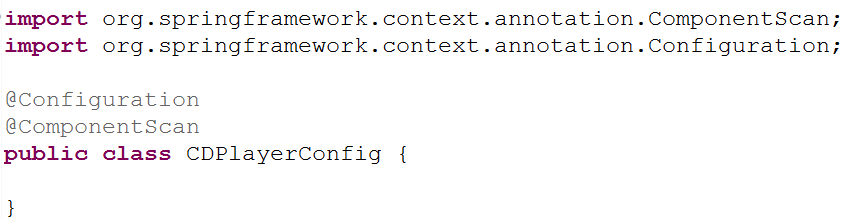
***Wiring beans with Java***

* 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.

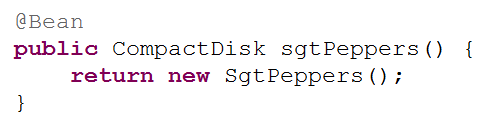
***Creating a configuration class***

Let’s revisit *CDPlayerConfig* from that example:

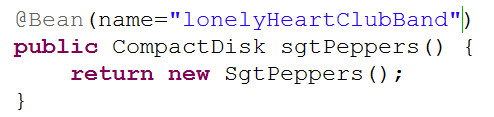
****

* 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.*

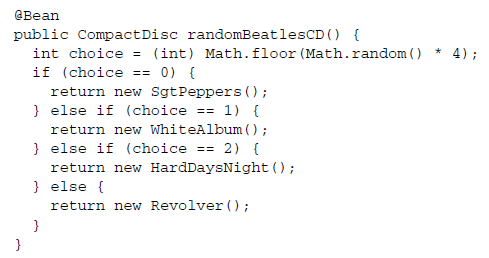
***Declaring a simple bean***

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:

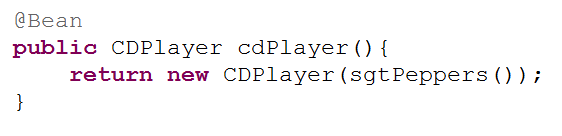


* No matter how you name the bean, this bean declaration is about as simple as they come. The body of the method returns a new instance of *SgtPeppers*. But because it’s expressed in Java, it has every capability afforded it by the Java language to do almost anything to arrive at the *CompactDisc* that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

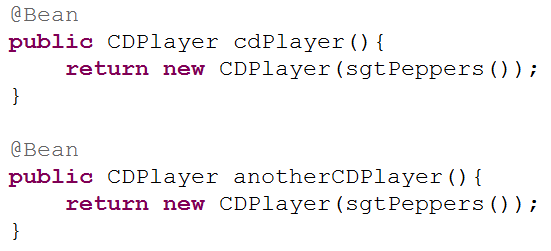


***Injecting with JavaConfig***

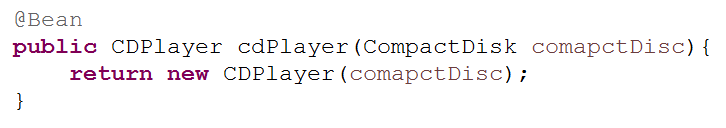
* You must declare the *CDPlayer* bean, which depends on a *CompactDisc.* How can you wire that up in JavaConfig?
* The simplest way to wire up beans in JavaCongig is to refer to the reference bean’s method. For example, here’s how you might declare the CDPlayer bean:



* 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:

