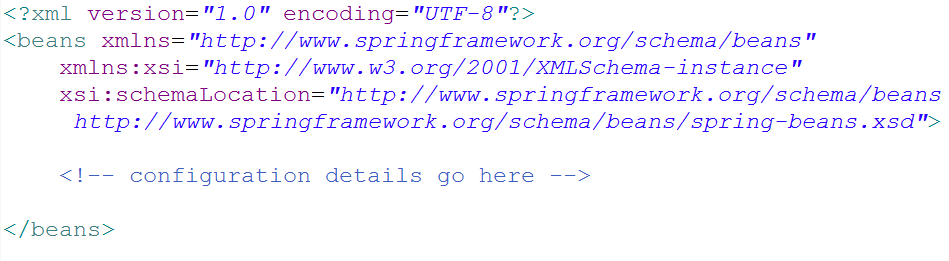
***Wiring beans with XML***

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?

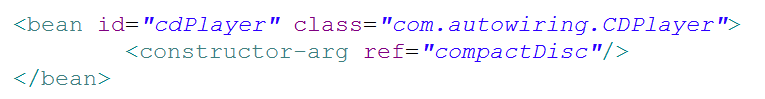
**CHECK XML VALIDITY WITH AN IDE:** *Using a Spring-aware IDE such as Spring Tool Suite can help a lot to ensure the validity of your Spring XML configuration.*

***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.

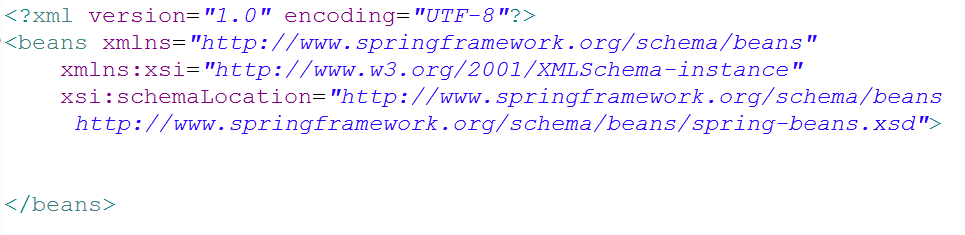
**INJECTING CONSTRUCTORS WITH BEAN REFERENCES**

* 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.*

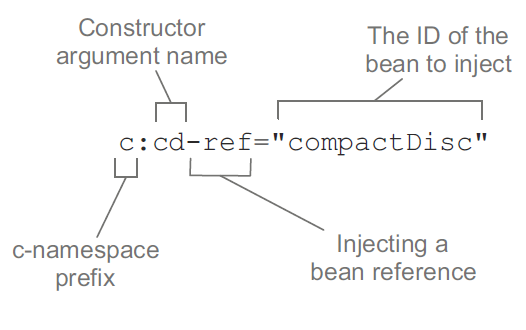
* The *<constructor-arg>* element tells it to pass a reference to the bean whose ID is *compactDisc* to the *CDPlayer’s* constructor.
* Alternatively, you can use Spring’s c-namespace. The c-namespace was introduced in Spring 3.0 as a more succinct way of expressing constructor args in XML. To use it, you must declare its schema in the preamble of the XML, like this:



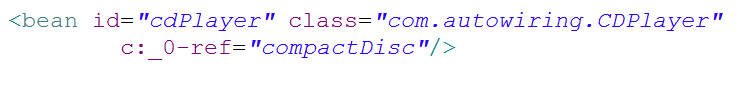
* With the c-namespace and schema declared, you can use it to declare a constructor argument like this:

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

* Here you’re using the c-namespace to declare a constructor argument as an attribute of the *<bean>* element. And it’s a rather odd-looking attribute name.



* The attribute name starts with *c:,* the namespace prefix.
* Following that is the name of the constructor argument being wired.
* After that *–ref,* a naming convention that indicates to Spring that you’re wiring a reference to a bean named *compactDisc* and not the literal *String* value “*compactDisc”.*
* It’s clear that using c-namespace attributes is much more shorter than using the *<constructor-arg>* element*.*
* One bug is it directly refers to the name of the constructor argument. Instead, you could refer to the parameter’s position in the parameter list:



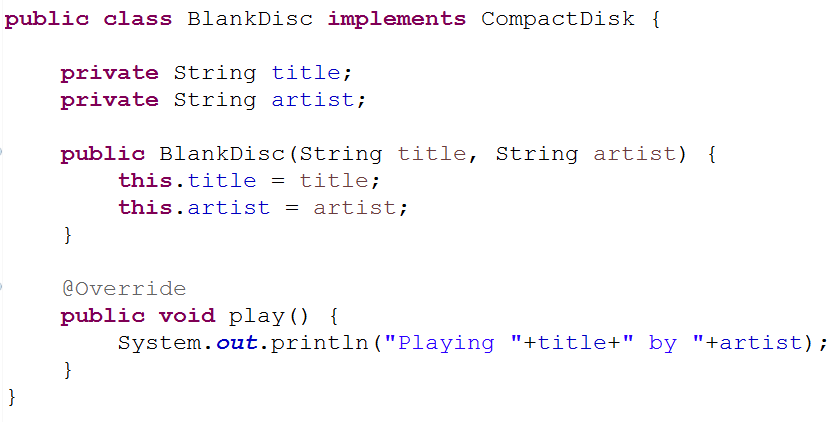
* The name of the parameter is replaced with 0, the parameter index, But because XML doesn’t allow digits as the first character of an attribute, I had to add an underscore as a prefix.
* If you have only one constructor argument, you have one more option – don’t identify the parameter at all:

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

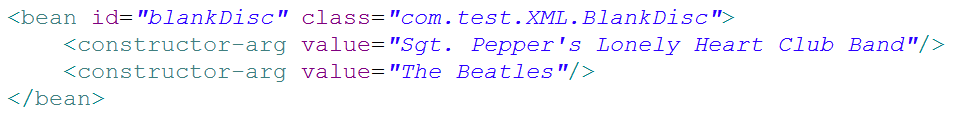
* There’s no parameter index or parameter name. There’s just an underscore placeholder followed by *–ref* to indicate that you’re wiring a reference.

**INJECTING CONSTRUCTORS WITH LITERAL VALUES**

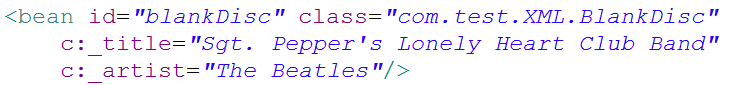
* Sometimes you need to do is configure an object with a literal value, To illustrate, suppose you were to create a new implementation of *CompactDisc,* as show here:

****

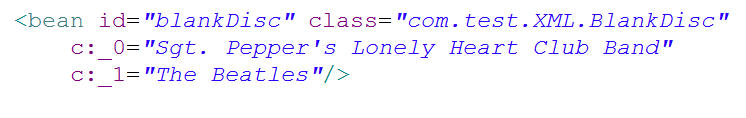
* Unlike *SgtPeppers*, which was hard-caded with a title and artis, this implementation of *CompactDisc* is considerably more flexible.

****

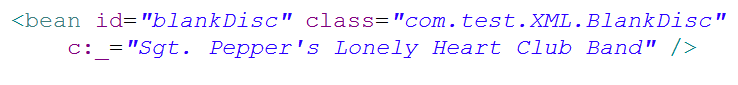
* Once again, the *<constructor-arg>* element is used to inject into constructor arguments. But this time, instead to using the *ref* attribute to reference another bean, you use the *value attribute* to indicate that the given value is to be taken literally and injected into the constructor.
* If you were to use c-namespace attributes instead? One possible rendition might reference the constructor arguments by name:

****

* As you can see, wiring literal values via the c-namespace differs from wiring reference in that the *–ref* suffix is left off the attribute name. Similarly, you could wire the same literal values using parameter indexes, like this:

****

* XML doesn’t allow more than one attribute on a given element to share the same name. Therefore, you can’t use the simple underscore when you have two or more constructor arguments. But you can use it when there’s only one constructor argument. For the sake of completeness, let’s pretend that *BlankDisc* has a single argument constructor that takes the album’s title. In that case, you could declare it in Spring like this:

****

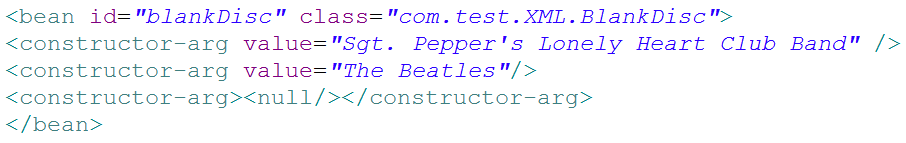
* When it comes to wiring bean reference and literal values, both *<constructor-arg>* and the c-namespace attributes are equally capable. But there’s one thing that *<constructor-arg>* can do that the c-namespace can’t do is to wire collection to constructor arguments.

**WIRING COLLECTIONS**

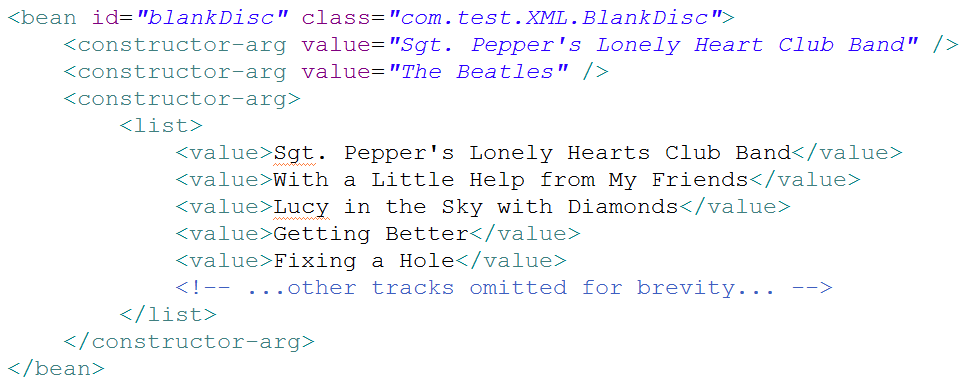
If CompactDisc is to truly model a real-world CD, then it must also have the notion of a list of tracks. Consider the following new *BlankDisc* shown here:



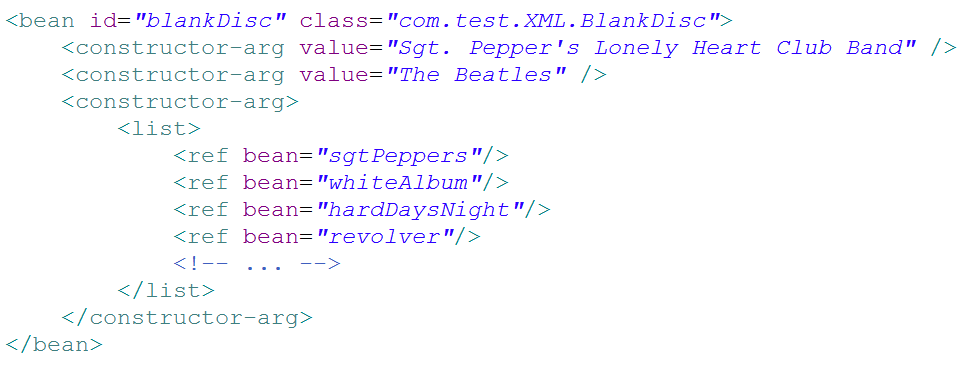
* This change has implication for how you configure the bean in Spring. You must provide a list of tracks when declaring the bean.
* The simplest thing you could do is leave the list null. Because it’s a constructor argument, you must specify it, but you can still pass *null* like this:



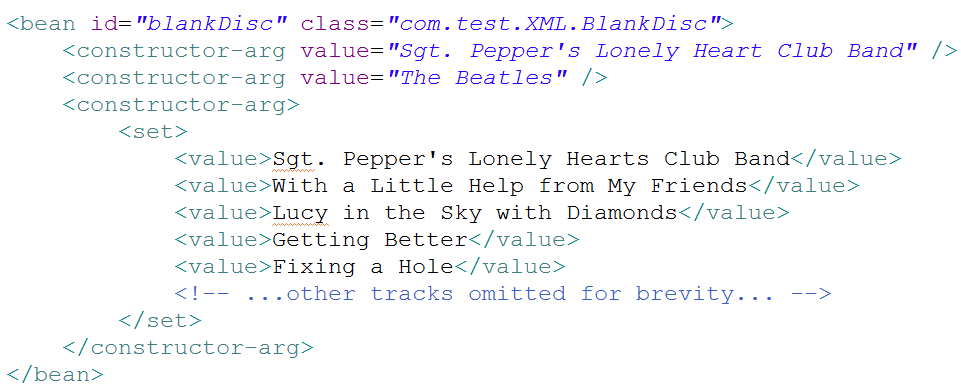
* The *<null/>* element does as you’d expect: it passes null into the constructor. It’s a dirty fix, but it will work at injection time. You’ll get a *NullPointerException* when the play() method is called, so it’s far from ideal.
* A better fix would be to supply a list of track names. For that you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



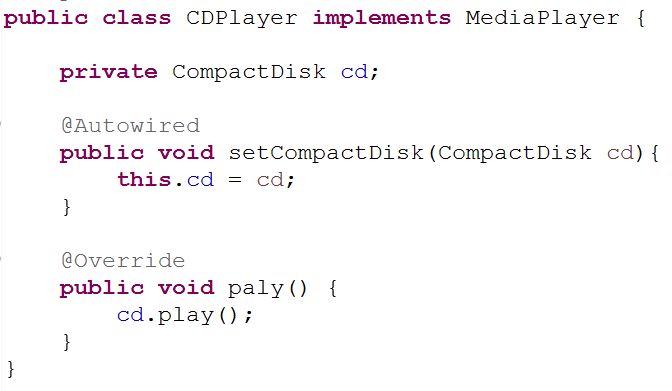
* It makes sense to use *<list>* when wiring a constructor argument of type *java.util.List.* Even so, you could also use the *<set>* element in the same way:



* 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.
* Wiring collections is one place where the *<constructor-arg>* has an advantage over the c-namespace attributes. There’s no obvious way to wire collection like this via c-namespace attributes.

***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:

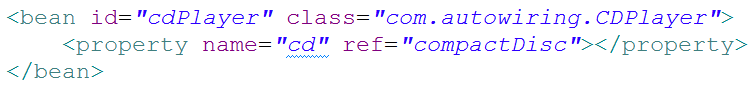


**CHOOSING BETWEEN CONSTRUCTOR INJECTION AND PROPERTY INJECTION.** As a general rule, constructor injection for hard dependencies and property injection for any optional dependencies.

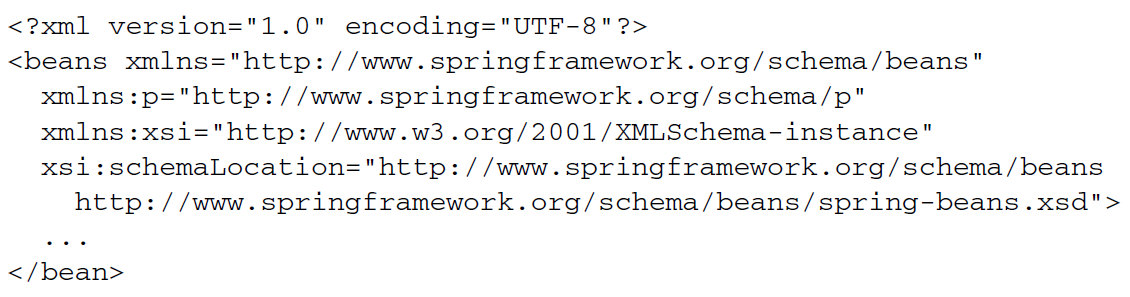
* Now that *CDPlayer* doesn’t have any constructor (aside from the implicit default constructor) ,it also doesn’t have any hard dependencies. Therefore, you could declare it as a Spring bean like this:

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

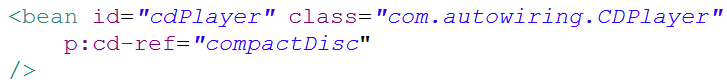
* Spring will have no problem creating that bean. Your *CDPlayerTest* would fail with a *NullPointerException,* however, because you never injected *CDPlayer’s compactDisc* property. But you can fix that with the following change to the XML:



* 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:



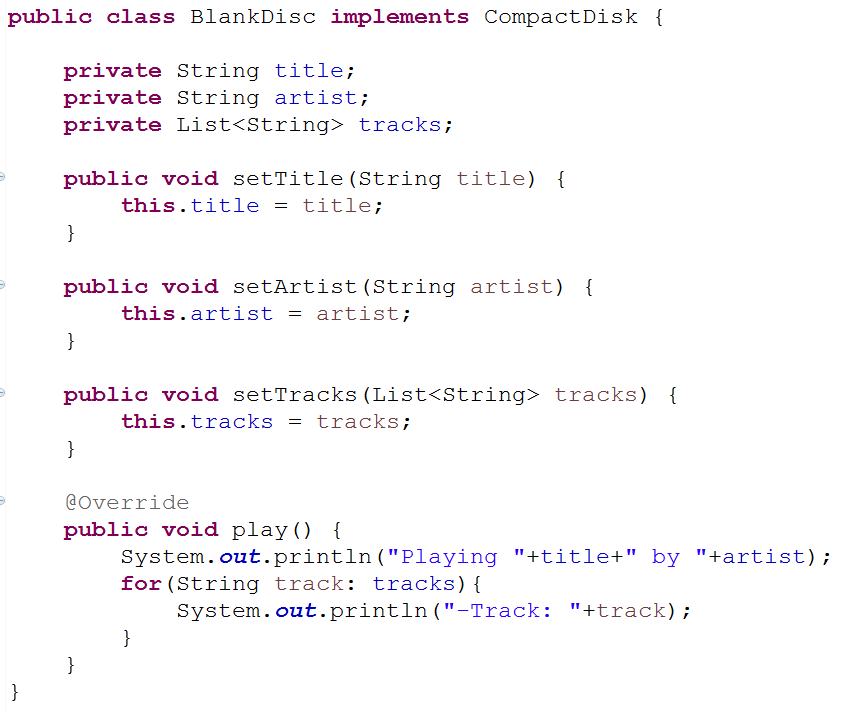
* Using the p-namespace, you can wire the *compactDisc* property like this:



* The p-namespace attributes follow a naming convention similar to that of the c-namespace attributes.
* First, the attribute name is prefixed with *p:* to indicate that you’re setting a property.
* Next up is the name of the property to be injected. Finally, the name ends with –ref as a clue to Spring that you’re wiring a reference to a bean and not a literal value.

**INJECTING PROPERTIES WITH LITERAL VALUES**

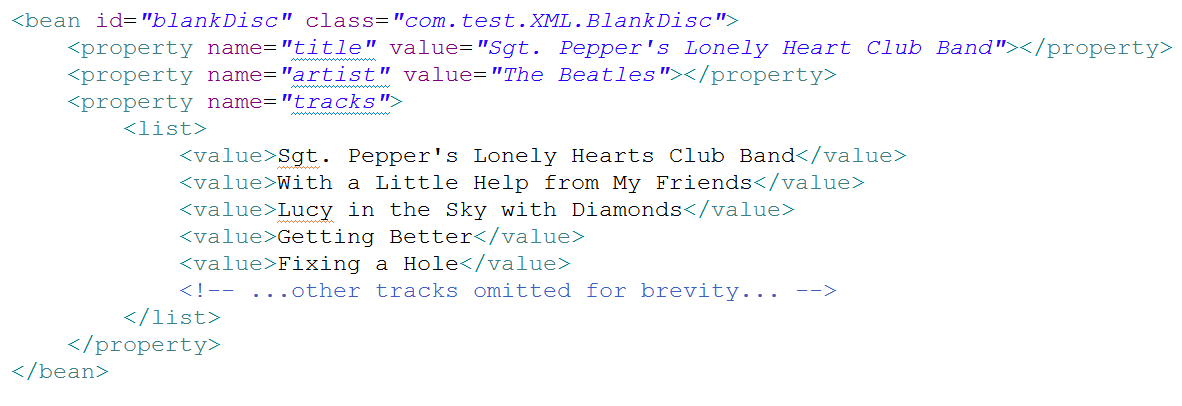
* Properties can be injected with literal values in much the same way as constructor arguments. As an example, let’s revisit the *BlankDisk* bean. This time, however, *BlankDisks* will be configured entirely by property injection, not constructor injection. The new *BlankDisc* class looks like this:

****

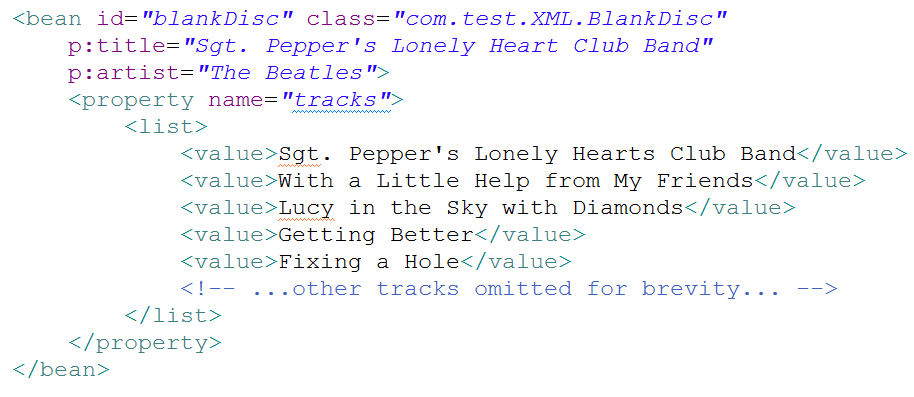
* Now you are no longer obligated to wire any of these properties. You could create a *BlankDisc* bean in its most blank form as follows:

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

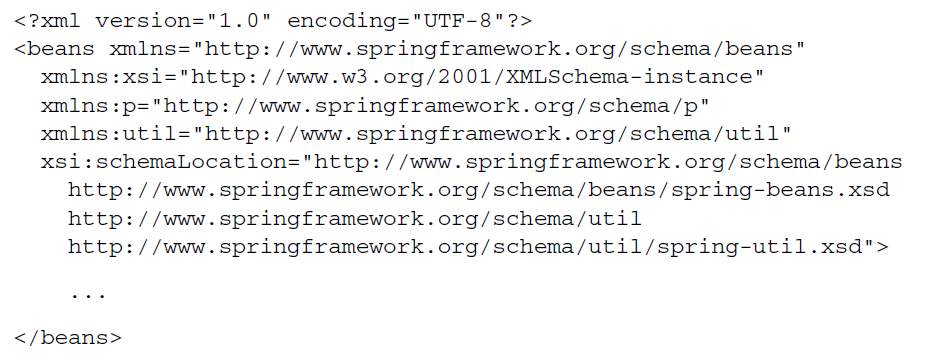
* Of course, wiring the bean without setting those properties wouldn’t play out well at runtime. The *play()* method would claim that it’s playing null by null just before a *NullPointerException* is thrown because there are no tracks. Therefore, you probably should wire up those properties. You can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

**

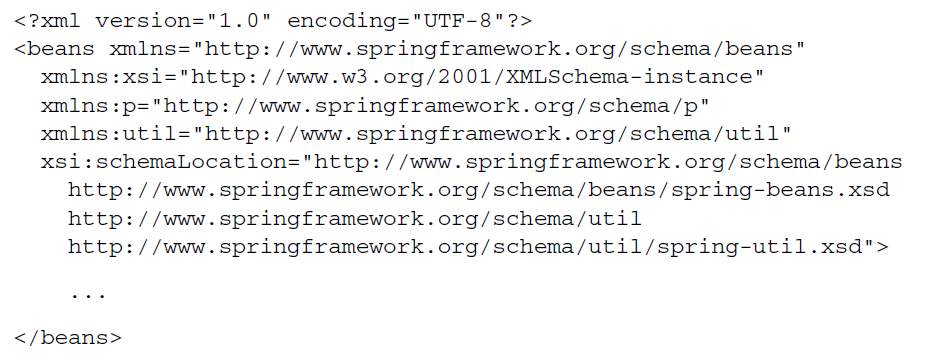
* Aside from using the *<property> element’s value* attribute to set the *title* and *artist properties, notice how you set* the *tracks* property with a nested *<list>* element, the same as before when wiring the tracks through *<constructor-arg> .*
* Optionally, you can accomplish the same thing using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



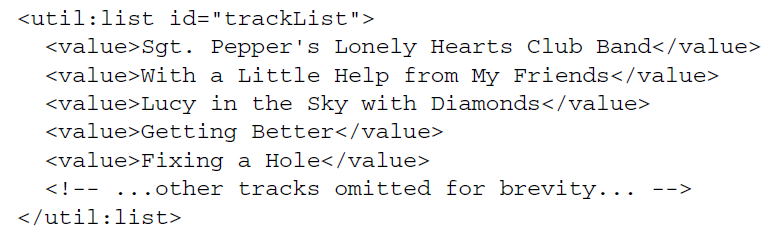
* As with c-namespace attributes, the only difference between wiring a bean reference and wiring a literal value is the presence or absence of a –ref suffix. Without the –*ref* suffix, you’re wiring literal values.
* Notice, however, that you can’t use the p-namespace when wiring a collection. Unfortunately, there’s no convenient way to specify a list of values (or bean references) with the p-namespace. But you can take advantabe of something from Spring’s util-namespace to simplify the *BlankDisc* bean.
* You need to declare the *util-namespace* and its schema in the XML:



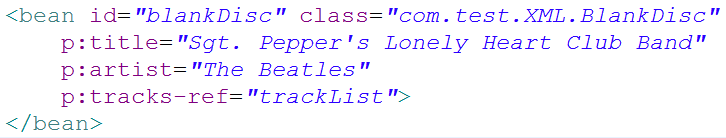
* One of the things that the *util-namespace* offers is the <util:list> element, which creates a list bean. Using *<util:list>,* you can shift the track list out of *BlankDisc* bean and into a bean of its own, like this:



* One of the things that the *util-namespace offers* is the *<util:list>* element, which creates a list bean. Using *<util:list>*, you can shift the track list out of the *BlankDisc* bean and into a bean of its own, like this:



Now you can wire the track-list bean into the *BlankDisc bean’s* tracks property just like any other bean:



* The *<util:list>* element is just one of several elements in the *util-namespace.*

