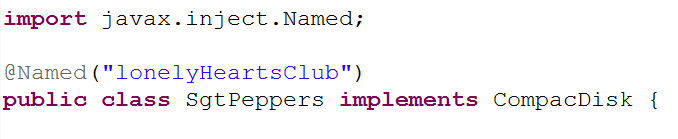
***Naming a component-scanned bean***

* 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:

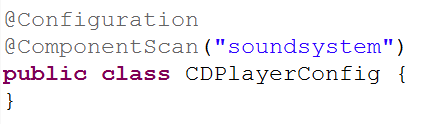


eclipse_2017-06-27_12-18-20.png

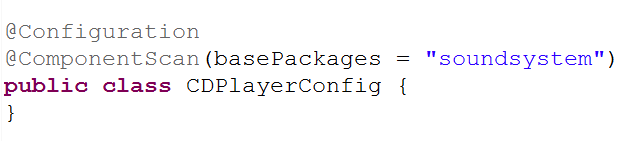
* Spring supports the @Named annotation as an alternative to @Component. There are a few subtle differences, but in most common cases they’re interchangeable.
* The author have a strong preference for the *@Component* annotation, largely because *@Named* is.. well … poorly named. It doesn’t describe 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***

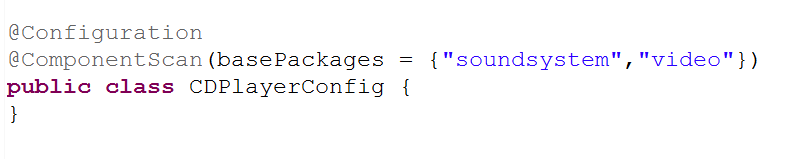
* So far we’ve used *@ComponentScan* with no attributes. That means it will default to the configuration class’s package as its base package to scan for components.
* But what if you want to scan a different package? Or what if you want to scan multiple base packages?
* One common reason for explicitly 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.



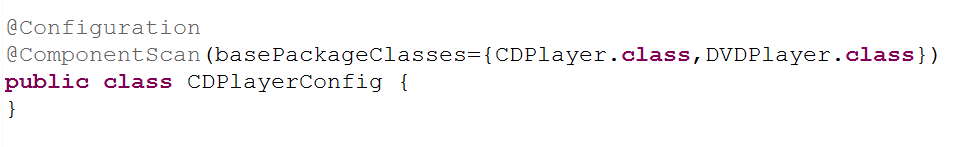
* Or, if you’d rather it be clear that you’re setting the base package, you can do so with the *basePackage* attribure:



* 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:

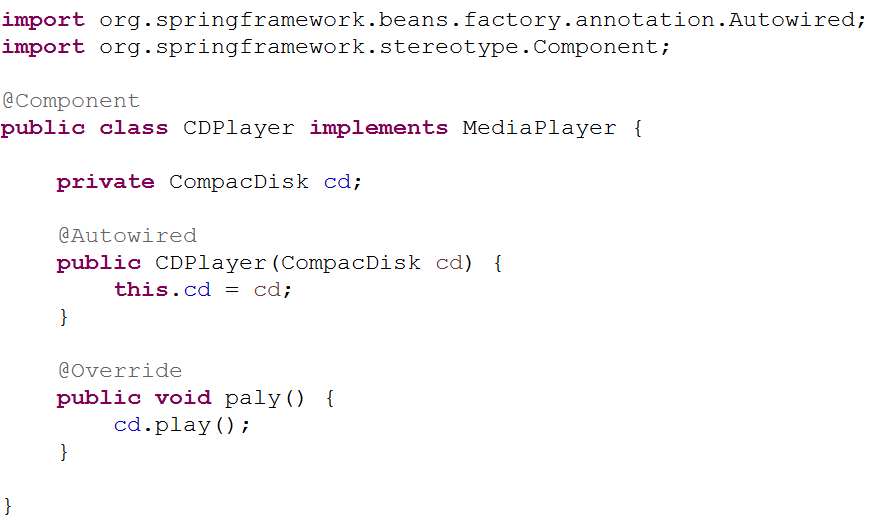


* The basePackages attribute has been replaced with *basePackageClasses*. And instead of identifying the packages with *String* names, the array given to *basePackageClasses* includes classes. Whatever packages those classes are in will used as the package for component scanning.
* You might consider 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).

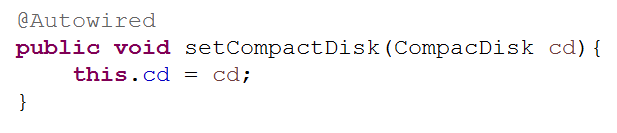
***Annotating beans to be automatically wired***

­­­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.

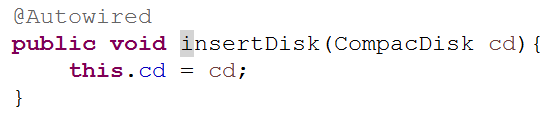
For example, consider the *CDPlayer* class in the following listing. Its constructor is annotated with @Autowired, indicating that when Spring creates the *CDPlayer* bean, it should instantiate it via that constructor and pass in a bean that is assignable to *CompactDisc.*



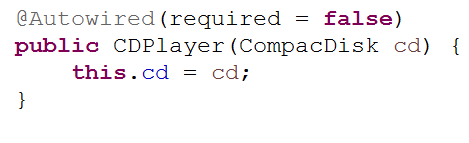
* The *@Autowired*  annotation’s use isn’t limited to constructors. It can also be used on a property’s setter method. For example, if CDPlayer had a *setCompactDisk()* method, you might annotate it for autowiring like this:



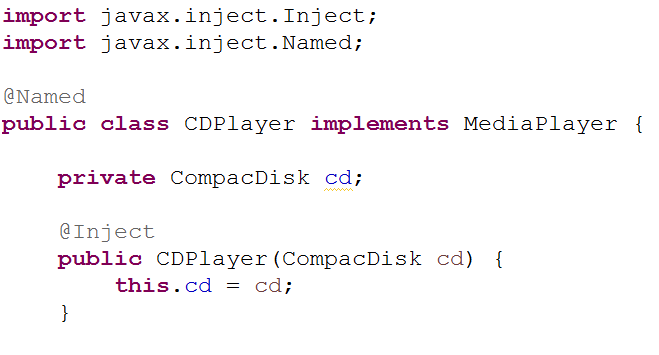
* After Spring has instantiated the bean, it will try to satisfy the dependencies expressed through methods such as the *setCopactDisc()* method that are annotated with *@Autowired.*
* There’s nothing special about setter methods. @*Autowired* can also beapplied on any method on the classs. Pretending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