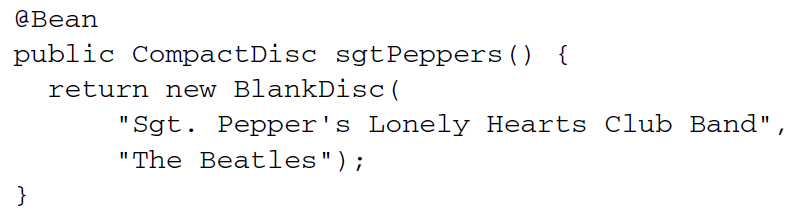
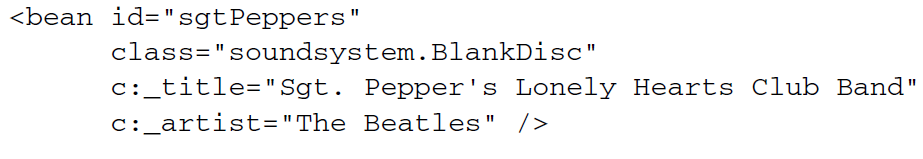
***Runtime value injection***

When we talk about dependency injection and wiring, we’re often talking about wiring a bean reference into a property or constructor argument of another bean. It’s often about associating one object with another object.

You might wired up a *BlankDisc* like this:

**

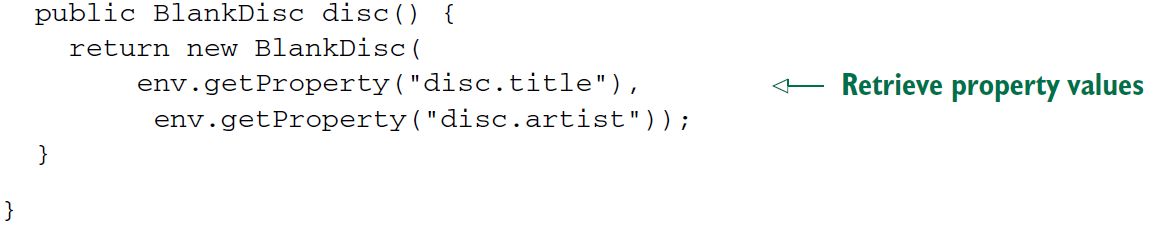
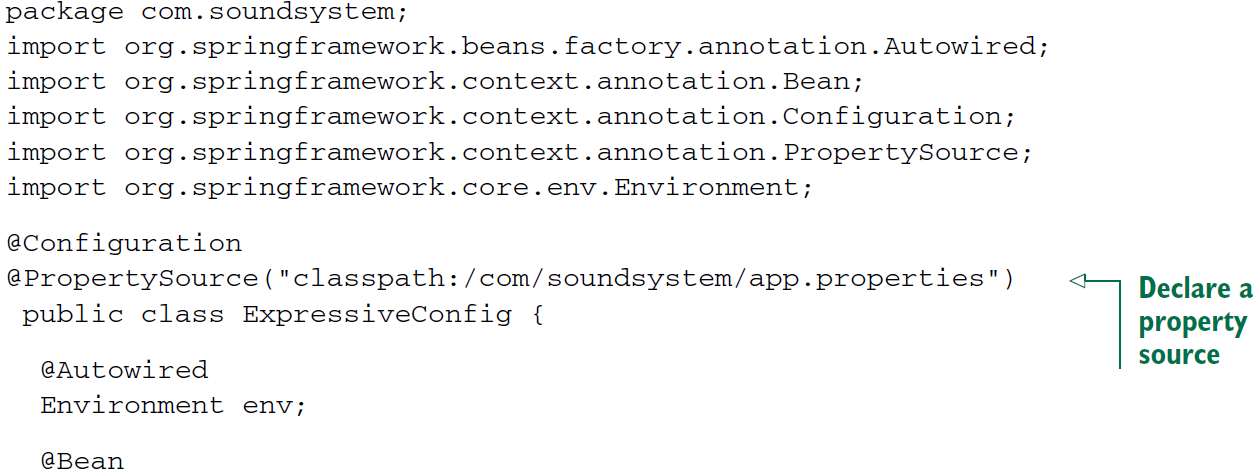
* Although this accomplished what you needed, setting the title and artist for the *BlankDisc* bean, it did so with values hard-coded in the configuration class. Likewise, if you had done this in XML, the values would have also been hard-coded:



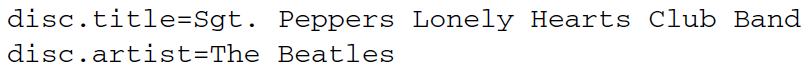
* Sometimes hard-coded values are fine. Other times, however, you may want to avoid hard-coded values and let the values be determined at runtime. For those cases Spring offers two ways of evaluating values at runtime:
* Property placeholders
* The Spring Expression Language (SpEL).

***Injecting external values***

The simplest way to resolve external values in Spring is to declare a property source and retrieve the properties via the Spring *Environment.* For example, the following listing shows a basic Spring Configuration class that uses external properties to wire up a *BlankDisc* bean.



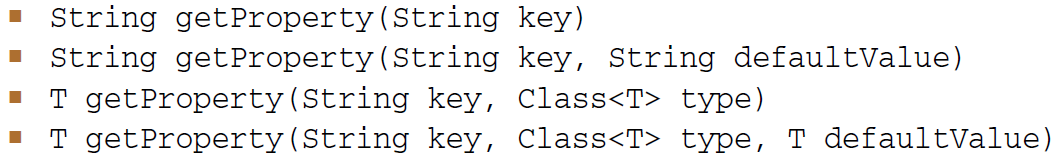
* In this example, *@PropertySource* references a file named app.properties in the class path. It might look something like this:



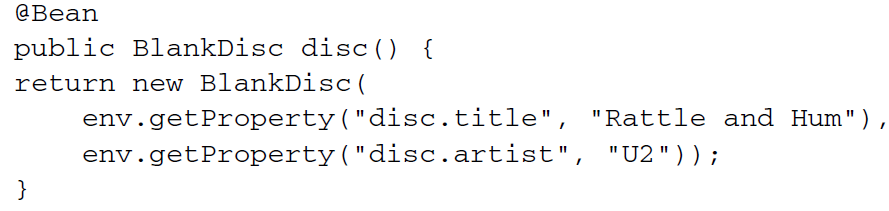
* This Properties file is loaded into Spring’s *Environment, from* which it can be retrieved later. Meanwhile, in the *disc()* method, a new *BlankDisc* is created; its constructor arguments are resolved from the properties file by calling *getProperty().*

**DIGGING INTO SPRING’S ENVIRONMENT**

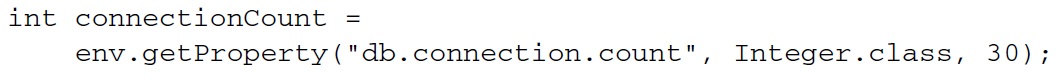
*getProperty()* is overloaded into four variations:



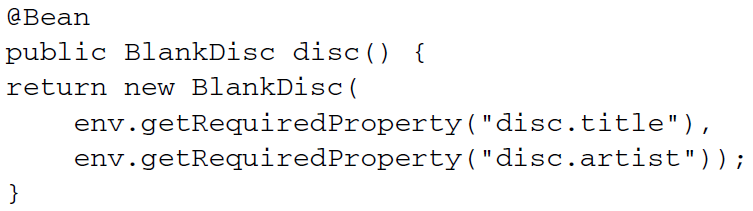
* ther first two forms of *getProperty()* always return a String value. You saw how to use the first form. But you can tweak the *@Bean* method slightly to work with default values if the specified properties don’t exist.



* The second two forms of *getProperty()* work much like the first two, but they recognize that not all values may be *Strings.* For example, suppose you’re retrieving a value representing the number of connections to maintain is a connection pool. If you receive a *String* value from the properties file, then you’ll need to convert it to an *Integer* before you can use it. But using one of the overloaded *getProperty()* methods handles that conversion for you:



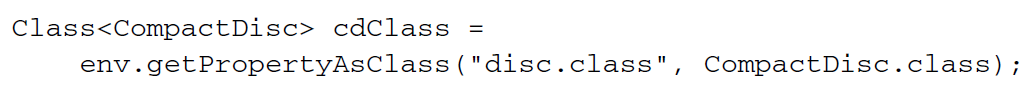
* A few more property-related methods are offered by *Environment.* If you use either of the *getProperty()* methods without specifying a default value, you’ll receive *null* if the property isn’t defined. If you want to require that the property be defined, you can use *getRequiredProperty()* like this:



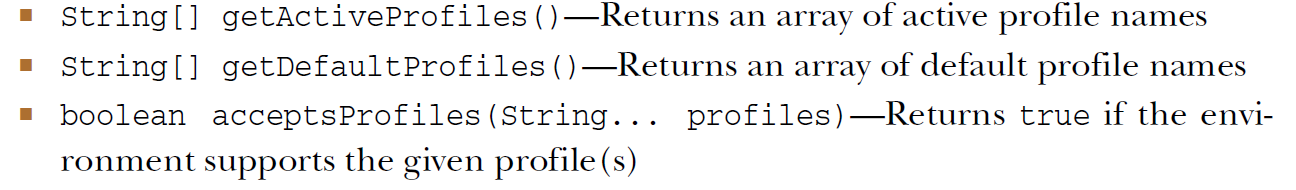
* Here, if either the *disc.title* property or the *disc.artist* property is undefined, an *IllegalStateException* will be throws.
* If you want to check for the existence of a property, you can call *containsPorperty()* on *Environment:*

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

* Finally, if you need to resolve a property into a Class, you can use the *getPropertyAsClass()* method:



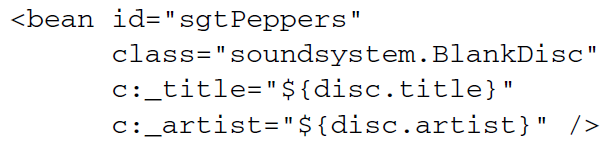
* Digressing a bit from the subject of properties, *Environment* also offers some methods for checking which profiles are active:



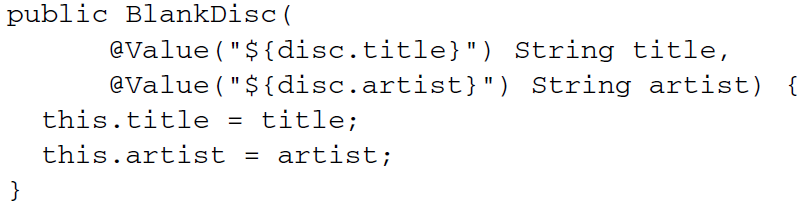
* We know how to use the *acceptsProfiles()* method. *Environment* was retrieved from *ConditionContext,* and the *acceptsProfiles()* method was used to ensure that a given bean’s profile was in play before allowing the bean to be created.
* Retrieving properties directly from *Environment* is handy, especially when you’re wiring beans in Java configuration. But Spring also offers the option of wiring properties with placeholder values that are resolved from a property source.

**RESOLVING PROPERTY PLACEHOLDERS**

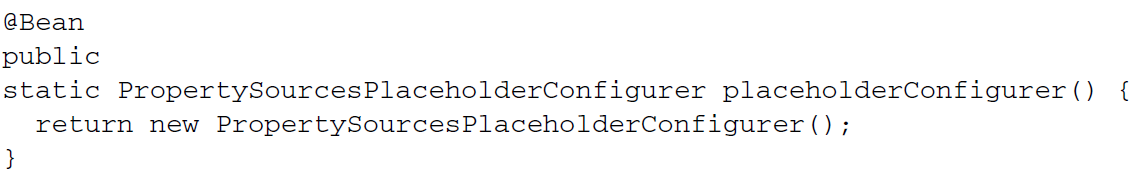
* Spring has always supported the option of externalizing properties into a properties file and then plugging them into Spring beans using placeholder values. In Spring wiring, placeholder values are property names wrapped with *${ … }.* As an wxample, you can resolve the constructor arguments for a *BlankDisc* in XML like this:

****

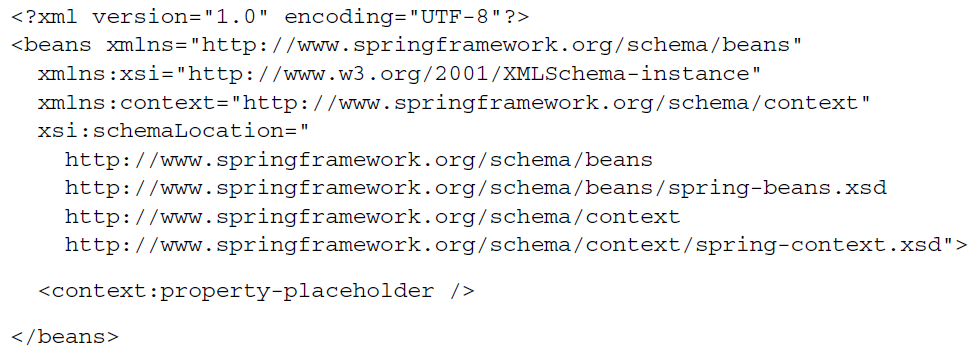
* As shown here, the *title* constructor argument is given a value that’s resolved from the property whose name is *disc.title.* And the *artist* argument is wired with the value of the property whose name is *disc.artist.* In this way, the XML configuration doesn’t use any hard-coded values . Instead, the values are resolved from a source external to configuration file.
* When relying on component-scanning and autowiring to create and initialize your application components, there’s no configuration file or class where you can specify the placeholders. Instead, you can use the *@Value* annotation in much the same way as you might use the *@Autowired* annotation. In the BlankDisc class, for example, the constructor might be written like this:

****

* In order to use placeholder values, you must configure either a *PropertyPlaceholder*Cofigurer bean or a *PropertySourcesPlaceholderConfigurer* bean. Starting with Spring 3.1, *PropertySourcesPlaceholderConfigurer* is preffered because it resolves placeholders against the Spring *Environment* and its set of property sources.
* The following *@Bean* method configures PropertySourcesPlaceholderConfigurer in java configuration:

****

* If you’d rather use XML configuration, the *<context:property-placeholder>* element from Spring’s *context* namespace will give you a *PropertySourcesPlaceholderConfigurer* bean:

****

* Resolving external properties is one way to defer value resolution until runtime, but its focus is finely tuned on resolving properties, by name, from Spring’s *Environment* and property sources.