covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

2 Wiring beans

This chapter covers

- Declaring beans
- Injecting constructors and setters
- Wiring beans
- Controlling bean creation and destruction

Have you ever stuck around after a movie long enough to watch the credits? It's incredible how many different people it takes to pull together a major motion picture. In addition to the obvious participants—the actors, scriptwriters, directors, and producers—there are the not-so-obvious—the musicians, special effects crew, and art directors. And that's not to mention the key grip, sound mixer, costumers, makeup artists, stunt coordinators, publicists, first assistant to the cameraperson, second assistant to the cameraperson, set designers, gaffer, and (perhaps most importantly) the caterers.

Now imagine what your favorite movie would've been like had none of these people talked to one another. Let's say that they all showed up at the studio and started doing their own thing without any coordination of any kind. If the director keeps to himself and doesn't say "roll 'em," then the cameraperson won't start shooting. It probably wouldn't matter anyway, because the lead actress would still

be in her trailer and the lighting wouldn't work because the gaffer wouldn't have been hired. Maybe you've seen a movie where it looks like this is what happened. But most movies (the good ones anyway) are the product of thousands of people working together toward the common goal of making a blockbuster movie.

In this respect, a great piece of software isn't much different. Any nontrivial application is made up of several objects that must work together to meet some business goal. These objects must be aware of one another and communicate with one another to get their jobs done. In an online shopping application, for instance, an order manager component may need to work with a product manager component and a credit card authorization component. All of these will likely need to work with a data access component to read from and write to a database.

But as we saw in chapter 1, the traditional approach to creating associations between application objects (via construction or lookup) leads to complicated code that's difficult to reuse and unit test. At best, these objects do more work than they should. At worst, they're highly coupled to one another, making them hard to reuse and hard to test.

In Spring, objects aren't responsible for finding or creating the other objects that they need to do their jobs. Instead, they're given references to the objects that they collaborate with by the container. An order manager component, for example, may need a credit card authorizer—but it doesn't need to create the credit card authorizer. It just needs to show up empty-handed and it'll be given a credit card authorizer to work with.

The act of creating these associations between application objects is the essence of dependency injection (DI) and is commonly referred to as *wiring*. In this chapter we'll explore the basics of bean wiring using Spring. As DI is the most elemental thing Spring does, these are techniques you'll use almost every time you develop Spring-based applications.

2.1 Declaring beans

At this point, I'd like to welcome you to the first (and likely the last) annual JavaBean talent competition. I've searched the nation (actually, just our IDE's workspace) for the most talented JavaBeans to perform and in the next few chapters, we'll set up the competition and our judges will weigh in. Spring programmers, this is your *Spring Idol*.

In our competition, we're going to need some performers, which are defined by the `Performer` interface:

```
package com.springinaction.springidol;

public interface Performer {
  void perform() throws PerformanceException;
}
```

In the *Spring Idol* talent competition, you'll meet several contestants, all of which implement the `Performer` interface. To get started, let's set the stage for the competition by looking at the essentials of a Spring configuration.

2.1.1 Setting up Spring configuration

As has been said already, Spring is a container-based framework. But if you don't configure Spring, then it's an empty container and doesn't serve much purpose. We need to configure Spring to tell it what beans it should contain and how to wire those beans so that they can work together.

As of Spring 3.0, there are two ways to configure beans in the Spring container. Traditionally, Spring configuration is defined in one or more XML files. But Spring 3.0 also offers a Java-based configuration option. We'll focus on the traditional XML option for now, but we'll look at Spring's Java-based configuration later in section 3.4.

When declaring beans in XML, the root element of the Spring configuration file is the `<beans>` element from Spring's beans schema. A typical Spring configuration XML file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Bean declarations go here -->

</beans>
```

Within the `<beans>` you can place all of your Spring configuration, including `<bean>` declarations. But the `beans` namespace isn't the only Spring namespace you'll encounter. All together, the core Spring Framework comes with ten configuration namespaces, as described in table 2.1.

Table 2.1 Spring comes with several XML namespaces through which you can configure the Spring container

Namespace	Purpose
`aop`	Provides elements for declaring aspects and for automatically proxying `@AspectJ`-annotated classes as Spring aspects.
`beans`	The core primitive Spring namespace, enabling declaration of beans and how they should be wired.
`context`	Comes with elements for configuring the Spring application context, including the ability to autodetect and autowire beans and injection of objects not directly managed by Spring.
`jee`	Offers integration with Java EE APIs such as JNDI and EJB.
`jms`	Provides configuration elements for declaring message-driven POJOs.
`lang`	Enables declaration of beans that are implemented as Groovy, JRuby, or BeanShell scripts.
`mvc`	Enables Spring MVC capabilities such as annotation-oriented controllers, view controllers, and interceptors.

Table 2.1 Spring comes with several XML namespaces through which you can configure the Spring container *(continued)*

Namespace	Purpose
`oxm`	Supports configuration of Spring's object-to-XML mapping facilities.
`tx`	Provides for declarative transaction configuration.
`util`	A miscellaneous selection of utility elements. Includes the ability to declare collections as beans and support for property placeholder elements.

In addition to the namespaces that come with the Spring Framework, many of the members of the Spring portfolio, such as Spring Security, Spring Web Flow, and Spring Dynamic Modules, also provide their own Spring configuration namespace.

We'll see more of Spring's namespaces as this book progresses. But for now, let's fill in that conspicuously empty space in the middle of the XML configuration by adding some `<bean>` elements within `<beans>`.

2.1.2 Declaring a simple bean

Unlike some similarly named talent competitions that you may have heard of, *Spring Idol* doesn't cater to only singers. Many of the performers can't carry a tune at all. For example, one of the performers is a `Juggler`.

Listing 2.1 A juggling bean

```
package com.springinaction.springidol;

public class Juggler implements Performer {
  private int beanBags = 3;

  public Juggler() {
  }

  public Juggler(int beanBags) {
    this.beanBags = beanBags;
  }

  public void perform() throws PerformanceException {
    System.out.println("JUGGLING " + beanBags + " BEANBAGS");
  }
}
```

As you can see, this `Juggler` class does little more than implement the `Performer` interface to report that it's juggling some beanbags. By default, the `Juggler` juggles three beanbags, but can be given some other number of beanbags through its constructor.

With the `Juggler` class defined, please welcome our first performer, Duke, to the stage. Duke is defined as a Spring bean. Here's how Duke is declared in the Spring configuration file (spring-idol.xml):

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler" />
```

The `<bean>` element is the most basic configuration unit in Spring. It tells Spring to create an object for you. Here you've declared Duke as a Spring-managed bean using what's nearly the simplest `<bean>` declaration possible. The `id` attribute gives the bean a name by which it'll be referred to in the Spring container. This bean will be known as `duke`. And, as you can see from the `class` attribute, Duke is a `Juggler`.

When the Spring container loads its beans, it'll instantiate the `duke` bean using the default constructor. In essence, `duke` will be created using the following Java code:[1]

```
new com.springinaction.springidol.Juggler();
```

To give Duke a try, you can load the Spring application context using the following code:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "com/springinaction/springidol/spring-idol.xml");

Performer performer = (Performer) ctx.getBean("duke");
performer.perform();
```

Although this isn't the real competition, the previous code gives Duke a chance to practice. When run, this code prints the following:

```
JUGGLING 3 BEANBAGS
```

By default, Duke juggles only three beanbags at once. But juggling three beanbags isn't all that impressive—anybody can do that. If Duke is to have any hope of winning the talent competition, he's going to need to juggle many more beanbags at once. Let's see how to configure Duke to be a champion juggler.

2.1.3 *Injecting through constructors*

To really impress the judges, Duke has decided to break the world record by juggling as many as 15 beanbags at once.[2]

Recall from listing 2.1 that the `Juggler` class can be constructed in two different ways:

- Using the default constructor
- Using a constructor that takes an `int` argument which indicates the number of beanbags that the `Juggler` will attempt to keep in the air

Although the declaration of the `duke` bean in section 2.1.2 is valid, it uses the `Juggler`'s default constructor, which limits Duke to juggling only three beanbags at once. To make Duke a world-record juggler, we'll need to use the other constructor. The following XML redeclares Duke as a 15-beanbag juggler:

---

[1] Emphasis on "in essence." Actually, Spring creates its beans using reflection.

[2] Juggling trivia: Who holds the actual world record for juggling beanbags depends on how many beanbags are juggled and for how long. Bruce Sarafian holds several records, including juggling 12 beanbags for 12 catches. Another record-holding juggler is Anthony Gatto, who juggled 7 balls for 10 minutes and 12 seconds in 2005. Another juggler, Peter Bone, claims to have juggled as many as 13 beanbags for 13 catches—but there's no video evidence of the feat.

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler">
    <constructor-arg value="15" />
</bean>
```

The `<constructor-arg>` element is used to give Spring additional information to use when constructing a bean. If no `<constructor-arg>`s are given, as in section 2.1.2, the default constructor is used. But here you've given a `<constructor-arg>` with a value attribute set to 15, so the `Juggler`'s other constructor will be used instead.

Now when Duke performs, the following is printed:

```
JUGGLING 15 BEANBAGS
```

Juggling 15 beanbags at once is mighty impressive. But there's something we didn't tell you about Duke. Not only is Duke a good juggler, but he's also skilled at reciting poetry. Juggling while reciting poetry takes a lot of mental discipline. If Duke can juggle while reciting a Shakespearean sonnet then he should be able to establish himself as the clear winner of the competition. (I told you this wouldn't be like those other talent shows!)

INJECTING OBJECT REFERENCES WITH CONSTRUCTORS

Because Duke is more than just an average juggler—he's a poetic juggler—we need to define a new type of juggler for him to be. `PoeticJuggler` is a class more descriptive of Duke's talent.

Listing 2.2 A juggler who waxes poetic

```
package com.springinaction.springidol;

public class PoeticJuggler extends Juggler {
  private Poem poem;

  public PoeticJuggler(Poem poem) {                    <-- Inject poem
    super();
    this.poem = poem;
  }

  public PoeticJuggler(int beanBags, Poem poem) {      <-- Inject beanbag count and poem
    super(beanBags);
    this.poem = poem;
  }

  public void perform() throws PerformanceException {
    super.perform();
    System.out.println("While reciting...");
    poem.recite();
  }
}
```

This new type of juggler does everything a regular juggler does, but it also has a reference to a poem to be recited. Speaking of the poem, here's an interface that generically defines what a poem looks like:

```
package com.springinaction.springidol;

public interface Poem {
  void recite();
}
```

One of Duke's favorite Shakespearean sonnets is "When in disgrace with fortune and men's eyes." `Sonnet29` is an implementation of the `Poem` interface that defines this sonnet.

Listing 2.3 A class that represents a great work of the Bard

```
package com.springinaction.springidol;

public class Sonnet29 implements Poem {
  private static String[] LINES = {
       "When, in disgrace with fortune and men's eyes,",
       "I all alone beweep my outcast state",
       "And trouble deaf heaven with my bootless cries",
       "And look upon myself and curse my fate,",
       "Wishing me like to one more rich in hope,",
       "Featured like him, like him with friends possess'd,",
       "Desiring this man's art and that man's scope,",
       "With what I most enjoy contented least;",
       "Yet in these thoughts myself almost despising,",
       "Haply I think on thee, and then my state,",
       "Like to the lark at break of day arising",
       "From sullen earth, sings hymns at heaven's gate;",
       "For thy sweet love remember'd such wealth brings",
       "That then I scorn to change my state with kings." };

  public Sonnet29() {
  }

  public void recite() {
    for (int i = 0; i < LINES.length; i++) {
      System.out.println(LINES[i]);
    }
  }
}
```

`Sonnet29` can be declared as a Spring `<bean>` with the following XML:

```
<bean id="sonnet29"
      class="com.springinaction.springidol.Sonnet29" />
```

With the poem chosen, all you need to do is give it to Duke. Now that Duke is a `PoeticJuggler`, his `<bean>` declaration will need to change slightly:

```
<bean id="poeticDuke"
      class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg ref="sonnet29" />
</bean>
```

As you can see from listing 2.2, there's no default constructor. The only way to construct a `PoeticJuggler` is to use a constructor that takes arguments. In this listing,

you're using the constructor that takes an `int` and a `Poem` as arguments. The `duke` bean declaration configures the number of beanbags as 15 through the `int` argument using `<constructor-arg>`'s `value` attribute.

But you can't use `value` to set the second constructor argument because a `Poem` isn't a simple type. Instead, the `ref` attribute is used to indicate that the value passed to the constructor should be a reference to the bean whose ID is `sonnet29`. Although the Spring container does much more than just construct beans, you may imagine that when Spring encounters the `sonnet29` and `duke <bean>`s, it performs some logic that's essentially the same as the following lines of Java:

```
Poem sonnet29 = new Sonnet29();
Performer duke = new PoeticJuggler(15, sonnet29);
```

Now when Duke performs, he not only juggles but will also recite Shakespeare, resulting in the following being printed to the standard output stream:

```
JUGGLING 15 BEANBAGS WHILE RECITING... When, in
disgrace with fortune and men's eyes, I all alone beweep my outcast
state And trouble deaf heaven with my bootless cries And look upon
myself and curse my fate, Wishing me like to one more rich in hope,
Featured like him, like him with friends possess'd, Desiring this
man's art and that man's scope, With what I most enjoy contented
least; Yet in these thoughts myself almost despising, Haply I think
on thee, and then my state, Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate; For thy sweet love
remember'd such wealth brings That then I scorn to change my state
with kings.
```

Creating beans through constructor injection is great, but what if the bean you want to declare doesn't have a public constructor? Let's see how to wire in beans that are created through factory methods.

CREATING BEANS THROUGH FACTORY METHODS

Sometimes the only way to instantiate an object is through a static factory method. Spring is ready-made to wire factory-created beans through the `<bean>` element's `factory-method` attribute.

To illustrate, consider the case of configuring a singleton[3] class as a bean in Spring. Singleton classes generally ensure that only one instance is created by only allowing creation through a static factory method. The `Stage` class in the following listing is a basic example of a singleton class.

Listing 2.4 The `Stage` singleton class

```
package com.springinaction.springidol;

public class Stage {
  private Stage() {
  }
```

[3] I'm talking about the Gang of Four Singleton pattern here, not the Spring notion of singleton bean definitions.

```
  private static class StageSingletonHolder {
    static Stage instance = new Stage();
  }

  public static Stage getInstance() {
    return StageSingletonHolder.instance;
  }
}
```

Lazily loads instance

Return instance

In the *Spring Idol* competition, we want to ensure that there's only one stage for the performers to show their stuff. `Stage` has been implemented as a singleton to ensure that there's no way to create more than one instance of `Stage`.

But note that `Stage` doesn't have a public constructor. Instead, the static `getInstance()` method returns the same instance every time it's called. (For thread safety, `getInstance()` employs a technique known as "initialization on demand holder" to create the singleton instance.[4]) How can we configure `Stage` as a bean in Spring without a public constructor?

Fortunately, the `<bean>` element has a `factory-method` attribute that lets you specify a static method to be invoked instead of the constructor to create an instance of a class. To configure `Stage` as a bean in the Spring context, you simply use `factory-method` as follows:

```
<bean id="theStage"
      class="com.springinaction.springidol.Stage"
      factory-method="getInstance" />
```

Here I've shown you how to use `factory-method` to configure a singleton as a bean in Spring, but it's perfectly suitable for any occasion where you need to wire an object produced by a static method. You'll see more of `factory-method` in chapter 4 when we use it to get references to AspectJ aspects so that they can be injected with dependencies.

2.1.4 Bean scoping

By default, all Spring beans are singletons. When the container dispenses a bean (either through wiring or as the result of a call to the container's `getBean()` method) it'll always hand out the exact same instance of the bean. But there may be times when you need a unique instance of a bean each time it's asked for. How can you override Spring's default singleton nature?

When declaring a `<bean>` in Spring, you have the option of declaring a scope for that bean. To force Spring to produce a new bean instance each time one is needed, you should declare the bean's `scope` attribute to be `prototype`. For example, suppose that tickets for a performance are declared as a bean in Spring:

```
<bean id="ticket"
      class="com.springinaction.springidol.Ticket" scope="prototype" />
```

[4] For information on the "initialization on demand holder" idiom, see http://mng.bz/IGYx.

It's important that everyone attending the performance be given a distinct ticket. If the `ticket` bean were a singleton, everyone would receive the same ticket. This would work out fine for the first person to arrive, but everyone else would be accused of ticket counterfeiting!

By setting the `scope` attribute to `prototype`, we can be assured that a distinct instance will be given out to everyone who the `ticket` bean is wired into.

In addition to `prototype`, Spring offers a handful of other scoping options out of the box, as listed in table 2.2.

Table 2.2 Spring's bean scopes let you declare the scope under which beans are created without hard-coding the scoping rules in the bean class itself.

Scope	What it does
`singleton`	Scopes the bean definition to a single instance per Spring container (default).
`prototype`	Allows a bean to be instantiated any number of times (once per use).
`request`	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
`session`	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
`global-session`	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

For the most part, you'll probably want to leave scoping to the default `singleton`, but `prototype` scope may be useful in situations where you want to use Spring as a factory for new instances of domain objects. If domain objects are configured as prototype beans, you can easily configure them in Spring, just like any other bean. But Spring is guaranteed to always dispense a unique instance each time a prototype bean is asked for.

The astute reader will recognize that Spring's notion of singletons is limited to the scope of the Spring context. Unlike true singletons, which guarantee only a single instance of a class per classloader, Spring's singleton beans only guarantee a single instance of the bean definition per the application context—nothing is stopping you from instantiating that same class in a more conventional way or even defining several `<bean>` declarations that instantiate the same class.

2.1.5 Initializing and destroying beans

When a bean is instantiated, it may be necessary to perform some initialization to get it into a usable state. Likewise, when the bean is no longer needed and is removed from the container, some cleanup may be in order. To accommodate setup and teardown of beans, Spring provides hooks into the bean lifecycle.

To define setup and teardown for a bean, simply declare the `<bean>` with `init-method` and/or `destroy-method` parameters. The `init-method` attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, `destroy-method` specifies a method that is called just before a bean is removed from the container.

To illustrate, imagine that we have a Java class called `Auditorium` which represents the performance hall where the talent competition will take place. `Auditorium` will likely do a lot of things, but for now let's focus on two things that are important at the beginning and the end of the show: turning on the lights and then turning them back off.

To support these essential activities, the `Auditorium` class might have `turnOnLights()` and `turnOffLights()` methods:

```
public class Auditorium {
    public void turnOnLights() {
        ...
    }

    public void turnOffLights() {
        ...
    }
}
```

The details of what takes place within the `turnOnLights()` and `turnOffLights()` methods isn't terribly important. What's important is that the `turnOnLights()` method be called at the start and `turnOffLights()` be called at the end. For that, let's use the `init-method` and `destroy-method` attributes when declaring the `auditorium` bean:

```
<bean id="auditorium"
      class="com.springinaction.springidol.Auditorium"
      init-method="turnOnLights"
     destroy-method="turnOffLights"/>
```

When declared this way, the `turnOnLights()` method will be called soon after the `auditorium` bean is instantiated, allowing it the opportunity to light up the performance venue. And, just before the bean is removed from the container and discarded, `turnOffLights()` will be called to turn the lights off.

`InitializingBean` and `DisposableBean`

An optional way to define init and destroy methods for a bean is to write the bean class to implement Spring's `InitializingBean` and `DisposableBean` interfaces. The Spring container treats beans that implement these interfaces in a special way by allowing them to hook into the bean lifecycle. `InitializingBean` declares 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.

(continued)
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-method` and `destroy-method` attributes 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.

DEFAULTING INIT-METHOD AND DESTROY-METHOD

If many of the beans in a context definition file will have initialization or destroy methods with the same name, you don't have to declare `init-method` or `destroy-method` on each individual bean. Instead you can take advantage of the `default-init-method` and `default-destroy-method` attributes on the `<beans>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
   default-init-method="turnOnLights"
   default-destroy-method="turnOffLights"> ...
</beans>
```

The `default-init-method` attribute sets an initialization method across all beans in a given context definition. Likewise, `default-destroy-method` sets a common destroy method for all beans in the context definition. In this case, we're asking Spring to initialize all beans in the context definition file by calling `turnOnLights()` and to tear them down with `turnOffLights()` (if those methods exist—otherwise nothing happens).

2.2 Injecting into bean properties

Typically, a JavaBean's properties are private and will have a pair of accessor methods in the form of `setXXX()` and `getXXX()`. Spring can take advantage of a property's setter method to configure the property's value through setter injection.

To demonstrate Spring's other form of DI, let's welcome our next performer to the stage. Kenny is a talented instrumentalist, as defined by the `Instrumentalist` class.

Listing 2.5 Defining a performer who is talented with musical instruments

```
package com.springinaction.springidol;

public class Instrumentalist implements Performer {
  public Instrumentalist() {
  }
```

```
  public void perform() throws PerformanceException {
    System.out.print("Playing " + song + " : ");
    instrument.play();
  }

  private String song;

  public void setSong(String song) {
    this.song = song;
  }

  public String getSong() {
    return song;
  }

  public String screamSong() {
    return song;
  }

  private Instrument instrument;

  public void setInstrument(Instrument instrument) {
    this.instrument = instrument;
  }
}
```

Inject song

Inject instrument

From listing 2.5, we can see that an `Instrumentalist` has two properties: `song` and `instrument`. The `song` property holds the name of the song that the instrumentalist will play and is used in the `perform()` method. The `instrument` property holds a reference to an `Instrument` that the instrumentalist will play. An `Instrument` is defined by the following interface:

```
package com.springinaction.springidol;

public interface Instrument {
  public void play();
}
```

Because the `Instrumentalist` class has a default constructor, Kenny could be declared as a `<bean>` in Spring with the following XML:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist" />
```

Although Spring will have no problem instantiating `kenny` as an `Instrumentalist`, Kenny will have a hard time performing without a `song` or an `instrument`. Let's look at how to give Kenny his `song` and `instrument` by using setter injection.

2.2.1 *Injecting simple values*

Bean properties can be configured in Spring using the `<property>` element. `<property>` is similar to `<constructor-arg>` in many ways, except that instead of injecting values through a constructor argument, `<property>` injects by calling a property's setter method.

To illustrate, let's give Kenny a song to perform using setter injection. The following XML shows an updated declaration of the `kenny` bean:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
</bean>
```

Once the `Instrumentalist` has been instantiated, Spring will use property setter methods to inject values into the properties specified by `<property>` elements. The `<property>` element in this XML instructs Spring to call `setSong()` to set a value of `"Jingle Bells"` for the `song` property.

In this case, the `value` attribute of the `<property>` element is used to inject a `String` value into a property. But `<property>` isn't limited to injecting `String` values. The `value` attribute can also specify numeric (`int`, `float`, `java.lang.Double`, and so on) values as well as `boolean` values.

For example, let's pretend that the `Instrumentalist` class has an `age` property of type `int` to indicate the age of the instrumentalist. You could set Kenny's age using the following XML:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="age" value="37" />
</bean>
```

Note that the `value` attribute is used exactly the same when setting a numeric value as it is when setting a `String` value. Spring will determine the correct type for the value based on the property's type. Since the `age` property is an `int`, Spring knows to convert 37 to an `int` value before calling `setAge()`.

Using `<property>` to configure simple properties of a bean is great, but there's more to DI than just wiring hardcoded values. The real value of DI is found in wiring an application's collaborating objects together so that they don't have to wire themselves together. To that aim, let's see how to give Kenny an instrument that he can play.

2.2.2 Referencing other beans

Kenny's a talented instrumentalist and can play virtually any instrument given to him. As long as it implements the `Instrument` interface, he can make music with it. Naturally, Kenny does have a favorite instrument. His instrument of choice is the saxophone, which is defined by the `Saxophone` class.

Listing 2.6 A saxophone implementation of `Instrument`

```
package com.springinaction.springidol;

public class Saxophone implements Instrument {
  public Saxophone() {
  }
```

```
  public void play() {
    System.out.println("TOOT TOOT TOOT");
  }
}
```

Before you can give Kenny a saxophone to play, you must declare it as a bean in the Spring container. The following XML should do:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone" />
```

Note that the `Saxophone` class has no properties that need to be set. Consequently, there's no need for `<property>` declarations in the `saxophone` bean.

With the `saxophone` declared, we're ready to give it to Kenny to play. The following modification to the `kenny` bean uses setter injection to set the `instrument` property:

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Now the `kenny` bean has been injected with all of its properties and Kenny is ready to perform. As with Duke, you can prompt Kenny to perform by executing the following Java code (perhaps in a `main()` method):

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
        "com/springinaction/springidol/spring-idol.xml");
Performer performer = (Performer) ctx.getBean("kenny");
performer.perform();
```

This isn't the exact code that will run the *Spring Idol* competition, but it does give Kenny a chance to practice. When it's run, the following will be printed:

```
Playing Jingle Bells : TOOT TOOT TOOT
```

At the same time, it illustrates an important concept. If you compare this code with the code that instructed Duke to perform, you'll find that it isn't much different. In fact, the only difference is the name of the bean retrieved from Spring. The code is the same, even though one causes a juggler to perform and the other causes an instrumentalist to perform.

This isn't a feature of Spring as much as it's a benefit of coding to interfaces. By referring to a performer through the `Performer` interface, we can blindly cause any type of performer to perform, whether it's a poetic juggler or a saxophonist. Spring encourages the use of interfaces for this reason. And, as you're about to see, interfaces work hand in hand with DI to provide loose coupling.

As mentioned, Kenny can play virtually any instrument that's given to him as long as it implements the `Instrument` interface. Although he favors the saxophone, we could also ask Kenny to play piano. For example, consider the `Piano` class.

Listing 2.7 A piano implementation of `Instrument`

```
package com.springinaction.springidol;

public class Piano implements Instrument {
  public Piano() {
  }

  public void play() {
    System.out.println("PLINK PLINK PLINK");
  }
}
```

The `Piano` class can be declared as a `<bean>` in Spring using the following XML:

```
<bean id="piano"
        class="com.springinaction.springidol.Piano" />
```

Now that a piano is available, changing Kenny's instrument is as simple as changing the `kenny` bean declaration as follows:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument" ref="piano" />
</bean>
```

With this change, Kenny will play a piano instead of a saxophone. But because the `Instrumentalist` class only knows about its `instrument` property through the `Instrument` interface, nothing about the `Instrumentalist` class needed to change to support a new implementation of `Instrument`. Although an `Instrumentalist` can play either a `Saxophone` or `Piano`, it's decoupled from both. If Kenny decides to take up the hammered dulcimer, the only change required will be to create a `HammeredDulcimer` class and to change the `instrument` property on the `kenny` bean declaration.

INJECTING INNER BEANS

We've seen that Kenny can play saxophone, piano, or any instrument that implements the `Instrument` interface. But it's also true that the `saxophone` and `piano` beans could also be shared with any other bean by injecting them into an `instrument` property. So, not only can Kenny play any `Instrument`, any `Instrumentalist` can play the `saxophone` bean. In fact, it's common for beans to be shared among other beans in an application.

The problem is that Kenny's concerned with the hygienic implications of sharing his saxophone with others. He'd rather keep his saxophone to himself. To help Kenny avoid germs, we'll use a handy Spring technique known as *inner beans*.

As a Java developer, you're probably already familiar with the concept of inner classes—classes that are defined within the scope of other classes. Similarly, inner beans are beans that are defined within the scope of another bean. To illustrate, consider this new configuration of the `kenny` bean where his saxophone is declared as an inner bean:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument">
    <bean class="org.springinaction.springidol.Saxophone" />
  </property>
</bean>
```

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.

Inner beans aren't limited to setter injection. You may also wire inner beans into constructor arguments, as shown in this new declaration of the `duke` bean:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler">
  <constructor-arg value="15" />
  <constructor-arg>
    <bean class="com.springinaction.springidol.Sonnet29" />
  </constructor-arg>
</bean>
```

Here, a `Sonnet29` instance will be created as an inner bean and sent as an argument to the `PoeticJuggler`'s constructor.

Note that the inner beans don't have an `id` attribute set. Though it's perfectly legal to declare an ID for 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 useful for 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.

2.2.3 *Wiring properties with Spring's p namespace*

Wiring values and references into bean properties using the `<property>` element isn't that daunting. Nevertheless, Spring's `p` namespace offers a way to wire bean properties that doesn't require so many angle brackets.

The `p` namespace has a schema URI of http://www.springframework.org/schema/p. To use it, simply add a declaration for it in the Spring XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:p="http://www.springframework.org/schema/p"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

With it declared, you can now use `p:`-prefixed attributes of the `<bean>` element to wire properties. An an example, look at the following declaration of the `kenny` bean:

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist"
      p:song = "Jingle Bells"
      p:instrument-ref = "saxophone" />
```

The `p:song` attribute is set to `"Jingle Bells"`, wiring the `song` property with that value. Meanwhile, the `p:instrument-ref` attribute is set to `"saxophone"`, effectively wiring the `instrument` property with a reference to the bean whose ID is *saxophone*. The `-ref` suffix serves as a clue to Spring that a reference should be wired instead of a literal value.

The choice between `<property>` and the `p` namespace is up to you. They work equally well. The primary benefit of the `p` namespace is that it's more terse. That works well when trying to write examples for a book with fixed margins. Therefore, you're likely to see me use the `p` namespace from time to time throughout this book—especially when horizontal space is tight.

At this point, Kenny's talent extends to virtually any instrument. Nevertheless, he does have one limitation: he can play only one instrument at a time. Next to take the stage in the *Spring Idol* competition is Hank, a performer who can simultaneously play multiple instruments.

2.2.4 Wiring collections

Up to now, you've seen how to use Spring to configure both simple property values (using the `value` attribute) and properties with references to other beans (using the `ref` attribute). But `value` and `ref` are only useful when your bean's properties are singular. How can Spring help you when your bean has properties that are plural—what if a property is a collection of values?

Spring offers four types of collection configuration elements that come in handy when configuring collections of values. Table 2.3 lists these elements and what they're good for.

The `<list>` and `<set>` elements are useful when configuring properties that are either arrays or some implementation of `java.util.Collection`. As you'll soon see, the actual implementation of the collection used to define the property has little correlation to the choice of `<list>` or `<set>`. Both elements can be used almost interchangeably with properties of any type of `java.util.Collection`.

As for `<map>` and `<props>`, these two elements correspond to collections that are `java.util.Map` and `java.util.Properties`, respectively. These types of collections

Table 2.3 Just as Java has several kinds of collections, Spring allows for injecting several kinds of collections

Collection element	Useful for...
`<list>`	Wiring a list of values, allowing duplicates
`<set>`	Wiring a set of values, ensuring no duplicates
`<map>`	Wiring a collection of name-value pairs where name and value can be of any type
`<props>`	Wiring a collection of name-value pairs where the name and value are both `String`s

are useful when you need a collection that's made up of a collection of key-value pairs. The key difference between the two is that when using `<props>`, both the keys and values are `String`s, whereas `<map>` allows keys and values of any type.

To illustrate collection wiring in Spring, please welcome Hank to the *Spring Idol* stage. Hank's special talent is that he's a one-man band. Like Kenny, Hank's talent is playing several instruments, but Hank can play several instruments at the same time. Hank is defined by the `OneManBand` class.

Listing 2.8 A performer that's a one-man-band

```
package com.springinaction.springidol;

import java.util.Collection;

public class OneManBand implements Performer {
  public OneManBand() {
  }

  public void perform() throws PerformanceException {
    for (Instrument instrument : instruments) {
      instrument.play();
    }
  }

  private Collection<Instrument> instruments;

  public void setInstruments(Collection<Instrument> instruments) {
    this.instruments = instruments;
  }
}
```

Inject instrument collection

As you can see, a `OneManBand` iterates over a collection of instruments when it performs. What's most important here is that the collection of instruments is injected through the `setInstruments()` method. Let's see how Spring can provide Hank with his collection of instruments.

WIRING LISTS, SETS, AND ARRAYS

To give Hank a collection of instruments to perform with, let's use the `<list>` configuration element:

```
<bean id="hank"
      class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <list>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
    </list>
  </property>
</bean>
```

The `<list>` element contains one or more values. Here `<ref>` elements are used to define the values as references to other beans in the Spring context, configuring Hank to play a guitar, cymbal, and harmonica. But it's also possible to use other

value-setting Spring elements as the members of a `<list>`, including `<value>`, `<bean>`, and `<null/>`. In fact, a `<list>` may contain another `<list>` as a member for multidimensional lists.

In listing 2.8, `OneManBand`'s `instruments` property is a `java.util.Collection` using Java 5 generics to constrain the collection to `Instrument` values. But `<list>` may be used with properties that are of any implementation of `java.util.Collection` or an array. In other words, the `<list>` element we just used would still work, even if the `instruments` property were to be declared as

```
java.util.List<Instrument> instruments;
```

or even if it were to be declared as

```
Instrument[] instruments;
```

Likewise, you could also use `<set>` to wire a collection or array property:

```
<bean id="hank"
      class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <set>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
      <ref bean="harmonica" />
    </set>
  </property>
</bean>
```

Again, either `<list>` or `<set>` can be used to wire any implementation of `java.util.Collection` or an array. Just because a property is a `java.util.Set`, that doesn't mean that you must use `<set>` to do the wiring. Even though it may seem odd to configure a `java.util.List` property using `<set>`, it's certainly possible. In doing so, you'll be guaranteed that all members of the `List` will be unique.

WIRING MAP COLLECTIONS

When a `OneManBand` performs, each instrument's sound is printed as the `perform()` method iterates over the collection of instruments. But suppose that you also want to see which instrument is producing each sound. To accommodate this, consider the following changes to the `OneManBand` class.

Listing 2.9 Changing `OneManBand`'s instrument collection to a `Map`

```
package com.springinaction.springidol;

import java.util.Map;
import com.springinaction.springidol.Instrument;
import com.springinaction.springidol.PerformanceException;
import com.springinaction.springidol.Performer;

public class OneManBand implements Performer {
  public OneManBand() {
  }
```

```
  public void perform() throws PerformanceException {
    for (String key : instruments.keySet()) {
      System.out.print(key + " : ");
      Instrument instrument = instruments.get(key);
      instrument.play();
    }
  }

  private Map<String, Instrument> instruments;

  public void setInstruments(Map<String, Instrument> instruments) {
    this.instruments = instruments;
  }
}
```

Inject instrument as map

In the new version of `OneManBand`, the `instruments` property is a `java.util.Map` where each member has a `String` as its key and an `Instrument` as its value. Because a `Map`'s members are made up of key-value pairs, a simple `<list>` or `<set>` configuration element won't suffice when wiring the property.

Instead, the following declaration of the `hank` bean uses the `<map>` element to configure the `instruments` property:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <map>
      <entry key="GUITAR" value-ref="guitar" />
      <entry key="CYMBAL" value-ref="cymbal" />
      <entry key="HARMONICA" value-ref="harmonica" />
    </map>
  </property>
</bean>
```

The `<map>` element declares a value of type `java.util.Map`. Each `<entry>` element defines a member of the `Map`. In the previous example, the `key` attribute specifies the key of the entry whereas the `value-ref` attribute defines the value of the entry as a reference to another bean within the Spring context.

Although our example uses the `key` attribute to specify a `String` key and `value-ref` to specify a reference value, the `<entry>` element actually has two attributes each for specifying the key and value of the entry. Table 2.4 lists those attributes.

`<map>` is only one way to inject key-value pairs into bean properties when either of the objects isn't a `String`. Let's see how to use Spring's `<props>` element to configure `String`-to-`String` mappings.

Table 2.4 An `<entry>` in a `<map>` is made up of a key and a value, either of which can be a primitive value or a reference to another bean. These attributes help specify the keys and values of an `<entry>`.

Attribute	Purpose
`key`	Specifies the key of the map entry as a `String`
`key-ref`	Specifies the key of the map entry as a reference to a bean in the Spring context
`value`	Specifies the value of the map entry as a `String`
`value-ref`	Specifies the value of the map entry as a reference to a bean in the Spring context

WIRING PROPERTIES COLLECTIONS

When declaring a `Map` of values for `OneManBand`'s `instrument` property, it was necessary to specify the value of each entry using `value-ref`. That's because each entry is ultimately another bean in the Spring context.

But if you find yourself configuring a `Map` whose entries have both `String` keys and `String` values, you may want to consider using `java.util.Properties` instead of a `Map`. The `Properties` class serves roughly the same purpose as `Map`, but limits the keys and values to `String`s.

To illustrate, imagine that instead of being wired with a map of `String`s and bean references, `OneManBand` is wired with a `String`-to-`String` `java.util.Properties` collection. The new `instruments` property might be changed to look like this:

```
private Properties instruments;
public void setInstruments(Properties instruments) {
    this.instruments = instruments;
}
```

To wire the instrument sounds into the `instruments` property, we use the `<props>` element in the following declaration of the `hank` bean:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <props>
      <prop key="GUITAR">STRUM STRUM STRUM</prop>
      <prop key="CYMBAL">CRASH CRASH CRASH</prop>
      <prop key="HARMONICA">HUM HUM HUM</prop>
    </props>
  </property>
</bean>
```

The `<props>` element constructs a `java.util.Properties` value where each member is defined by a `<prop>` element. Each `<prop>` element has a `key` attribute that defines the key of each `Properties` member, while the value is defined by the contents of the `<prop>` element. In our example, the element whose key is "GUITAR" has a value of "STRUM STRUM STRUM".

This may be the most difficult Spring configuration element to talk about. That's because the term *property* is highly overloaded. It's important to keep the following straight:

- `<property>` is the element used to inject a value or bean reference into a property of a bean class.
- `<props>` is the element used to define a collection value of type `java.util.Properties`.
- `<prop>` is the element used to define a member of a `<props>` collection.

Up to this point, we've seen how to wire several things into bean properties and constructor arguments. We've wired simple values, references to other beans, and collections. Now, let's see how to wire nothing.

2.2.5 Wiring nothing (null)

You read that right. In addition to all of the other things Spring can wire into a bean property or constructor argument, it can also wire nothing. Or more accurately, Spring can wire a `null`.

You're probably rolling your eyes and thinking, "What's this guy talking about? Why would I ever want to wire `null` into a property? Aren't all properties `null` until they're set? What's the point?"

Though it's often true that properties start out `null` and will remain that way until set, some beans may themselves set a property to a non-`null` default value. What if, for some twisted reason, you want to force that property to be `null`? If that's the case, it's not sufficient to just assume that the property will be `null`—you must explicitly wire `null` into the property.

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

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

Another reason for explicitly wiring `null` into a property is to override an autowired property value. What's autowiring, you ask? Keep reading—we're going to explore autowiring in the next chapter.

For now, you'll want to hold on to your seats. We'll wrap up this chapter by looking at one of the coolest new features in Spring: the Spring Expression Language.

2.3 Wiring with expressions

So far all of the stuff we've wired into bean properties and constructor arguments has been statically defined in the Spring configuration XML. When we wired the name of a song into the `Instrumentalist` bean, that value was determined at development time. And when we wired references to other beans, those references were also statically determined while we wrote the Spring configuration.

What if we want to wire properties with values that aren't known until runtime?

Spring 3 introduced the *Spring Expression Language (SpEL)*, a powerful yet succinct way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime. Using SpEL, you can pull off amazing feats of bean wiring that would be much more difficult (or even impossible) using Spring's traditional wiring style.

SpEL has a lot of tricks up its sleeves, including

- The ability to reference beans by their ID
- Invoking methods and accessing properties on objects
- Mathematical, relational, and logical operations on values
- Regular expression matching
- Collection manipulation

Writing a SpEL expression involves piecing together the various elements of the SpEL syntax. Even the most interesting SpEL expressions are often composed of simpler

expressions. So before we can start running with SpEL, let's take our first steps with some of the most basic ingredients of a SpEL expression.

2.3.1 Expressing SpEL fundamentals

The ultimate goal of a SpEL expression is to arrive at some value after evaluation. In the course of calculating that value, other values are considered and operated upon. The simplest kinds of values that SpEL can evaluate may be literal values, references to a bean's properties, or perhaps a constant on some class.

LITERAL VALUES

The simplest possible SpEL expression is one that contains only a literal value. For example, the following is a perfectly valid SpEL expression:

```
5
```

Not surprisingly, this expression evaluates to an integer value of 5. We could wire this value into a bean's property by using `#{}` markers in a `<property>` element's `value` attribute like this:

```
<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-SpEL values as well:

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

Floating-point numbers can also be expressed in SpEL. For example:

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

Numbers can even be expressed in scientific notation. As an example, the following snippet of code sets a `capacity` property to 10000.0 using scientific notation:

```
<property name="capacity" value="#{1e4}"/>
```

Literal `String` values can also be expressed in SpEL with either single or double quote marks. For example, to wire a literal `String` value into a bean property, we could express it like this:

```
<property name="name" value="#{'Chuck'}"/>
```

Or if you were using single quote marks for XML attribute values, then you'd want to use double quotes in the SpEL expression:

```
<property name='name' value='#{"Chuck"}'/>
```

A couple of other literal values you may use are the Boolean `true` and `false` values. For example, you could express a `false` like this:

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

Working with literal values in SpEL expressions is mundane. After all, we don't need SpEL to set an integer property to 5 or a Boolean property to `false`. I admit that

there's not much use to SpEL expressions that only contain literal values. But remember that more interesting SpEL expressions are composed of simpler expressions. So it's good to know how to work with literal values in SpEL. We'll eventually need them as our expressions get more complex.

REFERENCING BEANS, PROPERTIES, AND METHODS

Another basic thing that a SpEL expression can do is to reference another bean by its ID. For example, you could use SpEL to wire one bean into another bean's property by using the bean ID as the SpEL expression:

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

As you can see, we're using SpEL to wire the bean whose ID is `"saxophone"` into an `instrument` property. But wait… can't we do that without SpEL by using the `ref` attribute as follows?

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

Yes, the outcome is the same. And yes, we didn't need SpEL to do that. But it's interesting that we can do that, and in a moment I'll show you a few tricks that take advantage of being able to wire bean references with SpEL. Right now I want to show how you can use a bean reference to access the properties of the bean in a SpEL expression.

Let's say that you want to configure a new `Instrumentalist` bean whose ID is `carl`. The funny thing about Carl is that he's a copycat performer. Instead of performing his own song, he's going to be wired to perform whatever song Kenny plays. When you configure the `carl` bean, you can use SpEL to copy Kenny's song into the `song` property like this:

```
<bean id="carl"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="#{kenny.song}" />
</bean>
```

As illustrated in figure 2.1, the expression passed into Carl's `song` property is made up of two parts.

The first part (the part before the period delimiter) refers to the `kenny` bean by its ID. The second part refers to the `song` attribute of the `kenny` bean. By wiring the `carl` bean's `song` property this way, it's effectively as if you programmatically performed the following Java code:

```
Instrumentalist carl = new Instrumentalist();
carl.setSong(kenny.getSong());
```

Ah! We're finally doing something slightly interesting with SpEL. It's a humble expression, but I can't imagine an easier way to pull off the same thing without SpEL.

Trust me… we're just getting started.

Referencing a bean's properties isn't the only thing you can do with a bean. You could also invoke a method. For example, imagine

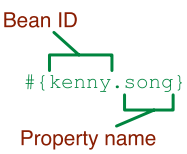

Figure 2.1 Referring to another bean's property using the Spring Expression Language

that you have a `songSelector` bean which has a `selectSong()` method on it that returns a song to be sung. In that case, Carl could quit copycatting and start singing whatever song the `songSelector` bean suggests:

```
<property name="song" value="#{songSelector.selectSong()}"/>
```

Now suppose that (for whatever reason), Carl wants the song given to him in all uppercase. No problem… all you need to do is invoke the `toUpperCase()` method on the `String` value you're given:

```
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
```

That'll do the trick every time… as long as the `selectSong()` method doesn't return a `null`. If `selectSong()` were to return `null`, then you'd get a `NullPointerException` as the SpEL expression is being evaluated.

The way to avoid the dreaded `NullPointerException` in SpEL is to use the null-safe accessor:

```
<property name="song" value="#{songSelector.selectSong()?.toUpperCase()}"/>
```

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 SpEL wouldn't even try to invoke `toUpperCase()` on it.

Writing expressions that work with other beans is a good start. But what if you need to invoke a static method or reference a constant? For that we'll need to see how to work with types in SpEL.

WORKING WITH TYPES

The key to working with class-scoped methods and constants in SpEL is to use the `T()` operator. For example, to express Java's `Math` class in SpEL, you'd need to use the `T()` operator like this:

```
T(java.lang.Math)
```

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

For example, suppose that you need to wire the value of pi into a bean property. In that case, simply reference the `Math` class's `PI` constant like this:

```
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>
```

Similarly, static methods can also be invoked on the result of the `T()` operator. For instance, here's how to wire a random number (between 0 and 1) into a bean property:

```
<property name="randomNumber" value="#{T(java.lang.Math).random()}"/>
```

When the application is starting up and Spring is wiring the `randomNumber` property, it'll use the `Math.random()` method to determine a value for that property. This is

another example of a SpEL expression that I can't think of a simpler way to do without SpEL.

Now that we've added a few of the most basic SpEL expressions to our bag of tricks, let's step things up a bit by looking at the types of operations we can perform on those simpler expressions.

2.3.2 Performing operations on SpEL values

SpEL offers several operations that you can apply on values in a SpEL expression. These operations are summarized in table 2.5.

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

Operation type	Operators
Arithmetic	`+`, `-`, `*`, `/`, `%`, `^`
Relational	`<`, `>`, `==`, `<=`, `>=`, `lt`, `gt`, `eq`, `le`, `ge`
Logical	`and`, `or`, `not`, `|`
Conditional	`?:` (ternary), `?:` (Elvis)
Regular expression	`matches`

The first kind of operations we'll play with are the ones that let us perform basic math on values in a SpEL expression.

DOING MATH WITH SPEL

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

For example, to add two numbers together, the `+` operator can be used like this:

```
<property name="adjustedAmount" value="#{counter.total + 42}"/>
```

Here we're adding 42 to the value of the `counter` bean's `total` property. Note that although both sides of the `+` operator are numeric, they don't have to be literal values. In this case, the left side is a SpEL expression in its own right.

The other arithmetic operators work in SpEL just as you'd expect them to in Java. The `-` operator, for instance, performs subtraction:

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

The `*` operator performs multiplication:

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

The `/` operator performs division:

```
<property name="average" value="#{counter.total / counter.count}"/>
```

And the `%` operator performs a modulo operation:

```
<property name="remainder" value="#{counter.total % counter.count}"/>
```

Unlike Java, SpEL also offers a power-of operator in the form of the carat:

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

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

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

Again, this is consistent Java in that the + operator can be used to concatenate `String` values there, too.

COMPARING VALUES

It's often useful to compare two values to decide whether they're equal or which is greater than the other. For that kind of comparison, SpEL offers all of the expected comparison operators that Java itself also offers.

As an example, to compare two numbers for equality, you can use the double-equal (==) operator:

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

In this case, it's assumed that the `equal` property is a Boolean and it'll be wired with a `true` if the `total` property is equal to 100.

Similarly, the less-than (<) and greater-than (>) operators can be used to compare different values. Likewise, SpEL also supports the greater-than-or-equals (>=) and the less-than-or-equals (<=) operators. For example, the following is a perfectly valid SpEL expression:

```
counter.total <= 100000
```

Unfortunately, the less-than and greater-than symbols pose a problem when using these expressions in Spring's XML configuration, as they have special meaning in XML. So, when using SpEL in XML,[5] it's best to use SpEL's textual alternatives to these operators. For example:

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

Here, the `le` operator means less than or equals. The other textual comparison operators are cataloged in table 2.6.

You'll notice that even though the symbolic equals operator (==) doesn't present any issues in XML, SpEL offers a textual `eq` operator in the interest of consistency with the other operators, and because some developers may prefer the textual operators over the symbolic ones.

Operation	Symbolic	Textual
Equals	==	eq
Less than	<	lt
Less than or equals	<=	le
Greater than	>	gt
Greater than or equals	>=	ge

Table 2.6 SpEL includes several operators that you can use to manipulate the values of an expression.

[5] We'll see how to use SpEL outside of Spring XML configuration in the next chapter.

LOGICAL EXPRESSIONS

It's great that we can evaluate comparisons in SpEL, but what if you need to evaluate based on two comparisons? Or what if you want to negate some Boolean value? That's where the logical operators come into play. Table 2.7 lists all of SpEL's logical operators.

Table 2.7 SpEL includes several operators that you can use to manipulate the values of an expression.

Operator	Operation
`and`	A logical AND operation; both sides must evaluate true for the expression to be true
`or`	A logical OR operation; either side must evaluate true for the expression to be true
`not or !`	A logical NOT operation; negates the target of the operation

As an example, consider the following use of the `and` operator:

```
<property name="largeCircle"
          value="#{shape.kind == 'circle' and shape.perimeter gt 10000}"/>
```

In this case, the `largeCircle` property will be set to `true` if the `kind` property of `shape` is `"circle"` and the `perimeter` property is a number greater than 10000. Otherwise, it'll remain false.

To negate a Boolean expression, you have two operators to choose from: either the symbolic `!` operator or the textual `not` operator. For example, the following use of the `!` operator

```
<property name="outOfStock" value="#{!product.available}"/>
```

is equivalent to this use of the `not` operator:

```
<property name="outOfStock" value="#{not product.available}"/>
```

Strangely, SpEL doesn't offer symbolic equivalents for the `and` and `or` operators.

CONDITIONALLY EVALUATING

What if you want a SpEL expression to evaluate to one value if a condition is true and a different value otherwise? For example, let's say that Carl (the `Instrumentalist` from earlier) wants to play a piano if the song is "Jingle Bells," but he'll play a saxophone otherwise. In that case, you can use SpEL's ternary (`?:`) operator:

```
<property name="instrument"
   value="#{songSelector.selectSong()=='Jingle Bells'?piano:saxophone}"/>
```

As you can see, SpEL's ternary operator works the same as Java's ternary operator. In this case, the `instrument` property will be wired with a reference to the `piano` bean if the song selected is "Jingle Bells." Otherwise, it'll be wired with the bean whose ID is `saxophone`.

A common use of the ternary operator is to check for a `null` value and to wire a default value in the event of a `null`. For example, suppose we want to configure Carl

to perform the same song as Kenny, unless Kenny doesn't have a song. In that case, Carl's song should default to "Greensleeves." The ternary operator could be used as follows to handle this case:

```
<property name="song"
          value="#{kenny.song != null ? kenny.song : 'Greensleeves'}"/>
```

Although that'll work, there's a bit of duplication in that we refer to `kenny.song` twice. SpEL offers a variant of the ternary operator that simplifies this expression:

```
<property name="song" value="#{kenny.song ?: 'Greensleeves'}"/>
```

As in the previous example, the expression will evaluate to the value of `kenny.song` or "Greensleeves" if `kenny.song` is `null`. When used this way, `?:` is referred to as the *Elvis operator*. This strange name comes from using the operator as a type of smiley where the question mark appears to form the shape of Elvis Presley's hair. [6]

REGULAR EXPRESSIONS IN SPEL

When working with text, it's sometimes useful to check whether that text matches a certain pattern. SpEL supports pattern matching in expressions with its `matches` operator.

The `matches` operator attempts to apply a regular expression (given as its right-side argument) against a `String` value (given as the left-side argument). The result of a `matches` evaluation is a Boolean value: `true` if the value matches the regular expression, `false` otherwise.

To demonstrate the `matches` operator, suppose that we want to check whether a `String` contains a valid email address. In that case, we could apply the `matches` operator like this:

```
<property name="validEmail" value=
  "#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'}"/>
```

Exploring the mysteries of the enigmatic regular expression syntax is outside the scope of this book. And I realize that the regular expression given here is not robust enough to cover all scenarios. But for the purposes of showing off the `matches` operator, it'll suffice.

Now that we've seen how to evaluate expressions concerning simple values, let's look at the kind of magic that SpEL can perform on collections.

2.3.3 Sifting through collections in SpEL

Some of SpEL's most amazing tricks involve working with collections. Sure, you can reference a single member of a collection in SpEL, just like in Java. But SpEL also has the power to select members of a collection based on the values of their properties. It can also extract properties out of the collection members into a new collection.

For demonstration purposes, suppose that you have a `City` class that's defined as follows (with getter/setter methods removed to conserve space):

---

[6] Don't blame me. I didn't come up with that name. But it kind of looks like Elvis's hair, doesn't it?

```
package com.habuma.spel.cities;
public class City {
  private String name;
  private String state;
  private int population;
}
```

And, let's suppose that you've configured a list of `City` objects in Spring by using the `<util:list>` element as shown next.

Listing 2.10 A list of cities, defined using Spring's `<util:list>` element

```
<util:list id="cities">
  <bean class="com.habuma.spel.cities.City"
     p:name="Chicago" p:state="IL" p:population="2853114"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Atlanta" p:state="GA" p:population="537958"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Dallas" p:state="TX" p:population="1279910"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Houston" p:state="TX" p:population="2242193"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Odessa" p:state="TX" p:population="90943"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="El Paso" p:state="TX" p:population="613190"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Jal" p:state="NM" p:population="1996"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Las Cruces" p:state="NM" p:population="91865"/>
</util:list>
```

The `<util:list>` element comes from Spring's `util` namespace. It effectively creates a bean of type `java.util.List` that contains all of the values or beans that it contains. In this case, that's a list of eight `City` beans.

SpEL offers a few handy operators for working with collections such as this.

ACCESSING COLLECTION MEMBERS

The most basic thing we could do here is extract a single element out of the list and wire it into a property:

```
<property name="chosenCity" value="#{cities[2]}"/>
```

In this case, I've selected the third city out of the zero-based `cities` list and wired it into the `chosenCity` property. To spice up the example, I suppose you could randomly choose a city:

```
<property name="chosenCity"
          value="#{cities[T(java.lang.Math).random() * cities.size()]}"/>
```

In any event, the square-braces (`[]`) operator serves to access a member of the collection by its index.

The `[]` operator is also good for retrieving a member of a `java.util.Map` collection. For example, suppose the `City` objects were in a `Map` with their name as the key. In that case, we could retrieve the entry for Dallas like this:

```
<property name="chosenCity" value="#{cities['Dallas']}"/>
```

Another use of the `[]` operator is to retrieve a value from a `java.util.Properties` collection. For example, suppose that you were to load a properties configuration file into Spring using the `<util:properties>` element as follows:

```
<util:properties id="settings"
    location="classpath:settings.properties"/>
```

Here the `settings` bean will be a `java.util.Properties` that contains all of the entries in the file named `settings.properties`. With SpEL, you can access a property from that file in the same way you access a member of a `Map`. For example, the following use of SpEL reads a property whose name is `twitter.accessToken` from the `settings` bean:

```
<property name="accessToken" value="#{settings['twitter.accessToken']}"/>
```

In addition to reading properties from a `<util:properties>`-declared collection, Spring makes two special selections of properties available to SpEL: `systemEnvironment` and `systemProperties`.

`systemEnvironment` contains all of the environment variables on the machine running the application. It's just a `java.util.Properties` collection, so the square braces can be used to access its members by their key. For example, on my MacOS X machine, I can inject the user's home directory path into a bean property like this:

```
<property name="homePath" value="#{systemEnvironment['HOME']}"/>
```

Meanwhile, `systemProperties` contains all of the properties that were set in Java as the application started (typically using the `-D` argument). Therefore, if the JVM were started with `-Dapplication.home=/etc/myapp`, then you could wire that value into the `homePath` property with the following SpEL incantation:

```
<property name="homePath" value="#{systemProperties['application.home']}"/>
```

Although it doesn't have much to do with working with collections, it's worth noting that the `[]` operator can also be used on `String` values to retrieve a single character by its index within the `String`. For example, the following expression will evaluate to `"s"`:

```
'This is a test'[3]
```

Accessing individual members of a collection is handy. But with SpEL, we can also select members of a collection that meet certain criteria. Let's give collection selection a try.

SELECTING COLLECTION MEMBERS

Let's say that you want to narrow the list of cities down to only those whose population is greater than 100,000. One way to do this is to wire the entire `cities` bean into a property and place the burden of sifting out the smaller cities on the receiving bean. But with SpEL, it's a simple matter of using a selection operator (`.?[]`) when doing the wiring:

```
<property name="bigCities" value="#{cities.?[population gt 100000]}"/>
```

The selection operator will create a new collection whose members include only those members from the original collection that meet the criteria expressed between the square braces. In this case, the `bigCities` property will be wired with a list of `City` objects whose `population` property exceeds 100,000.

SpEL also offers two other selection operators, `.^[]` and `.$[]`, for selecting the first and last matching items (respectively) from a 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 `City` representing Chicago would be wired into the `aBigCity` property. Likewise, the `City` object representing El Paso could be selected as follows:

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

We'll revisit collection selection in a moment. But first, let's see how to project properties from a collection into a new collection.

PROJECTING COLLECTIONS

Collection projection involves collecting a particular property from each of the members of a collection into a new collection. SpEL's projection operator (`.![]`) can do exactly that.

For example, suppose that instead of a list of `City` objects, what you want is just a list of `String` objects containing the names of the cities. To get a list of just the city names, you could wire a `cityNames` property like this:

```
<property name="cityNames" value="#{cities.![name]}"/>
```

As a result of this expression, the `cityNames` property will be given a list of `String`s, including values such as Chicago, Atlanta, Dallas, and so forth. The `name` property within the square braces decides what each member of the resulting list will contain.

But projection isn't limited to projecting a single property. With a slight change to the previous example, you can get a list of city and state names:

```
<property name="cityNames" value="#{cities.![name + ', ' + state]}"/>
```

Now the `cityNames` property will be given a list containing values such as "Chicago, IL", "Atlanta, GA", and "Dallas, TX".

For my final SpEL trick, let me bring collection selection and projection together. Here's how you might wire a list of only big city names into the `cityNames` property:

```
<property name="cityNames"
    value="#{cities.?[population gt 100000].![name + ', ' + state]}"/>
```

Since the outcome of the selection operation is a new list of `City` objects, there's no reason why I can't use projection on that new collection to get the names of all of the big cities.

This demonstrates that you can assemble simple SpEL expressions into more interesting (and more complex) expressions. It's easy to see how that's a powerful feature.

But it doesn't take much of a stretch to realize that it's also dangerous. SpEL expressions are ultimately just `String`s that are tricky to test and have no IDE support for syntax checking.

I encourage you to use SpEL wherever it can simplify what would otherwise be difficult (or even impossible) wirings. But be careful to not get too carried away with SpEL. Fight the temptation to put too much logic into a SpEL expression.

We'll see some more SpEL later on and used in ways other than bean wiring. In the next chapter we'll break SpEL out of XML and use it in annotation-driven wiring. And, as we'll see in chapter 9, SpEL plays a significant role in the latest version of Spring Security.

2.4 Summary

At the core of the Spring Framework is the Spring container. Spring comes with several implementations of its container, but they all fall into one of two categories. A `BeanFactory` is the simplest form of container, providing basic DI and bean-wiring services. But when more advanced framework services are needed, Spring's `ApplicationContext` is the container to use.

In this chapter, you've seen how to wire beans together within the Spring container. Wiring is typically performed within a Spring container using an XML file. This XML file contains configuration information for all of the components of an application, along with information that helps the container perform DI to associate beans with other beans that they depend on.

Now that you know how to wire beans using XML, I'll show you how to use less XML. In the next chapter, we'll look at how to take advantage of automatic wiring and annotations to reduce the amount of XML configuration in a Spring application.