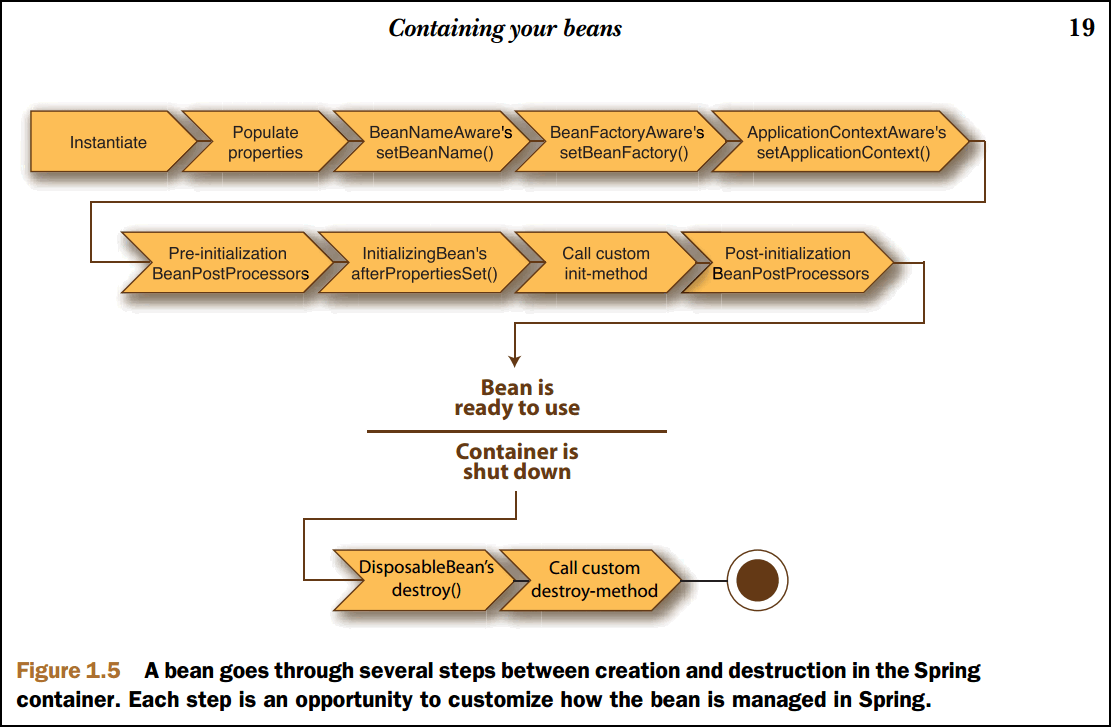
# Chapter1

The difference between using FileSystemXmlApplicationContextand ClassPathXmlApplicationContextis that FileSystemXmlApplicationContextwill look for

foo.xml in a specific location within the file system, whereas ClassPathXmlApplicationContextwill look for foo.xml anywherein the classpath (including JAR

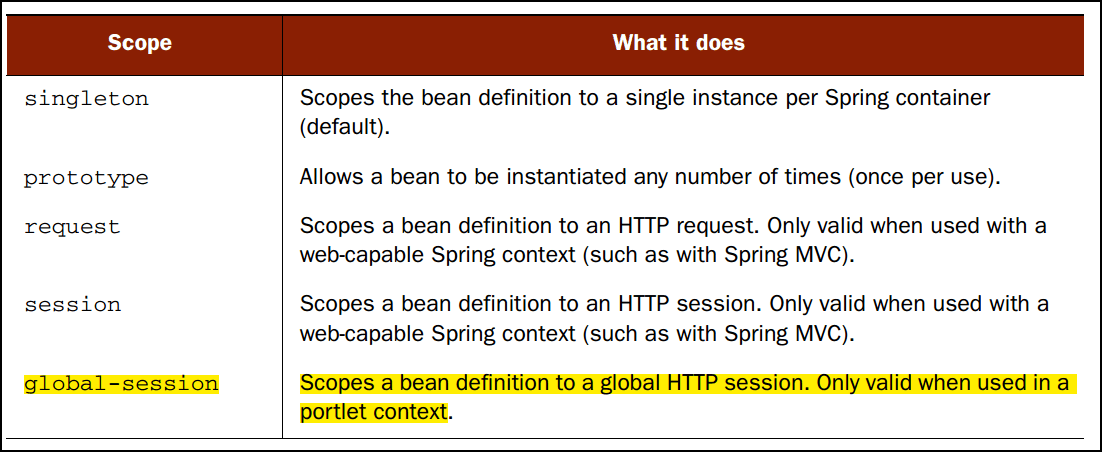
files).



As you can see, a spring application context bean factory performs several setup steps before a bean is ready to use. Breaking down figure 1.5 in more detail:

1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean’s properties
3. If the bean implements BeanNameAware, Spring passes the bean’s IDto the setBeanName()method
4. If the bean implements BeanFactoryAware, Spring calls the setBeanFactory() method, passing in the bean factory itself.
5. If the bean implements ApplicationContextAware, Spring will call the setApplicationContext()method, passing in a reference to the enclosing application context.
6. If any of the beans implement the BeanPostProcessorinterface, Spring calls their postProcessBeforeInitialization()method.
7. If any beans implement the InitializingBeaninterface, Spring calls their afterPropertiesSet()method. Similarly, if the bean was declared with an init-method, then the specified initialization method will be called.
8. If there are any beans that implement BeanPostProcessor, Spring will call their postProcessAfterInitialization()method.
9. At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed.
10. If any beans implement the DisposableBeaninterface, then Spring will call their destroy()methods. Likewise, if any bean was declared with a destroymethod, then the specified method will be called.

# Chapter 2

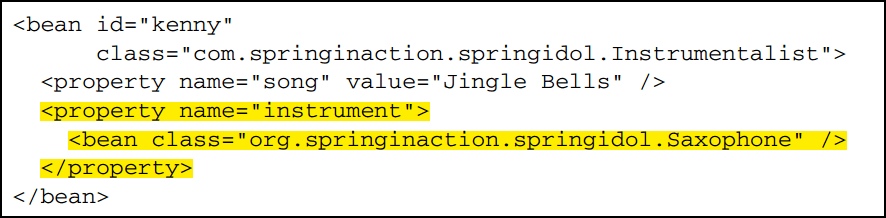
Table 2.2 Spring’s bean scopes let you declare the scope under which beans are created without hardcoding the scoping rules in the bean class itself.

InitializingBeanand DisposableBean :

An optional way to define init and destroy methods for a bean is to write the bean class to implement Spring’s InitializingBeanand DisposableBeaninterfaces. The Spring container treats beans that implement these interfaces in a special way by allowing them to hook into the bean lifecycle. InitializingBeandeclares an afterPropertiesSet()method that serves as the init method. As for DisposableBean, it declares a destroy() method that gets called when a bean is removed from the application context.

The chief benefit of using these lifecycle interfaces is that the Spring container can automatically detect beans that implement them without any external configuration. The disadvantage of implementing these interfaces is that you couple your application’s beans to Spring’s API. For this reason alone, I recommend that you rely on the init-methodand destroy-methodattributes to initialize and destroy your beans.

The only scenario where you might favor Spring’s lifecycle interfaces is if you’re developing a framework bean that’s to be used specifically within Spring’s container

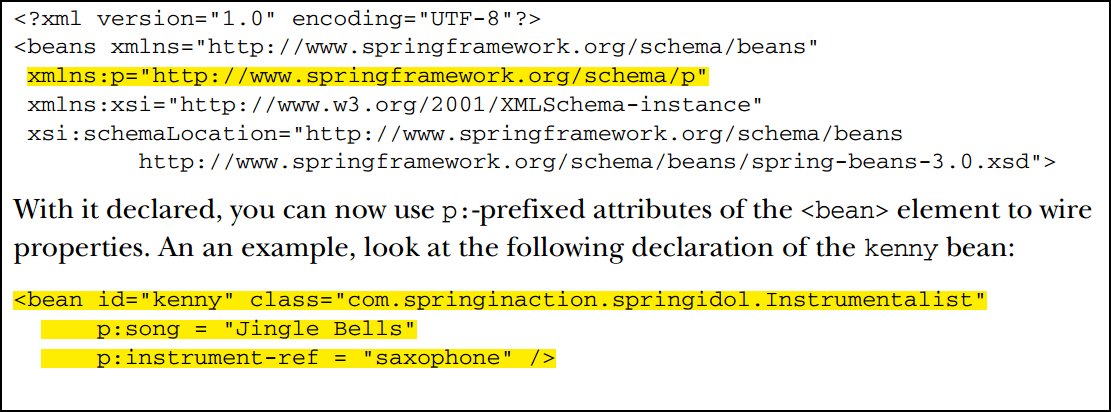


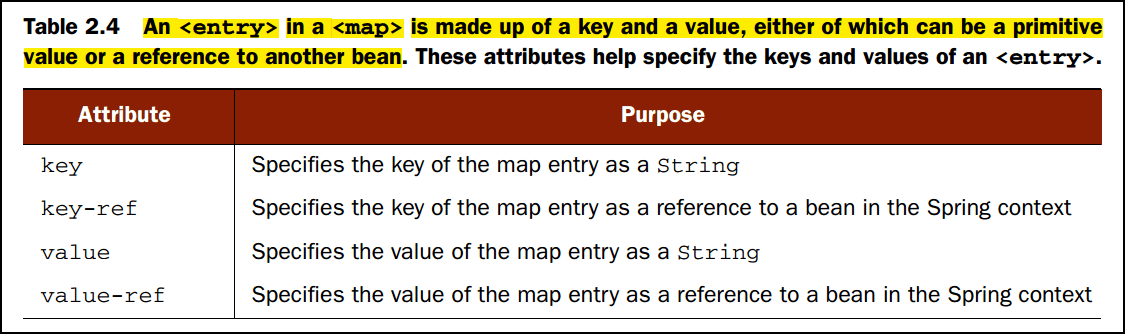
As you can see, an inner bean is defined by declaring a <bean>element directly as a child of the <property>element to which it’ll be injected. In this case, a Saxophone will be created and wired into Kenny’s instrument property

Note that the inner beans don’t have an idattribute set. Though it’s perfectly legal to declare an IDfor an inner bean, it’s not necessary because you’ll never refer to the inner bean by name. This highlights the main drawback of using inner beans: they can’t be reused. Inner beans are only usefulfor injection once and can’t be referred to by other beans.

You may also find that using inner-bean definitions has a negative impact on the readability of the XML in the Spring context files.

Wiring properties with Spring’s p namespace





The Propertiesclass serves roughly the same purpose as Map, but limits the keys and values to Strings.

To set a property to null, you simply use the <null/>element. For example:

<property name="someNonNullProperty"><null/></property>

Another reason for explicitly wiring nullinto a property is to override an autowired property value

**Wiring with expressions （Since Spring 3)**

Spring 3 introduced the Spring ExpressionLanguage (SpEL), a powerful yet succinct way of wiring values into a bean’s properties or constructor arguments using expressions that are evaluated at runtime.

SpELhas a lot of tricks up its sleeves, including

1. The ability to reference beans by their ID
2. Invoking methods and accessing properties on objects
3. Mathematical, relational, and logical operations on values
4. Regular expression matching
5. Collection manipulation

<property name="count" value="#{5}"/>

The #{}markers are a clue to Spring that the content that they contain is a SpEL expression. They could be mixed with non-SpELvalues as well:

<property name="message" value="The value is #{5}"/>

Float

<property name="frequency" value="#{89.7}"/>

String

<property name="name" value="#{‘Chuck’}"/>

or

<property name=‘name’ value=‘#{"Chuck"}’/>

Boolean

<property name="enabled" value="#{false}"/>

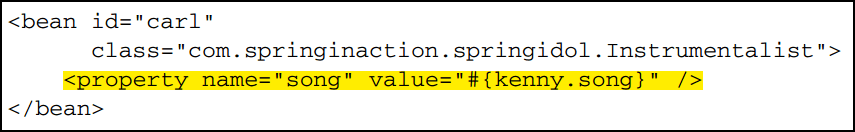
For

<property name="instrument" value="#{saxophone}"/>

and

<property name="instrument" ref="saxophone"/>

the outcome is the same



The first part (the part before the period delimiter) refers to the kennybean by its ID. The second part refers to the songattribute of the kennybean. By wiring the carl bean’s songproperty this way, it’s effectively asif you programmatically performed the following Java code:

Instrumentalist carl = new Instrumentalist();

carl.setSong(kenny.getSong());

C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\4050aaedf58a4dc8b697c4d9c4832947\clipboard.png

C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\1e173c8df2bc463fb49c9d7c4cfb6ac5\clipboard.png

The way to avoid the dreaded NullPointerExceptionin SpELis to use the nullsafe accessorC:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\d9c8818f9b0f4092b1e7322dbb2c1996\clipboard.png

Instead of a lonely dot (.) to access the toUpperCase()method, now you’re using **?.**operator. This operator makes sure that the item to its left isn’t null before accessing

the thing to its right. So, if selectSong()were to return a null, then SpELwouldn’t even try to invoke toUpperCase()on it.

T(java.lang.Math)

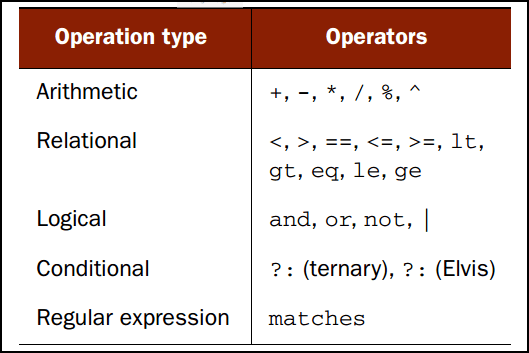
The result of the T()operator, as shown here, is a Classobject that represents java.lang.Math. You could even wire it into a bean property of type Classif you want to. But the real value of the T()operator is that it gives usaccess to static methods and constants on a given class.

C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\8085a229ba5d4cd8b371952a9c772825\clipboard.png

C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\d56486b85ef548a5b23758b527a5e9d6\clipboard.png

**Performing operations on SpEL values**

Table 2.5 SpEL includes several operators that you can use to manipulate the values of  an expression.



SpELsupports all of the basic arithmetic operators that Java supports, plus the carat (^) operator for performing a power of operation.

<property name="adjustedAmount" value="#{counter.total - 20}"/>

<property name="circumference" value="#{2 \* T(java.lang.Math).PI \* circle.radius}"/>

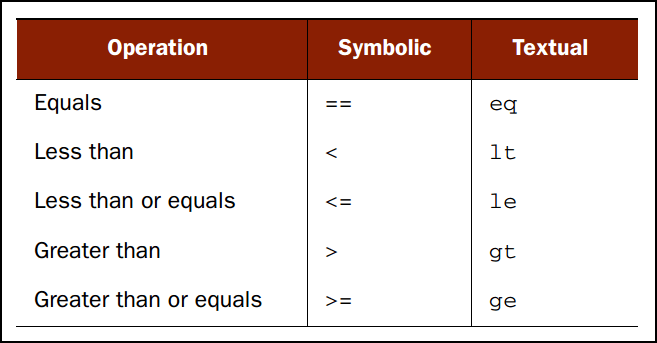
Even though we’re talking about SpEL’s arithmetic operators,it’s worth mentioning that the +operator is overloaded toperform concatenation on String values. For example:

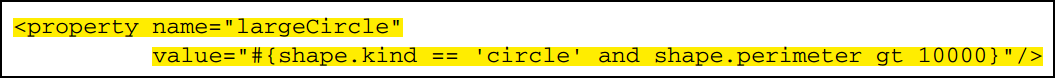
<property name="fullName" value="#{performer.firstName+’’+performer.lastName}"/>

<property name="equal" value="#{counter.total **==**100}"/>

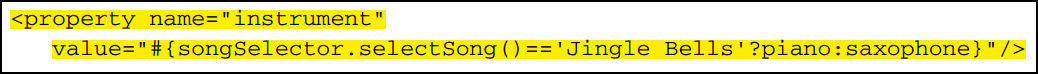
Unfortunately, the less-than and greater-than symbols pose a problem when using these expressions in Spring’s XMLconfiguration, as they have special meaning in XML. So, when using SpELin XML, it’s best to use SpEL’s textual alternatives to these operators. For example:

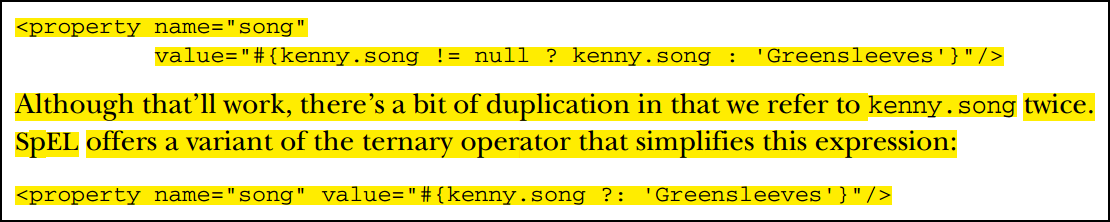
<property name="hasCapacity" value="#{counter.total le 100000}"/>



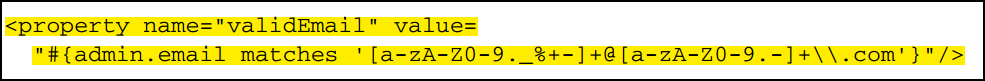


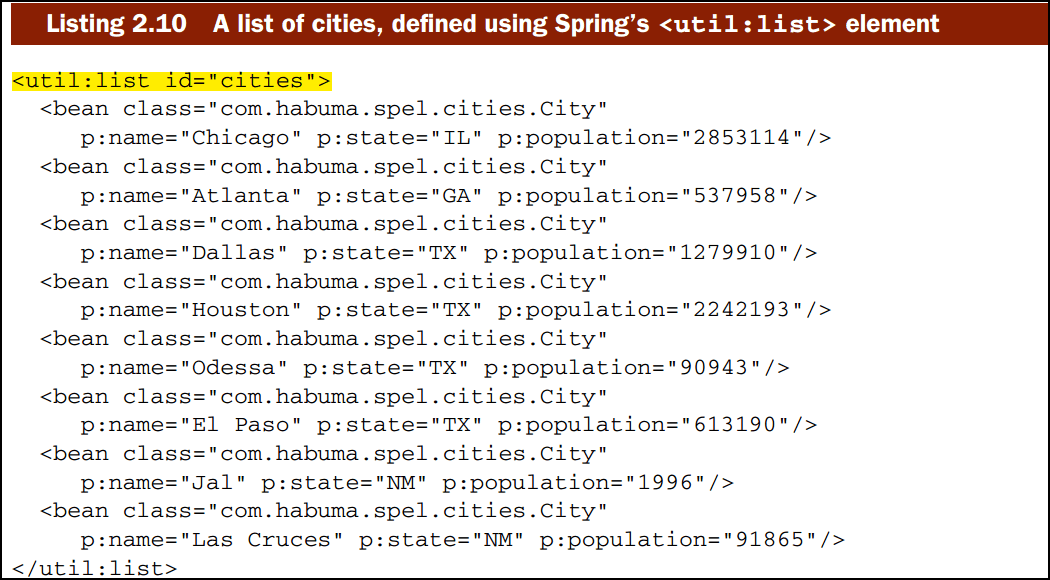
you can use SpEL’s ternary (?:) operator:





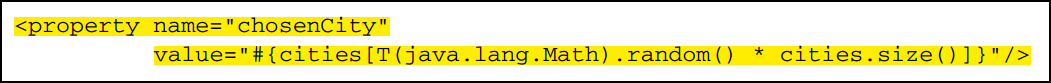
The matchesoperator attempts to apply a regular expression (given as its rightside argument) against a Stringvalue (given as the left-side argument).



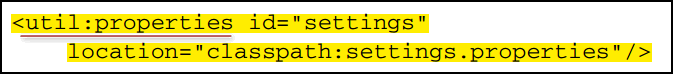


The <util:list>element comes from Spring’s utilnamespace. It effectively creates a bean of type java.util.Listthat contains all of the values or beans that it contains. In this case, that’s a list of eight Citybeans

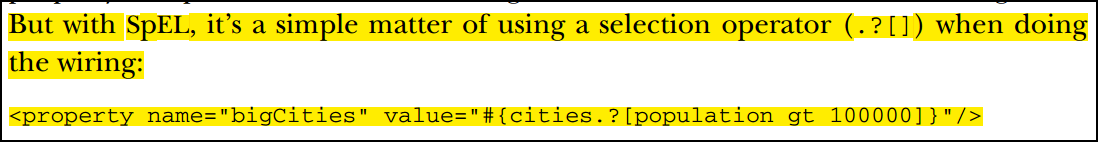
C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\7507efa6c10140e39cd878f49ef16f5f\clipboard.png



C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\fae3f17a256e4ca39fb278b56bfe9a91\clipboard.png



C:\Users\Administrator\AppData\Local\YNote\Data\fushcpc@gmail.com\67396b861b45400f805f7259f8f49510\clipboard.png



SpELalso offers two other selection operators, .^[]and .$[], for selecting the first and last matching items (respectively) froma collection.

For example, to select the first big city from cities:

<property name="aBigCity" value="#{cities.^[population gt 100000]}"/>

No ordering is done on the collection prior to selection, so the Cityrepresenting Chicago would be wired into the aBigCityproperty

Likewise, the Cityobject representing El Paso could be selected as follows:

<property name="aBigCity" value="#{cities.$[population gt 100000]}"/>

# Chapter 10 Working with remote services

# 

What’s more, if the call to the remote service results in a java.rmi.RemoteException,the proxy handles that exception and rethrows it as an unchecked RemoteAccessException. Remote exceptions usually signal problems such as network or configuration issues that can’t be gracefully recovered from. Since a client can usually do little to gracefully recover from a remote exception, rethrowing a RemoteAccessException makes it optional for the client to handle the exception