**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.

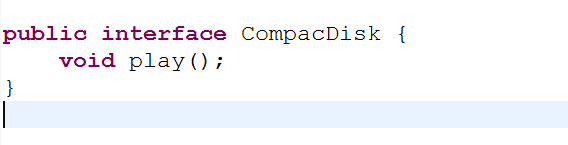
* It’s great that you have many choices about how to wire beans in Spring, but at some point you must select one.
* Spring’s configuration styles are mix-and-match, so you could choose XML to wire up some beans, Use Spring’s Java-based configuration (JavaConfig) for other beans, and let other beans be 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.

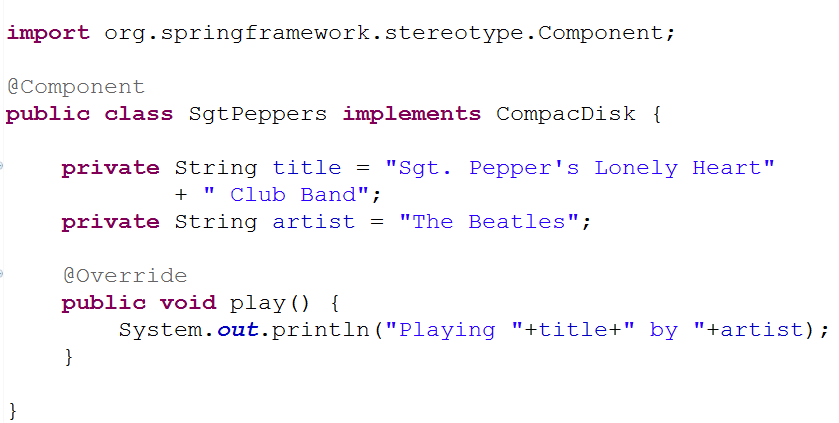
**Automatic Wiring**

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

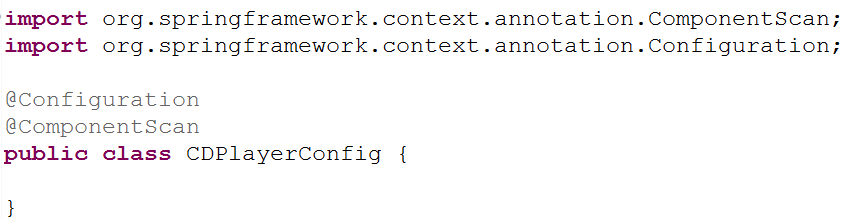
* Spring do automatic wiring from two angles:
* *Component scanning –* Spring automatically discovers beans to be created in the application context.
* *Autowiring –* Spring automatically satisfies bean dependencies.

**Creating discoverable beans**

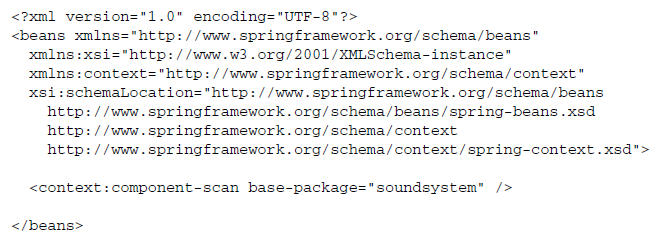




* As with the CompactDisc interface, the specifies of SgtPeppers aren’t important to this discussion. What you should take note of it, is that SgtPeppers is annotated with **@Component**.
* This simple annotation identifies this class as a component class and serves as a component class and serves as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



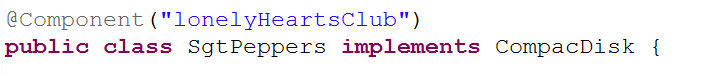
* The CDPlayerConfig class defines a Spring wiring specification, expressed in Java.
* For Now, observe that CDPlayerConfig doesn’t explicitly define any beans itself. Instead, it’s annotated with @ComponentScan to enable component scanning in Spring.
* @ComponentScan will default to scanning the same 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*

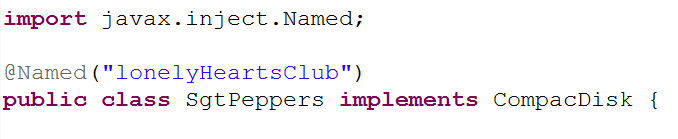
***Naming a component-scanned bean***

* 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:

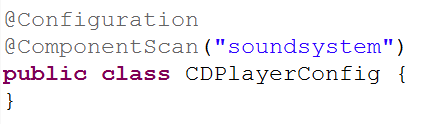


eclipse_2017-06-27_12-18-20.png

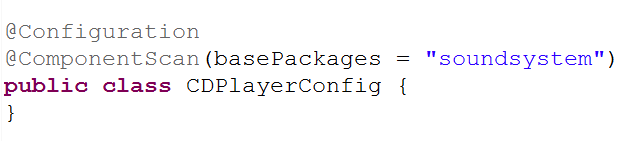
* Spring supports the @Named annotation as an alternative to @Component. There are a few subtle differences, but in most common cases they’re interchangeable.
* The author have a strong preference for the *@Component* annotation, largely because *@Named* is.. well … poorly named. It doesn’t describe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

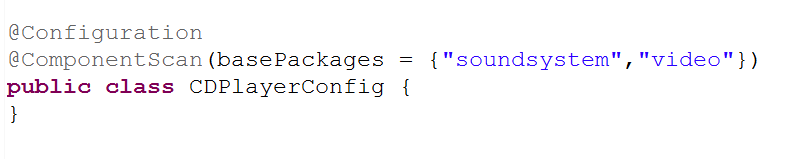
* So far we’ve used *@ComponentScan* with no attributes. That means it will default to the configuration class’s package as its base package to scan for components.
* But what if you want to scan a different package? Or what if you want to scan multiple base packages?
* One common reason for explicitly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



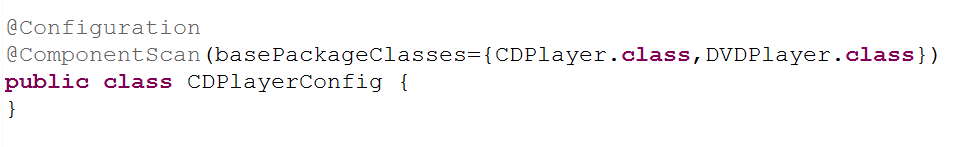
* Or, if you’d rather it be clear that you’re setting the base package, you can do so with the *basePackage* attribure:



* 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:

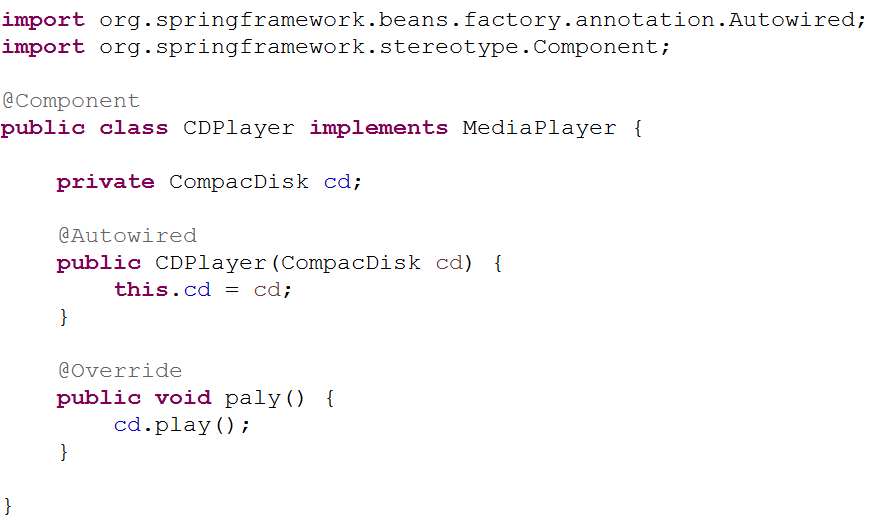


* The basePackages attribute has been replaced with *basePackageClasses*. And instead of identifying the packages with *String* names, the array given to *basePackageClasses* includes classes. Whatever packages those classes are in will used as the package for component scanning.
* You might consider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

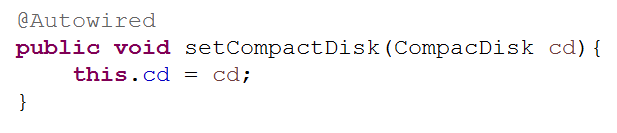
***Annotating beans to be automatically wired***

­­­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.

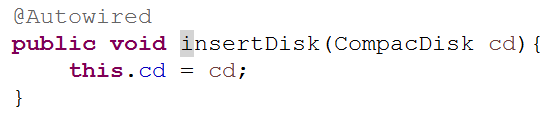
For example, consider the *CDPlayer* class in the following listing. Its constructor is annotated with @Autowired, indicating that when Spring creates the *CDPlayer* bean, it should instantiate it via that constructor and pass in a bean that is assignable to *CompactDisc.*



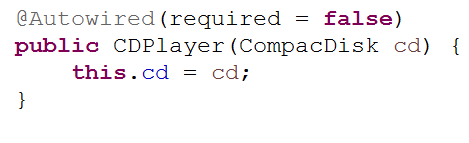
* The *@Autowired*  annotation’s use isn’t limited to constructors. It can also be used on a property’s setter method. For example, if CDPlayer had a *setCompactDisk()* method, you might annotate it for autowiring like this:



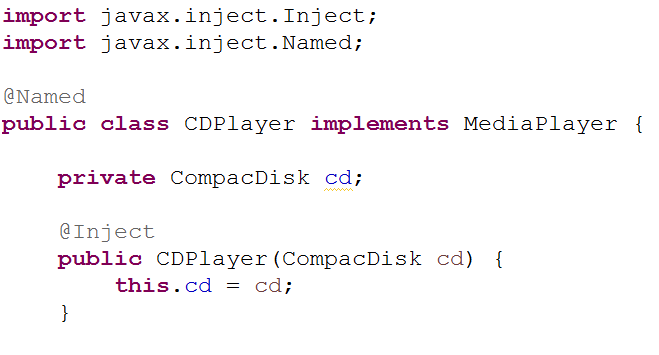
* After Spring has instantiated the bean, it will try to satisfy the dependencies expressed through methods such as the *setCopactDisc()* method that are annotated with *@Autowired.*
* There’s nothing special about setter methods. @*Autowired* can also beapplied on any method on the classs. Pretending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



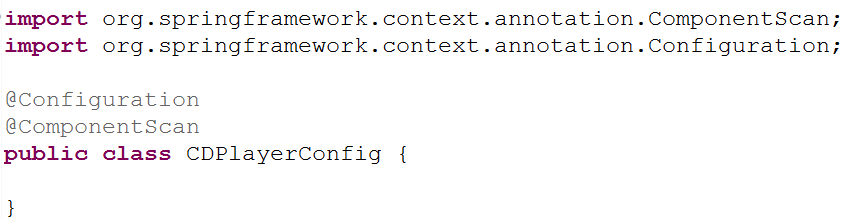
* *@Inject* comes from Java Dependency Injection specification, the same specification that gave us *@Named.* Spring supports the @*Inject* annotation for autowring alongside its own *@Autowired.* Although there are some subtle differences between *@Inject* and @*Autowired,* they’re interchangeable in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

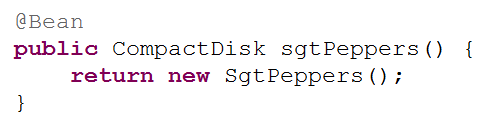
***Creating a configuration class***

Let’s revisit *CDPlayerConfig* from that example:

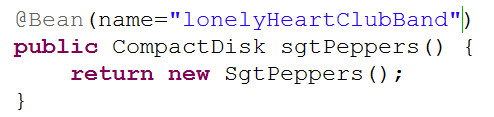


* The key to creating a JavaConfig class is to annotate it with *@Configuration*. The *@Configuration* annotation identifies this as a configuration class, and it’s expected to contain details on beans that are to be created in the Spring application context.
* If you remove the *@Component* annotation 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.*

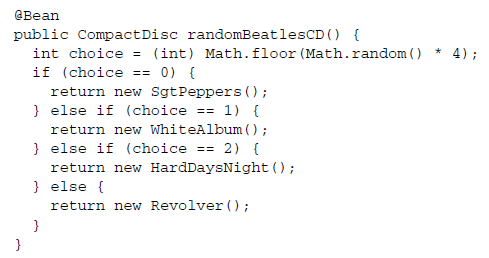
***Declaring a simple bean***

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:

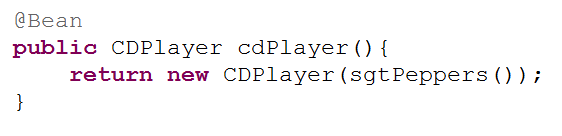


* No matter how you name the bean, this bean declaration is about as simple as they come. The body of the method returns a new instance of *SgtPeppers*. But because it’s expressed in Java, it has every capability afforded it by the Java language to do almost anything to arrive at the *CompactDisc* that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

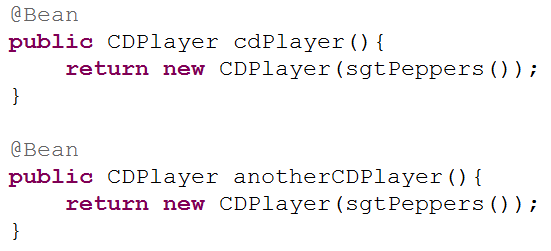


***Injecting with JavaConfig***

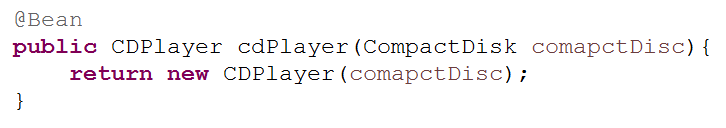
* You must declare the *CDPlayer* bean, which depends on a *CompactDisc.* How can you wire that up in JavaConfig?
* The simplest way to wire up beans in JavaCongig is to refer to the reference bean’s method. For example, here’s how you might declare the CDPlayer bean:



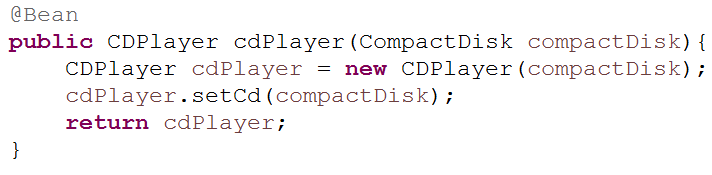
* The *cdPlayer*() method, like the *sgtPeppers*() method, is annotated with *@Bean* to indicate that it will produce an instance of a bean to be registered in the Spring application context. The ID of the bean will be *cdPlayer*, the same as the method’s name.
* The body of the *cdPlayer()* method differs 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:



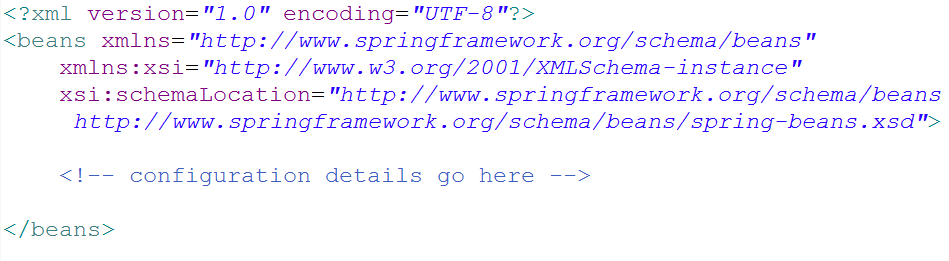
***Wiring beans with XML***

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.*