PART 2

Spring Basics

CHAPTER 4

Introducing Inversion of Control

In Chapter 1, during the first discussion of Inversion of Control (IoC), you might recall that we mentioned that it had been renamed, by Martin Fowler, the more descriptive Dependency Injection (DI). However, this is not strictly true; in reality, DI is a specialized form of IoC, although you will often find that the two terms are used interchangeably. In this chapter, we take a much more detailed look at IoC and DI, formalizing the relationship between the two concepts and looking in great detail at how Spring fits into the picture.

After defining both and looking at Spring's relationship with them, we will explore the concepts that are essential to Spring's implementation of DI. This chapter only covers the basics of Spring's DI implementation; we discuss more advanced DI features in Chapter 5 and look at DI in the context of application design in both Chapters 5 and 11. More specifically this chapter will cover the following topics:

Inversion of Control concepts: In this section, we discuss the various kinds of IoC including Dependency Injection and Dependency Lookup. This section looks at the differences between the various IoC approaches and presents the pros and cons of each.

Inversion of Control in Spring: This section looks at IoC capabilities available in Spring and how these capabilities are implemented. In particular, this section looks at Dependency Injection and the setter-based and constructor-based approaches Spring offers. This section also provides the first full discussion of the `BeanFactory` interface, which is central to the whole Spring framework.

XML configuration for Spring `BeanFactories`: The final part of this chapter focuses on using the XML-based configuration approach for the `BeanFactory` configuration. This section starts out with a discussion of DI configuration and moves on to look at additional services provided by the `BeanFactory` such as bean inheritance, lifecycle management, and autowiring.

Inversion of Control and Dependency Injection

At its core, IoC, and therefore DI also, aims to offer a simpler mechanism for provisioning component dependencies (often referred to as an object's **collaborators**) and managing these dependencies throughout their lifecycles. A component that requires certain dependencies is often referred to as the **dependent object** or, in the case of IoC, the target. This is a rather grand way of saying that IoC provides services through which a component can access its dependencies and services for interacting with the dependencies throughout their life. In general, IoC can be decomposed into two subtypes: Dependency Injection and Dependency Lookup. These subtypes are further decomposed into concrete implementations of the IoC services. From this definition, you can clearly see that when we are talking about DI we are always talking about IoC, but when we are talking about IoC we are not always talking about DI.

Types of Inversion of Control

You may be wondering why there are two different types of IoC and why these types are split further into different implementations. There seems to be no clear answer to this question; certainly the different types provide a level of flexibility, but to us, it seems that IoC is more of a mixture of old and new ideas; the two different types of IoC represent this.

Dependency Lookup is a much more traditional approach and at first glance, it seems more familiar to Java programmers. Dependency Injection is a newer, less well-established approach that, although it appears counterintuitive at first, is actually much more flexible and usable than Dependency Lookup.

With Dependency Lookup–style IoC, a component must acquire a reference to a dependency, whereas with Dependency Injection, the dependencies are literally injected into the component by the IoC container. Dependency Lookup comes in two types: Dependency Pull and Contextualized Dependency Lookup (CDL). Dependency Injection also has two common flavors: Constructor Dependency Injection and Setter Dependency Injection.

■ **Note** For the discussions in this section, we are not concerned with how the fictional IoC container comes to know about all the different dependencies, just that at some point, it performs the actions described for each mechanism.

Dependency Pull

To a Java developer, Dependency Pull is the most familiar type of IoC. In Dependency Pull, dependencies are pulled from a registry as required. Anyone who has ever written code to access an EJB has used Dependency Pull. Spring also offers Dependency Pull as a mechanism for retrieving components the framework manages; you saw this in action in Chapter 2. Listing 4-1 shows a typical Dependency Pull lookup in a Spring-based application.

Listing 4-1. *Dependency Pull in Spring*

```
public static void main(String[] args) throws Exception {

    // get the bean factory
    BeanFactory factory = getBeanFactory();

    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
}
```

This kind of IoC is not only prevalent in J2EE-based applications, which make extensive use of JNDI lookups to obtain dependencies from a registry, but it is also pivotal to working with Spring in many environments.

Contextualized Dependency Lookup

Contextualized Dependency Lookup (CDL) is similar, in some respects, to Dependency Pull, but in CDL, lookup is performed against the container that is managing the resource, not from some central registry, and it is usually performed at some set point. CDL works by having the component implement an interface similar to that in Listing 4-2.

Listing 4-2. *Component Interface for CDL*

```
package com.apress.prospring.ch4;

public interface ManagedComponent {

    public void performLookup(Container container);
}
```

By implementing this interface, a component is signaling to the container that it wishes to obtain a dependency. When the container is ready to pass dependencies to a component, it calls `performLookup()` on each component in turn. The component can then look up its dependencies using the `Container` interface, as shown in Listing 4-3.

Listing 4-3. *Obtaining Dependencies in CDL*

```
package com.apress.prospring.ch4;

public class ContextualizedDependencyLookup implements ManagedComponent {

    private Dependency dep;

    public void performLookup(Container container) {
        this.dep = (Dependency) container.getDependency("myDependency");
    }

}
```

Constructor Dependency Injection

Constructor Dependency Injection is Dependency Injection where a component's dependencies are provided to it in its constructor(s). The component declares a constructor or a set of constructors taking as arguments its dependencies, and the IoC container passes the dependencies to the component when it instantiates it, as shown in Listing 4-4.

Listing 4-4. *Constructor Dependency Injection*

```
package com.apress.prospring.ch4;

public class ConstructorInjection {

    private Dependency dep;

    public ConstructorInjection(Dependency dep) {
        this.dep = dep;
    }
}
```

Setter Dependency Injection

In Setter Dependency Injection, the IoC container injects a component's dependencies into the component via JavaBean-style setter methods. A component's setters expose the set of the dependencies the IoC container can manage. Listing 4-5 shows a typical Setter Dependency Injection–based component.

Listing 4-5. *Setter Dependency Injection*

```
package com.apress.prospring.ch4;

public class SetterInjection {

    private Dependency dep;

    public void setMyDependency(Dependency dep) {
        this.dep = dep;
    }
}
```

Within the container, the dependency requirement exposed by the `setMyDependency()` method is referred to by the JavaBeans-style name, `myDependency`. In practice, setter injection is the most widely used injection mechanism, and it is one of the simplest IoC mechanisms to implement.

Injection vs. Lookup

Choosing which style of IoC to use—Injection or Lookup—is not usually a difficult decision. In many cases, the type of IoC you use is mandated by the container you are using. For instance,

if you are using EJB 2.0, then you must use Lookup-style IoC to obtain the EJB from the J2EE container. In Spring, aside from initial bean lookups, your components and their dependencies are always wired together using Injection-style IoC.

■ **Note** When you are using Spring, you can access EJB resources without needing to perform an explicit lookup. Spring can act as an adapter between Lookup- and Injection-style IoC systems, thus allowing you to manage all resources using Injection.

The real question is this: Given the choice, which method should you use, Injection or Lookup? The answer to this is most definitely Injection. If you look at the code in Listings 4-4 and 4-5, you can clearly see that using Injection has zero impact on your components' code. The Dependency Pull code, on the other hand, must actively obtain a reference to the registry and interact with it to obtain the dependencies, and using CDL requires your classes to implement a specific interface and look up all dependencies manually. When you are using Injection, the most your classes have to do is allow dependencies to be injected using either constructors or setters.

Using Injection, you are free to use your classes completely decoupled from the IoC container supplying dependent objects with their collaborators manually, whereas with Lookup, your classes are always dependent on classes and interfaces defined by the container. Another drawback with Lookup is that it becomes very difficult to test your classes in isolation from the container. Using Injection, testing your components is trivial, because you can simply provide the dependencies yourself using the appropriate constructor or setter.

■ **Note** For a more complete discussion of testing using Dependency Injection and Spring, refer to Appendix A.

Lookup-based solutions are, by necessity, more complex than Injection-based ones. Although complexity is nothing to be afraid of, we question the validity of adding unneeded complexity to a process as core to your application as dependency management.

All of these reasons aside, the biggest reason to choose Injection over Lookup is that it makes your life easier. You write substantially less code when you are using Injection, and the code that you do write is simple and can, in general, be automated by a good IDE. You will notice that all of the code in the Injection samples is passive, in that it doesn't actively try to accomplish a task; the most exciting thing you see in Injection code is objects getting stored in a field—not much can go wrong there! Passive code is much simpler to maintain than active code, because there is very little that can go wrong. Consider the following code taken from Listing 4-3:

```
public void performLookup(Container container) {
    this.dep = (Dependency) container.getDependency("myDependency");
}
```

In this code, plenty could go wrong: the dependency key could change, the container instance could be null, or the returned dependency might be the incorrect type. We refer to this code as having a lot of moving parts because plenty of things can break. Using Lookup might decouple the components of your application, but it adds complexity in the additional code required to couple these components back together in order to perform any useful tasks.

Setter Injection vs. Constructor Injection

Now that we have established which method of IoC is preferable, we still need to choose whether to use setter injection or constructor injection. Constructor injection is particularly useful when you absolutely must have an instance of the dependency class before your component is used. Many containers, Spring included, provide a mechanism for ensuring that all dependencies are defined when you use setter injection, but by using constructor injection you assert the requirement for the dependency in a container-agnostic manner.

Setter injection is useful in a variety of different cases. If the component is exposing its dependencies to the container but is happy to provide its own defaults, then setter injection is usually the best way to accomplish this. Another benefit of setter injection is that it allows dependencies to be declared on an interface, although this is not as useful as you might first think. Consider a typical business interface with one business method, `defineMeaningOfLife()`. If, in addition to this method, you define a setter for injection such as `setEncylopedia()`, then you are mandating that all implementations must use or at least be aware of the encyclopedia dependency. You do not need to define this setter at all—any decent IoC container, Spring included, can work with the component in terms of the business interface but still provide the dependencies of the implementing class. An example of this may clarify this matter slightly. Consider the business interface in Listing 4-6.

Listing 4-6. *The Oracle Interface*

```
package com.apress.prospring.ch4;

public interface Oracle {

    public String defineMeaningOfLife();
}
```

Notice that the business interface does not define any setters for dependency injection. This interface could be implemented as shown in Listing 4-7.

Listing 4-7. *Implementing the Oracle Interface*

```
package com.apress.prospring.ch4;

public class BookwormOracle implements Oracle {

    private Encyclopedia enc;
```

```
    public void setEncyclopedia(Encyclopedia enc) {
        this.enc = enc;
    }

    public String defineMeaningOfLife() {
        return "Encyclopedias are a waste of money - use the Internet";
    }

}
```

As you can see, the `BookwormOracle` class not only implements the `Oracle` interface but also defines the setter for Dependency Injection. Spring is more than comfortable dealing with a structure like this—there is absolutely no need to define the dependencies on the business interface. The ability to use interfaces to define dependencies is an often-touted benefit of setter injection, but in actuality, you should strive to keep setters used solely for injection out of your business interfaces. Unless you are absolutely sure that all implementations of a particular business interface require a particular dependency, let each implementation class define its own dependencies and keep the business interface for business methods.

Although you shouldn't always place setters for dependencies in a business interface, placing setters and getters for configuration parameters in the business interface is a good idea and makes setter injection a valuable tool. We consider configuration parameters to be a special case for dependencies. Certainly your components depend on the configuration data, but configuration data is significantly different from the types of dependency you have seen so far. We will discuss the differences shortly, but for now, consider the business interface shown in Listing 4-8.

Listing 4-8. *The NewsletterSender Interface*

```
package com.apress.prospring.ch4;

public interface NewsletterSender {

    public void setSmtpServer(String smtpServer);
    public String getSmtpServer();

    public void setFromAddress(String fromAddress);
    public String getFromAddress();

    public void send();
}
```

The `NewsletterSender` interface is implemented by classes that send a set of newsletters via e-mail. The `send()` method is the only business method, but notice that we have defined two JavaBean properties on the interface. Why are we doing this, when we just said that you shouldn't define dependencies in the business interface? The reason is that these values, the SMTP server address and the address the e-mails are sent from, are not dependencies in the practical sense; rather, they are configuration details that affect how all implementations of the `NewsletterSender` interface function. Spring's Dependency Injection capabilities form the

ideal solution to the external configuration of application components, not for dependency provision but as a mechanism for externalizing component configuration settings. The question here then is: What is the difference between a configuration parameter and any other kind of dependency? In most cases, you can clearly see whether a dependency should be classed as a configuration parameter, but if you are not sure, look for the following three characteristics that point to a configuration parameter:

1. **Configuration parameters are passive.** In the `NewsletterSender` example shown in Listing 4-8, the SMTP server parameter is an example of a passive dependency. Passive dependencies are not used directly to perform an action; instead, they are used internally or by another dependency to perform their actions. In the `MessageRenderer` example from Chapter 2, the `MessageProvider` dependency was not passive—it performed a function that was necessary for the `MessageRenderer` to complete its task.

2. **Configuration parameters are usually information, not other components.** By this we mean that a configuration parameter is usually some piece of information that a component needs to complete its work. Clearly the SMTP server is a piece of information required by the `NewsletterSender`, but the `MessageProvider` is really another component that the `MessageRenderer` needs to function correctly.

3. **Configuration parameters are usually simple values or collections of simple values.** This is really a by-product of points 1 and 2, but configuration parameters are usually simple values. In Java this means they are a primitive (or the corresponding wrapper class) or a `String` or collections of these values. Simple values are generally passive. This means you can't do much with a `String` other than manipulate the data it represents; and you almost always use these values for information purposes—for example, an `int` value that represents the port number that a network socket should listen on, or a `String` that represents the SMTP server through which an e-mail program should send messages.

When considering whether to define configuration options in the business interface, also consider whether the configuration parameter is applicable to all implementations of the business interface or just one. For instance, in the case of implementations of `NewsletterSender`, it is obvious that all implementations need to know which SMTP server to use when sending e-mails. However, we would probably choose to leave the configuration option that flags whether to send secure e-mail off the business interface, because not all e-mail APIs are capable of this and it is correct to assume that many implementations will not take security into consideration at all.

■ **Note** Recall that in Chapter 2, we chose to define the dependencies in the business purposes. This was for illustration purposes and should not be treated in any way as best practice.

Setter injection also allows you to swap dependencies for a different implementation on the fly without creating a new instance of the parent component. Currently Spring doesn't support this feature, but as soon as Spring is JMX aware, this feature will present itself. Perhaps the biggest benefit of setter injection is that it is the least intrusive of the Injection mechanisms.

If you are defining constructors for injection on a class that would otherwise just have the default constructor, then you are affecting all code that uses that class in a non-IoC environment. Extra setters that are defined on a class for IoC purposes do not affect the ability of other classes to interact with it.

In general, setter-based injection is the best choice, because it has the least effect on your code's usability in non-IoC settings. Constructor injection is a good choice when you want to ensure that dependencies are being passed to a component, but bear in mind that many containers provide their own mechanism for doing this with setter injection. Most of the code in the sample application uses setter injection, although there are a few examples of constructor injection.

Inversion of Control in Spring

As we mentioned earlier, Inversion of Control is a big part of what Spring does, and the core of Spring's implementation is based on Dependency Injection, although Dependency Lookup features are provided as well. When Spring provides collaborators to a dependent object automatically, it does so using Dependency Injection. In a Spring-based application, it is always preferable to use Dependency Injection to pass collaborators to dependent objects rather than have the dependent objects obtain the collaborators via Lookup. Although Dependency Injection is the preferred mechanism for wiring together collaborators and dependent objects, you need Dependency Lookup to access the dependent objects. In many environments, Spring cannot automatically wire up **all** of your application components using Dependency Injection and you must use Dependency Lookup to access the initial set of components. When you are building web applications using Spring's MVC support, Spring can avoid this by gluing your entire application together automatically. Wherever it is possible to use Dependency Injection with Spring, you should do so; otherwise you can fall back on the Dependency Lookup capabilities. You will see examples of both in action during the course of this chapter, and we are sure to point them out when they first arise.

An interesting feature of Spring's IoC container is that it has the ability to act as an adaptor between its own Dependency Injection container and external Dependency Lookup containers. We look at this in more detail in Chapter 5.

Spring supports both constructor and setter injection and bolsters the standard IoC feature set with a whole host of useful additions to make your life easier.

The rest of this chapter introduces the basics of Spring's DI container complete with plenty of examples.

Dependency Injection with Spring

Spring's support for Dependency Injection is comprehensive and, as you will see in Chapter 5, goes beyond the standard IoC feature set we have discussed so far. The rest of this chapter addresses the basics of Spring's Dependency Injection container, looking at both setter and constructor injection, along with a detailed look at how Dependency Injection is configured in Spring.

Beans and BeanFactories

The core of Spring's Dependency Injection container is the `BeanFactory`. A `BeanFactory` is responsible for managing components and their dependencies. In Spring, the term **bean** is used to refer to any component managed by the container. Typically your beans adhere, at some level, to the JavaBeans specification, but this is not required, especially if you plan to use Constructor Injection to wire your beans together.

Your application interacts with the Spring DI container via the `BeanFactory` interface. At some point, your application must create an instance of a class that implements the `BeanFactory` interface and configure it with bean and dependency information. After this is complete, your application can access the beans via the `BeanFactory` and get on with its processing. In some cases, all of this setup is handled automatically, but in many cases, you need to code the setup yourself. All of the examples in this chapter require manual setup of the `BeanFactory` implementation.

Although a `BeanFactory` can be configured programmatically, it is more common to see it configured externally using some kind of configuration file. Internally, bean configuration is represented by instances of classes that implement the `BeanDefinition` interface. The bean configuration stores not only information about a bean itself, but also about the beans that it depends on. For any `BeanFactory` class that also implements the `BeanDefinitionRegistry` interface, you can read the `BeanDefinition` data from a configuration file, using either `PropertiesBeanDefinitionReader` or `XmlBeanDefinitionReader`. The two main `BeanFactory` implementations that come with Spring implement `BeanDefinitionRegistry`.

So you can identify your beans within the `BeanFactory`, each bean is assigned a name. Each bean has at least one name but can have any number. Any names after the first are considered aliases for the same bean. You use bean names to retrieve a bean from the `BeanFactory` and also to establish dependency relationships—that is, bean X depends on bean Y.

BeanFactory Implementations

The description of the `BeanFactory` might make using it seem overly complex, but in practice, this is not the case. In fact, we discussed all of the concepts in the previous section and in the simple example in Chapter 2. Listing 4-9 shows the code from Chapter 2.

Listing 4-9. *Using the BeanFactory*

```
package com.apress.prospring.ch2;

import java.io.FileInputStream;
import java.util.Properties;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
```

```
public class HelloWorldSpringWithDI {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();

        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

        // create a definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
                factory);

        // load the configuration options
        Properties props = new Properties();
        props.load(new FileInputStream("./ch2/src/conf/beans.properties"));

        rdr.registerBeanDefinitions(props);

        return factory;
    }
}
```

In this example, you can see that we are using the `DefaultListableBeanFactory`—one of the two main `BeanFactory` implementations supplied with Spring—and that we are reading in the `BeanDefinition` information from a properties file using the `PropertiesBeanDefinitionReader`. Once the `BeanFactory` implementations is created and configured, we retrieve the `MessageRenderer` bean using its name, `renderer`, which is configured in the properties file.

In addition to the `PropertiesBeanDefinitionReader`, Spring also provides `XmlBeanDefinitionReader`, which allows you to manage your bean configuration using XML rather than properties. Although properties are ideal for small, simple applications, they can quickly become cumbersome when you are dealing with a large number of beans. For this reason, it is preferable to use the XML configuration format for all but the most trivial of applications. This leads nicely to a discussion of the second of the two main `BeanFactory` implementations: `XmlBeanFactory`.

The `XmlBeanFactory` is derived from `DefaultListableBeanFactory` and simply extends it to perform automatic configuration using the `XmlBeanDefinitionReader`. So rather than create some code like this:

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.FileSystemResource;

public class XmlConfig {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);
        rdr.loadBeanDefinitions(new FileSystemResource("ch4/src/conf/beans.xml"));
        Oracle oracle = (Oracle)factory.getBean("oracle");
    }
}
```

you can do this instead:

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class XmlConfigWithBeanFactory {

    public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "ch4/src/conf/beans.xml"));
        Oracle oracle = (Oracle)factory.getBean("oracle");
    }
}
```

For the rest of this book, including the sample application, we will be using the XML configuration format exclusively. You are free to investigate the properties format yourself—you will find plenty of examples throughout the Spring codebase.

Of course, you are free to define your own `BeanFactory` implementations, although be aware that doing so is quite involved; you need to implement a lot more interfaces than just `BeanFactory` to get the same level of functionality you have with the supplied `BeanFactory` implementations. If all you want to do is define a new configuration mechanism, then create your definition reader and wrap this in a simple `BeanFactory` implementation derived from `DefaultListableBeanFactory`. This is the approach the `XmlBeanFactory` class takes; check the Spring code for more details.

Configuring the BeanFactory

The key to getting set up with any Spring-based application is creating the `BeanFactory` configuration file for your application. A basic configuration without any bean definitions looks like this:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
</beans>
```

Each bean is then defined using a `<bean>` tag under the root of the `<beans>` tag. The `<bean>` tag has two required attributes: `id` and `class`. The `id` attribute is used to give the bean its default name and the `class` attribute specifies the type of the bean. Listing 4-10 returns to the Hello World example from Chapter 2 to show how the two beans, `renderer` and `provider`, are defined within the configuration file.

Listing 4-10. *Configuring the Hello World Example with XML*

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="renderer"
          class="com.apress.prospring.ch2.StandardOutMessageRenderer"/>
    <bean id="provider"
          class="com.apress.prospring.ch2.HelloWorldMessageProvider"/>
</beans>
```

We can modify the code from Chapter 2 to read this configuration using the `XmlBeanFactory`. Listing 4-11 shows how.

Listing 4-11. *Reading the Hello World Configuration from XML*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

import com.apress.prospring.ch2.MessageProvider;
import com.apress.prospring.ch2.MessageRenderer;

public class HelloWorldXml {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();

        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        MessageProvider mp = (MessageProvider) factory.getBean("provider");
```

```
        mr.setMessageProvider(mp);
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "ch4/src/conf/beans.xml"));

        return factory;
    }
}
```

Notice that there is no difference between the `main()` in Listing 4-11 and the `main()` method in Listing 2-10 in Chapter 2, but here there is substantially less code in the `getBeanFactory()` method. The interesting point to note here is that the code in the `main()` method didn't change. This is because it was working with the `BeanFactory` interface, not some subinterface or class. Although you may need to work with the specific `BeanFactory` type during configuration, you do not need to during the rest of your application when all you are doing is locating beans using `getBean()`. This is a good pattern to follow and you should avoid coupling your application too closely to any particular `BeanFactory` implementation.

Note that there is a problem with the code in Listing 4-11, a problem we encountered and fixed in Chapter 2—the application still has to pass the provider bean to the reference bean to satisfy its dependency. In Chapter 2, we modified the configuration; Spring did this for us using setter injection. We can, of course, also do this using the XML configuration provider.

Using Setter Injection

To configure setter injection using the XML provider, you need to specify `<property>` tags under the `<bean>` tag for each `<property>` into which you wish to inject a dependency. For example, to assign the provider bean to the `messageProvider` property of the `renderer` bean, we simply change the `<bean>` tag for the `renderer` bean as follows:

```
<bean id="renderer"
      class="com.apress.prospring.ch2.StandardOutMessageRenderer">
    <property name="messageProvider">
        <ref local="provider"/>
    </property>
</bean>
```

From this code, you can see that we are assigning the `provider` bean to the `messageProvider` property. We use the `<ref>` tag to assign a bean reference to a property (discussed in more detail shortly). Now we can remove the unnecessary property assignments from the Hello World example, as shown in Listing 4-12.

Listing 4-12. *Configuring Dependency Injection with XML*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

import com.apress.prospring.ch2.MessageRenderer;

public class HelloWorldXmlWithDI {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "ch4/src/conf/beans.xml"));

        return factory;
    }
}
```

This example makes full use of Spring's Dependency Injection capabilities and is fully configured using the XML format.

Using Constructor Injection

In the previous example, the `MessageProvider` implementation, `HelloWorldMessageProvider`, returned the same hard-coded message for each call of the `getMessage()` method. In the Spring configuration file, you can easily create a configurable `MessageProvider` that allows the message to be defined externally, as shown in Listing 4-13.

Listing 4-13. *The ConfigurableMessageProvider Class*

```
package com.apress.prospring.ch4;

import com.apress.prospring.ch2.MessageProvider;

public class ConfigurableMessageProvider implements MessageProvider {
```

```
    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```

As you can see, it is impossible to create an instance of `ConfigurableMessageProvider` without providing a value for the message (unless you supply `null`). This is exactly what we want, and this class is ideally suited for use with Constructor Injection. Listing 4-14 shows how you can redefine the `provider` bean definition to create an instance of `ConfigurableMessageProvider`, injecting the message using Constructor Injection.

Listing 4-14. *Using Constructor Injection*

```
<bean id="provider" class="com.apress.prospring.ch4.ConfigurableMessageProvider">
    <constructor-arg>
        <value>This is a configurable message</value>
    </constructor-arg>
</bean>
```

In this code, instead of using a `<property>` tag, we used a `<constructor-arg>` tag. Because we are not passing in another bean this time, just a `String` literal, we use the `<value>` tag instead of the `<ref>` to specify the value for the constructor argument.

When you have more than one constructor argument or your class has more than one constructor, you need to give each `<constructor-arg>` tag an `index` attribute to specify the index of the argument, starting at 0, in the constructor signature. It is always best to use the `index` attribute whenever you are dealing with constructors that have multiple arguments to avoid confusion between the parameters and ensure that Spring picks the correct constructor.

Avoiding Constructor Confusion

In some cases, Spring finds it impossible to tell which constructor you want it to use for constructor injection. This usually arises when you have two constructors with the same number of arguments and the types used in the arguments are represented in exactly the same way. Consider the code in Listing 4-15.

Listing 4-15. *Constructor Confusion*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class ConstructorConfusion {

    private String someValue;

    public ConstructorConfusion(String someValue) {
        System.out.println("ConstructorConfusion(String) called");
        this.someValue = someValue;
    }

    public ConstructorConfusion(int someValue) {
        System.out.println("ConstructorConfusion(int) called");
        this.someValue = "Number: " + Integer.toString(someValue);
    }

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"));

        ConstructorConfusion cc = (ConstructorConfusion)
                                        factory.getBean("constructorConfusion");
        System.out.println(cc);
    }

    public String toString() {
        return someValue;
    }
}
```

Here, you can clearly see what this code does—it simply retrieves a bean of type `ConstructorConfusion` from the `BeanFactory` and writes the value to `stdout`. Now look at the configuration code in Listing 4-16.

Listing 4-16. *Confused Constructors*

```
<bean id="constructorConfusion"
      class="com.apress.prospring.ch4.ConstructorConfusion">
    <constructor-arg>
        <value>90</value>
    </constructor-arg>
</bean>
```

Which of the constructors is called in this case? Running the example yields the following output:

```
ConstructorConfusion(String) called
90
```

This shows that the constructor with the `String` argument was called. This is not the desired effect, since we want to prefix any integer values passed in using constructor injection with `Number:`, as shown in the `int` constructor. To get around this, we need to make a small modification to the configuration, shown in Listing 4-17.

Listing 4-17. *Overcoming Constructor Confusion*

```
<bean id="constructorConfusion"
      class="com.apress.prospring.ch4.ConstructorConfusion">
    <constructor-arg type="int">
        <value>90</value>
    </constructor-arg>
</bean>
```

Notice now that the `<constructor-arg>` tag has an additional attribute, `type`, that specifies the type of argument that Spring should look for. Running the example again with the corrected configuration yields the correct output:

```
ConstructorConfusion(int) called
Number: 90
```

Injection Parameters

In the two previous examples, you saw how to inject other components and values into a bean using both setter injection and constructor injection. Spring supports a myriad of options for injection parameters, allowing you to inject not only other components and simple values, but also Java Collections, externally defined properties, and even beans in another factory. You can use all of these injection parameter types for both setter injection and constructor injection by using the corresponding tag under the `<property>` and `<constructor-args>` tags, respectively.

Injecting Simple Values

Injecting simple values into your beans is easy. To do so, simply specify the value in the configuration tag, wrapped inside a `<value>` tag. By default, the `<value>` tag can not only read `String` values, but it can also convert these values to any primitive or primitive wrapper class. Listing 4-18 shows a simple bean that has a variety of properties exposed for injection.

Listing 4-18. *Injecting Simple Values*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class InjectSimple {

    private String name;

    private int age;

    private float height;

    private boolean isProgrammer;

    private Long ageInSeconds;

    public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"));
        InjectSimple simple = (InjectSimple)factory.getBean("injectSimple");
        System.out.println(simple);
    }

    public void setAgeInSeconds(Long ageInSeconds) {
        this.ageInSeconds = ageInSeconds;
    }

    public void setIsProgrammer(boolean isProgrammer) {
        this.isProgrammer = isProgrammer;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }

    public void setHeight(float height) {
        this.height = height;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return    "Name :" + name + "\n"
                + "Age:" + age + "\n"
                + "Age in Seconds: " + ageInSeconds + "\n"
                + "Height: " + height + "\n"
                + "Is Programmer?: " + isProgrammer;
    }
}
```

In addition to the properties, the `InjectSimple` class also defines the `main()` method that creates an `XmlBeanFactory` and then retrieves an `InjectSimple` bean from Spring. The property values of this bean are then written to `stdout`. The configuration for this bean is shown in Listing 4-19.

Listing 4-19. *Configuring Simple Value Injection*

```
<bean id="injectSimple" class="com.apress.prospring.ch4.InjectSimple">
        <property name="name">
            <value>John Smith</value>
        </property>
        <property name="age">
            <value>35</value>
        </property>
        <property name="height">
            <value>1.78</value>
        </property>
        <property name="isProgrammer">
            <value>true</value>
        </property>
        <property name="ageInSeconds">
                <value>1103760000</value>
        </property>
    </bean>
```

You can see from Listings 4-18 and 4-19 that is it is possible to define properties on your bean that accept `String` values, primitive values, or primitive wrapper values and then inject

values for these properties using the <value> tag. Here is the output created by running this example as expected:

```
Name: John Smith
Age: 35
Age in Seconds: 1103760000
Height: 1.78
Is Programmer?: true
```

In Chapter 5, you will see how to expand the range of types that can be injected using the <value> tag.

Injecting Beans in the Same Factory

As you have already seen, it is possible to inject one bean into another using the <ref> tag. Listing 4-20 shows a class that exposes a setter to allow a bean to be injected.

Listing 4-20. *Injecting Beans*

```
package com.apress.prospring.ch4;

public class InjectRef {

    private Oracle oracle;

    public void setOracle(Oracle oracle) {
        this.oracle = oracle;
    }
}
```

To configure Spring to inject one bean into another, you first need to configure two beans: one to be injected and one to be the target of the injection. Once this is done, you simply configure the injection using the <ref> tag on the target bean. Remember that <ref> must come under either <property> or <constructor-arg> depending on whether you are using setter or constructor injection. Listing 4-21 shows an example of this configuration.

Listing 4-21. *Configuring Bean Injection*

```
<bean id="injectRef" class="com.apress.prospring.ch4.InjectRef">
    <property name="oracle">
        <ref local="oracle"/>
    </property>
</bean>
<bean id="oracle" class="com.apress.prospring.ch4.BookwormOracle"/>
```

An important point to note is that the type being injected does not have to be the exact type defined on the target; the types just need to be compatible. Compatible means that if the declared type on the target is an interface, then the injected type must implement this interface. If the declared type is a class, then the injected type must either be the same type or a subtype. In this example, the InjectRef class defines the setOracle() method to receive an

instance of `Oracle`, which is an interface, and the injected type is `BookwormOracle`, a class that implements `Oracle`. This is a point that causes confusion for some developers, but it is really quite simple. Injection is subject to the same typing rules as any Java code, so as long as you are familiar with how Java typing works, then understanding typing in injection is easy.

In the previous example, the `id` of the bean to inject was specified using the `local` attribute of the `<ref>` tag. As you will see later, in the section titled "Understanding Bean Naming," you can give a bean more than one name so that you can refer to it using a variety of aliases. When you use the `local` attribute, it means that the `<ref>` tag only ever looks at the bean's `id` and never at any of its aliases. To inject a bean by any name, use the `bean` attribute of the `<ref>` tag instead of the `local` attribute. Listing 4-22 shows an alternative configuration for the previous example using an alternative name for the injected bean.

Listing 4-22. *Injecting Using Bean Aliases*

```
<bean id="injectRef" class="com.apress.prospring.ch4.InjectRef">
    <property name="oracle">
        <ref bean="wiseworm"/>
    </property>
</bean>
<bean id="oracle"
      name="wiseworm"
      class="com.apress.prospring.ch4.BookwormOracle"/>
```

In this example, the `oracle` bean is given an alias using the `name` attribute, and then it is injected into the `injectRef` bean by using this alias in conjunction with the `bean` attribute of the `<ref>` tag. Don't worry too much about the naming semantics at this point—we discuss this in much more detail later in the chapter.

Injection and BeanFactory Nesting

So far the beans we have been injecting have been located in the same bean factory as the beans they are injected into. However, Spring supports a hierarchical structure for `BeanFactories` so that one factory is considered the parent of another. By allowing `BeanFactories` to be nested, Spring allows you to split your configuration into different files—a godsend on larger projects with lots of beans.

When nesting `BeanFactories`, Spring allows beans in what is considered the child factory to reference beans in the parent factory. The only drawback is that this can only be done in configuration. It is impossible to call `getBean()` on the child `BeanFactory` to access a bean in the parent `BeanFactory`.

`BeanFactory` nesting using the `XmlBeanFactory` is very simple to get a grip on. To nest one `XmlBeanFactory` inside another, simply pass the parent `XmlBeanFactory` as a constructor argument to the child `XmlBeanFactory`. This is shown in Listing 4-23.

Listing 4-23. *Nesting XmlBeanFactories*

```
BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/parent.xml"));
        BeanFactory child = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"), parent);
```

Inside the configuration file for the child `BeanFactory`, referencing a bean in the parent `BeanFactory` works exactly like referencing a bean in the child `BeanFactory`, unless you have a bean in the child `BeanFactory` that shares the same name. In that case, you simply replace the `bean` attribute of the `<ref>` tag with `parent` and you are on your way. Listing 4-24 shows a sample configuration file for the parent `BeanFactory`.

Listing 4-24. *Parent BeanFactory Configuration*

```
    <bean id="injectBean" class="java.lang.String">
           <constructor-arg>
               <value>Bean In Parent</value>
           </constructor-arg>
    </bean>
    <bean id="injectBeanParent" class="java.lang.String">
           <constructor-arg>
               <value>Bean In Parent</value>
           </constructor-arg>
    </bean>
```

As you can see, this configuration simply defines two beans: `injectBean` and `injectBeanParent`. Both are `String` objects with the value `Bean In Parent`. Listing 4-25 shows a sample configuration for the child `BeanFactory`.

Listing 4-25. *Child BeanFactory Configuration*

```
<bean id="target1" class="com.apress.prospring.ch4.SimpleTarget">
        <property name="val">
            <ref bean="injectBeanParent"/>
        </property>
    </bean>

    <bean id="target2" class="com.apress.prospring.ch4.SimpleTarget">
        <property name="val">
            <ref bean="injectBean"/>
        </property>
    </bean>

    <bean id="target3" class="com.apress.prospring.ch4.SimpleTarget">
        <property name="val">
            <ref parent="injectBean"/>
        </property>
    </bean>
```

```
    <bean id="injectBean" class="java.lang.String">
           <constructor-arg>
               <value>Bean In Child</value>
           </constructor-arg>
    </bean>
```

Notice that we have defined four beans here. The `injectBean` in this listing is similar to the `injectBean` in the parent except that the `String` it represents has a different value, indicating that it is located in the child `BeanFactory`.

The `target1` bean is using the `bean` attribute of the `<ref>` tag to reference the bean named `injectBeanParent`. Because this bean only exists in the parent `BeanFactory`, `target1` receives a reference to that bean. There are two points of interest here. First, you can use the `bean` attribute to reference beans in both the child and parent `BeanFactories`. This makes it easy to reference the beans transparently, allowing you to move beans between configuration files as your application grows. The second point of interest is that you can't use the `local` attribute to refer to beans in the parent `BeanFactory`. The XML parser checks to see that the value of the `local` attribute exists as a valid element in the same file, preventing it from being used to reference beans in the parent factory.

The `target2` bean is using the `bean` attribute of the `<ref>` tag to reference the `injectBean`. Because that bean is defined in both `BeanFactories`, the `target2` bean receives a reference to the `injectBean` in its own `BeanFactory`.

The `target3` bean is using the `parent` attribute of the `<ref>` tag to reference the `injectBean` directly in the parent `BeanFactory`. Because `target3` is using the `parent` attribute of the `<ref>` tag, the `injectBean` declared in the child `BeanFactory` is ignored completely.

The code in Listing 4-26 demonstrates the semantics discussed here by retrieving each of the three `target`*X* beans from the child `BeanFactory` and outputting the value of the `val` property in each case.

Listing 4-26. *The HierarchicalBeanFactoryUsage Class*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class HierarchicalBeanFactoryUsage {

    public static void main(String[] args) {
        BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/parent.xml"));
        BeanFactory child = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"), parent);

        SimpleTarget target1 = (SimpleTarget) child.getBean("target1");
        SimpleTarget target2 = (SimpleTarget) child.getBean("target2");
        SimpleTarget target3 = (SimpleTarget) child.getBean("target3");
```

```
        System.out.println(target1.getVal());
        System.out.println(target2.getVal());
        System.out.println(target3.getVal());
    }
}
```

Here is the output from running this example:

```
Bean In Parent
Bean In Child
Bean In Parent
```

As expected, the `target1` and `target3` beans both get a reference to beans in the parent `BeanFactory`, whereas the `target2` bean gets a reference to a bean in the child `BeanFactory`.

Using Collections for Injection

Often your beans need access to collections of objects rather than just individual beans or values. Therefore, it should come as no surprise that Spring allows you to inject a collection of objects into one of your beans. Using the collection is simple: you choose either `<list>`, `<map>`, `<set>`, or `<props>` to represent a `List`, `Map`, `Set`, or `Properties` instance, and then you pass in the individual items just as you would with any other injection. The `<props>` tag only allows for `String`s to be passed in as the value because the `Properties` class only allows for property values to be `String`s. When using `<list>`, `<map>`, or `<set>`, you can use any tag you use when injecting into a property, even another collection tag. This allows you to pass in a `List` of `Map`s, a `Map` of `Set`s, or even a `List` of `Map`s of `Set`s of `List`s! Listing 4-27 shows a class that can have all four collection types injected into it.

Listing 4-27. *Collection Injection*

```
package com.apress.prospring.ch4;

import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class CollectionInjection {

    private Map map;

    private Properties props;

    private Set set;
```

```
    private List list;

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(
                              new FileSystemResource("./ch4/src/conf/beans.xml"));

        CollectionInjection instance = (CollectionInjection)

        factory.getBean("injectCollection");
        instance.displayInfo();
    }

    public void setList(List list) {
        this.list = list;
    }

    public void setSet(Set set) {
        this.set = set;
    }

    public void setMap(Map map) {
        this.map = map;
    }

    public void setProps(Properties props) {
        this.props = props;
    }

    public void displayInfo() {

        // display the Map
        Iterator i = map.keySet().iterator();

        System.out.println("Map contents:\n");
        while (i.hasNext()) {
            Object key = i.next();
            System.out.println("Key: " + key + " - Value: " + map.get(key));
        }

        // display the properties
        i = props.keySet().iterator();
        System.out.println("\nProperties contents:\n");
        while (i.hasNext()) {
            String key = i.next().toString();
            System.out.println("Key: " + key + " - Value: "
                    + props.getProperty(key));
        }
```

```
        // display the set
        i = set.iterator();
        System.out.println("\nSet contents:\n");
        while (i.hasNext()) {
            System.out.println("Value: " + i.next());
        }

        // display the list
        i = list.iterator();
        System.out.println("\nList contents:\n");
        while (i.hasNext()) {
            System.out.println("Value: " + i.next());
        }
    }
}
```

That is quite a lot of code, but it actually does very little. The `main()` method retrieves a `CollectionInjection` bean from Spring and then calls the `displayInfo()` method. This method just outputs the contents of the `List`, `Map`, `Properties`, and `Set` instances that will be injected from Spring. In Listing 4-28, you can see the configuration required to inject values for each of the properties on the `CollectionInjection` class.

Listing 4-28. *Configuring Collection Injection*

```
<bean id="injectCollection" class="com.apress.prospring.ch4.CollectionInjection">
        <property name="map">
            <map>
                <entry key="someValue">
                    <value>Hello World!</value>
                </entry>
                <entry key="someBean">
                    <ref local="oracle"/>
                </entry>
            </map>
        </property>
        <property name="props">
            <props>
                <prop key="firstName">
                    Rob
                </prop>
                <prop key="secondName">
                    Harrop
                </prop>
            </props>
        </property>
```

```
        <property name="set">
            <set>
                <value>Hello World!</value>
                <ref local="oracle"/>
            </set>
        </property>
        <property name="list">
            <list>
                <value>Hello World!</value>
                <ref local="oracle"/>
            </list>
        </property>
    </bean>
```

In this code, you can see that we have injected value into all four setters exposed on the `ConstructorInjection` class. For the `map` property, we have injected a `Map` instance using the `<map>` tag. Notice that each entry is specified using an `<entry>` tag and each has a `String` key and then an entry value. This entry value can be any value you can inject into a property separately; this example shows the use of the `<value>` and `<ref>` tags to add a `String` value and a bean reference to the `Map`. For the `props` property, we use the `<props>` tag to create an instance of `java.util.Properties` and populate it using `<prop>` tags. Notice that although the `<prop>` tag is keyed in a similar manner to the `<entry>` tag, you can only specify a `String` value for each property that goes in the `Properties` instance.

Both the `<list>` and `<set>` tags work in exactly the same way: you specify each element using any of the individual value tags such as `<value>` and `<ref>` that are used to inject a single value into a property. In Listing 4-28, you can see that we have added a `String` value and a bean reference to both the `List` and the `Set`.

Here is the output generated by Listing 4-28. As expected, it simply lists the elements added to the collections in the configuration file.

```
Map contents:

Key: someValue - Value: Hello World!
Key: someBean - Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c

Properties contents:

Key: secondName - Value: Harrop
Key: firstName - Value: Rob

Set contents:

Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c
Value: Hello World!

List contents:

Value: Hello World!
Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c
```

Remember, with the <list>, <map>, and <set> elements, you can employ any of the tags used to set the value of noncollection properties to specify the value of one of the entries in the collection. This is quite a powerful concept because you are not limited just to injecting collections of primitive values, you can also inject collections of beans or other collections.

Using this functionality, it is much easier to modularize your application and provide different, user-selectable implementations of key pieces of application logic. Consider a system that allows corporate staff to create, proofread, and order their personalized business stationery online. In this system, the finished artwork for each order is sent to the appropriate printer when it is ready for production. The only complication is that some printers want to receive the artwork via e-mail, some via FTP, and still more using Secure Copy Protocol (SCP). Using Spring's collection injection, you can create a standard interface for this functionality, as shown in Listing 4-29.

Listing 4-29. *The ArtworkSender Interface*

```
package com.apress.prospring.ch4;

public interface ArtworkSender {

    public void sendArtwork(String artworkPath, Recipient recipient);

    public String getFriendlyName();

    public String getShortName();
}
```

From this interface, you can create multiple implementations, each of which is capable of describing itself to a human, such as the ones shown in Listing 4-30.

Listing 4-30. *The FtpArtworkSender Class*

```
package com.apress.prospring.ch4;

public class FtpArtworkSender implements ArtworkSender {

    public void sendArtwork(String artworkPath, Recipient recipient) {
        // ftp logic here...
    }

    public String getFriendlyName() {
        return "File Transfer Protocol";
    }

    public String getShortName() {
        return "ftp";
    }
}
```

With the implementations in place, you simply pass a `List` to your `ArtworkManager` class and you are on your way. Using the `getFriendlyName()` method, you can display a list of delivery options for the system administrator to choose from when you are configuring each stationery template. In addition, your application can remain fully decoupled from the individual implementations if you just code to the `ArtworkSender` interface.

Understanding Bean Naming

Spring supports quite a complex bean naming structure that allows you the flexibility to handle many different situations. Every bean must have at least one name that is unique within the containing `BeanFactory`. Spring follows a simple resolution process to determine what name is used for the bean. If you give the `<bean>` tag an `id` attribute, then the value of that attribute is used as the name. If no `id` attribute is specified, Spring looks for a `name` attribute and, if one is defined, it uses the first name defined in the `name` attribute. (We say the first name because it is possible to define multiple names within the `name` attribute; this is covered in more detail shortly.) If neither the `id` nor the `name` attribute is specified, Spring uses the bean's class name as the name, provided, of course, that no other bean is using the same name. Listing 4-31 shows a sample configuration that uses all three naming schemes.

Listing 4-31. *Bean Naming*

```
<bean id="string1" class="java.lang.String"/>
<bean name="string2" class="java.lang.String"/>
<bean class="java.lang.String"/>
```

Each of these approaches is equally valid from a technical point of view, but which is the best choice for your application? To start with, avoid using the automatic name by class behavior. This doesn't allow you much flexibility to define multiple beans of the same type, and it is much better to define your own names. That way, if Spring changes the default behavior in the future, your application continues to work. When choosing whether to use `id` or `name`, always use `id` to specify the bean's default name. The `id` attribute in the XML is declared as an XML identity in the DTD for the Spring configuration file. This means that not only can the XML parser perform validation on your file, but any good XML editor should be able to do the same, thus reducing the number of errors as a result of mistyped bean names. Essentially, this allows your XML editor to validate that the bean you are referencing in the `local` attribute of a `<ref>` tag actually exists.

The only drawback of using the `id` attribute is that you are limited to characters that are allowed within XML element IDs. If you find that you cannot use a character you want in your name, then you can specify that name using the `name` attribute, which does not have to adhere to the XML naming rules. That said, you should still consider giving your bean a name using `id`, and then you can define the desirable name using name aliasing as discussed in the next section.

Bean Name Aliasing

Spring allows a bean to have more than one name. You can achieve this by specifying a comma- or semicolon-separated list of names in the `name` attribute of the bean's `<bean>` tag. You can do this in place of, or in conjunction with, using the `id` attribute. Listing 4-32 shows a simple `<bean>` configuration that defines multiple names for a single bean.

Listing 4-32. *Configuring Multiple Bean Names*

```
<bean  id="name1" name="name2,name3,name4" class="java.lang.String"/>
```

As you can see, we have defined four names: one using the `id` attribute, and the other three as a comma-separated list in the `name` attribute. Listing 4-33 shows a sample Java routine that grabs the same bean from the `BeanFactory` four times using the different names and verifies that they are the same bean.

Listing 4-33. *Accessing Beans Using Aliases*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class BeanNameAliasing {

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(
                              new FileSystemResource("./ch4/src/conf/beans.xml"));

        String s1 = (String)factory.getBean("name1");
        String s2 = (String)factory.getBean("name2");
        String s3 = (String)factory.getBean("name3");
        String s4 = (String)factory.getBean("name4");

        System.out.println((s1 == s2));
        System.out.println((s2 == s3));
        System.out.println((s3 == s4));
    }
}
```

This code prints `true` three times to `stdout` for the configuration contained in Listing 4-32 verifying that the beans accessed using different names are in fact the same bean.

You can retrieve a list of the bean's aliases by calling `BeanFactory.getAliases(String)` and passing in any one of the bean's names. The number of names returned in the list is always one less than the total number of names assigned to the bean, because Spring considers one of the names to be the default. Which name is the default depends on how you configured the bean. If you specified a name using the `id` attribute, then that is always the default. If you did not use the `id` attribute, then the first name in the list passed to the `name` attribute may be used.

Bean name aliasing is a strange beast because it is not something you tend to use when you are building a new application. If you are going to have many other beans inject another bean, then they may as well use the same name to access that bean. However, as your application goes into production and maintenance work gets carried out, modifications made, and so on, bean name aliasing becomes more useful.

Consider the following scenario: you have an application in which 50 different beans, configured using Spring, all require an implementation of the `Foo` interface. Twenty-five of the

beans use the `StandardFoo` implementation with the bean name `standardFoo` and the other 25 use the `SuperFoo` implementation with the `superFoo` bean name. Six months after you put the application into production, you decide to move the first 25 beans to the `SuperFoo` implementation. To do this you have three options.

The first is to change the implementation class of the `standardFoo` bean to `SuperFoo`. The drawback of this approach is that you have two instances of the `SuperFoo` class lying around when you really only need one. In addition, you now have two beans to make changes to when the configuration changes.

The second option is to update the injection configuration for the 25 beans that are changing, which changes the beans' names from `standardFoo` to `superFoo`. This approach is not the most elegant way to proceed—you could perform a find and replace, but then rolling back your changes when management isn't happy means retrieving an old version of your configuration from your version control system.

The third, and most ideal, approach is to remove (or comment out) the definition for the `standardFoo` bean and make `standardFoo` an alias to the `superFoo`. This change requires minimal effort and restoring the system to its previous configuration is just as simple.

Bean Instantiation Modes

By default, all beans in Spring are singletons. This means that Spring maintains a single instance of the bean, all dependent objects use the same instance, and all calls to `BeanFactory.getBean()` return the same instance. We demonstrated this in the previous example shown in Listing 4-33, where we were able to use identity comparison (`==`) rather than the `equals()` comparison to check if the beans were the same.

The term **singleton** is used interchangeably in Java to refer to two distinct concepts: an object that has a single instance within the application, and the Singleton design pattern. We refer to the first concept as singleton and to the Singleton pattern as Singleton. The Singleton design pattern was popularized in the seminal *Design Patterns: Elements of Reusable Object Oriented Software* by Erich Gamma, et al. (Addison-Wesley, 1995). The problem arises when people confuse the need for singleton instances with the need to apply the Singleton pattern. Listing 4-34 shows a typical implementation of the Singleton pattern in Java.

Listing 4-34. *The Singleton Design Pattern*

```
package com.apress.prospring.ch4;

public class Singleton {

    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

This pattern achieves its goal of allowing you to maintain and access a single instance of a class throughout your application, but it does so at the expense of increased coupling. Your application code must always have explicit knowledge of the `Singleton` class in order to obtain the instance—completely removing the ability to code to interfaces. In reality, the Singleton pattern is actually two patterns in one. The first, and desired, pattern involves maintenance of a single instance of an object. The second, and less desirable, is a pattern for object lookup that completely removes the possibility of using interfaces. Using the Singleton pattern also makes it very difficult to swap out implementations arbitrarily, because most objects that require the `Singleton` instance access the Singleton object directly. This can cause all kinds of headaches when you are trying to unit test your application because you are unable to replace the Singleton with a mock for testing purposes.

Fortunately, with Spring you can take advantage of the singleton instantiation model without having to work around the Singleton design pattern. All beans in Spring are, by default, created as `Singleton` instances, and Spring uses the same instances to fulfill all requests for that bean. Of course, Spring is not just limited to use of the singleton instance; it can still create a new instance of the bean to satisfy every dependency and every call to `getBean()`. It does all of this without any impact on your application code, and for this reason, we like to refer to Spring as being **instantiation mode agnostic**. This is a very powerful concept. If you start off with an object that is a singleton, but then discover that it is not really suited to multithread access, you can change it to a non-singleton without affecting any of your application code.

■ **Note** Although changing the instantiation mode of your bean won't affect your application code, it does cause some problems if you rely on Spring's lifecycle interfaces. We cover this in more detail in Chapter 5.

Changing the instantiation mode from singleton to non-singleton is simple (see Listing 4-35).

Listing 4-35. *Non-Singleton Bean Configuration*

```
<bean id="nonSingleton" class="java.lang.String" singleton="false">
        <constructor-arg>
            <value>Rob Harrop</value>
        </constructor-arg>
    </bean>
```

As you can see, the only difference between this bean declaration and any of the declarations you have seen so far is that we set the singleton attribute to `false`. Listing 4-36 shows the effect this setting has on your application.

Listing 4-36. *Non-Singleton Beans in Action*

```
package com.apress.prospring.ch4;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class NonSingleton {

    public static void main(String[] args) {

        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"));

        String s1 = (String)factory.getBean("nonSingleton");
        String s2 = (String)factory.getBean("nonSingleton");

        System.out.println("Identity Equal?: " + (s1 ==s2));
        System.out.println("Value Equal:? " + s1.equals(s2));
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Running this example gives you the following output:

```
Identity Equal?: false
Value Equal:? true
Rob Harrop
Rob Harrop
```

You can see from this that although the values of the two `String` objects are clearly equal, the identities are not, despite the fact that both instances were retrieved using the same bean name.

Choosing an Instantiation Mode

In most scenarios, it is quite easy to see which instantiation mode is suitable. Typically we find that singleton is the default mode for our beans. In general, singletons should be used in the following scenarios:

- **Shared objects with no state:** When you have an object that maintains no state and has many dependent objects. Because you do not need synchronization if there is no state, you do not really need to create a new instance of the bean each time a dependent object needs to use it for some processing.
- **Shared object with read-only state:** This is similar to the previous point, but you have some read-only state. In this case, you still do not need synchronization, so creating an instance to satisfy each request for the bean is just adding additional overhead.

- **Shared object with shared state:** If you have a bean that has state that must be shared, then singleton is the ideal choice. In this case, ensure that your synchronization for state writes is as granular as possible.
- **High throughput objects with writable state:** If you have a bean that is used a great deal in your application, then you may find that keeping a singleton and synchronizing all write access to the bean state allows for better performance than constantly creating hundreds of instances of the bean. When using this approach, try to keep the synchronization as granular as possible without sacrificing consistency. You will find that this approach is particularly useful when your application creates a large number of instances over a long period of time, when your shared object has only a small amount of writable state, or when the instantiation of a new instance is expensive.

You should consider using non-singletons in the following scenarios:

- **Objects with writable state:** If you have a bean that has a lot of writable state, then you may find that the cost of synchronization is greater than the cost of creating a new instance to handle each request from a dependent object.
- **Objects with private state:** In some cases, your dependent objects need a bean that has private state so that they can conduct their processing separately from other objects that depend on that bean. In this case, singleton is clearly not suitable and you should use non-singleton.

The main benefit you gain from Spring's instantiation management is that your applications can immediately benefit from the lower memory usage associated with singletons, and with very little effort on your part. Then, if you find that singleton does not meet the needs of your application, it is a trivial task to modify your configuration to use non-singleton mode.

Resolving Dependencies

During normal operation, Spring is able to resolve dependencies by simply looking at your configuration file. In this way, Spring can ensure that each bean is configured in the correct order so that each bean has its dependencies correctly configured. If Spring did not perform this and just created the beans and configured them in any order, a bean could be created and configured before its dependencies. This is obviously not what you want and would cause all sorts of problems within your application.

Unfortunately, Spring is not aware of any dependencies that exist between beans in your code. For instance, take one bean, bean A, which obtains an instance of another bean, bean B, in the constructor via a call to `getBean()`. In this case, Spring is unaware that bean A depends on bean B and, as a result, it may instantiate bean A before bean B. You can provide Spring with additional information about your bean dependencies using the `depends-on` attribute of the `<bean>` tag. Listing 4-37 shows how the scenario for bean A and bean B would be configured.

Listing 4-37. *Manually Defining Dependencies*

```
<bean id="A" class="com.apress.prospring.ch4.BeanA" depends-on="b"/>
<bean id="B" class="com.apress.prospring.ch4.BeanB"/>
```

In this configuration, we are asserting that bean A depends on bean B. Spring takes this into consideration when instantiating the beans and ensures that bean B is created before bean A.

When developing your applications, avoid designing your applications to use this feature; instead, define your dependencies by means of setter and constructor injection contracts. However, if you are integrating Spring with legacy code, then you may find that the dependencies defined in the code require you to provide extra information to the Spring framework.

Auto-Wiring Your Beans

In all the examples so far, we have had to define explicitly, via the configuration file, how the individual beans are wired together. If you don't like having to wire all your components together, then you can have Spring attempt to do so automatically. By default, auto-wiring is disabled. To enable it, you specify which method of auto-wiring you wish to use using the `autowire` attribute of the bean you wish to auto-wire.

Spring supports four modes for auto-wiring: **`byName`**, **`byType`**, **`constructor`**, and **`autodetect`**. When using `byName` auto-wiring, Spring attempts to wire each property to a bean of the same name. So, if the target bean has a property named `foo` and a `foo` bean is defined in the `BeanFactory`, the `foo` bean is assigned to the `foo` property of the target.

When using `byType` auto-wiring, Spring attempts to wire each of the properties on the target bean automatically using a bean of the same type in the `BeanFactory`. So if you have a property of type `String` on the target bean and a bean of type `String` in the `BeanFactory`, then Spring wires the `String` bean to the target bean's `String` property. If you have more than one bean of the same type, in this case `String`, in the same `BeanFactory`, then Spring is unable to decide which one to use for the auto-wiring and throws an exception.

The `constructor` auto-wiring mode functions just like `byType` wiring, except that it uses constructors rather than setters to perform the injection. Spring attempts to match the greatest numbers of arguments it can in the constructor. So, if your bean has two constructors, one that accepts a `String` and one that accepts a `String` and an `Integer`, and you have both a `String` and an `Integer` bean in your `BeanFactory`, Spring uses the two-argument constructor.

The final mode, `autodetect`, instructs Spring to choose between `constructor` and `byType` modes automatically. If your bean has a default (no arguments) constructor, then Spring uses `byType`; otherwise, it uses `constructor`.

Listing 4-38 shows a simple configuration that auto-wires four beans of the same type using each of the different modes.

Listing 4-38. *Configuring Auto-Wiring*

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="foo" class="com.apress.prospring.ch4.autowiring.Foo"/>
    <bean id="bar" class="com.apress.prospring.ch4.autowiring.Bar"/>

    <bean id="targetByName" autowire="byName"
            class="com.apress.prospring.ch4.autowiring.Target"/>
    <bean id="targetByType" autowire="byType"
            class="com.apress.prospring.ch4.autowiring.Target"/>
    <bean id="targetConstructor" autowire="constructor"
            class="com.apress.prospring.ch4.autowiring.Target"/>
    <bean id="targetAutodetect" autowire="autodetect"
            class="com.apress.prospring.ch4.autowiring.Target"/>
</beans>
```

This configuration should look very familiar to you now. Notice that each of the `Target` beans has a different value for the `autowire` attribute. Listing 4-39 shows a simple Java application that retrieves each of the `Target` beans from the `BeanFactory`.

Listing 4-39. *Auto-Wiring Collaborators*

```
package com.apress.prospring.ch4.autowiring;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class Target {

    private Foo foo;
    private Foo foo2;

    private Bar bar;

    public Target() {

    }

    public Target(Foo foo) {
        System.out.println("Target(Foo) called");
    }
```

```
    public Target(Foo foo, Bar bar) {
        System.out.println("Target(Foo, Bar) called");
    }

    public void setFoo(Foo foo) {
        this.foo = foo;
        System.out.println("Property foo set");
    }

    public void setFoo2(Foo foo) {
        this.foo2 = foo;
        System.out.println("Property foo2 set");
    }

    public void setMyBarProperty(Bar bar) {
        this.bar = bar;
        System.out.println("Property myBarProperty set");
    }

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/autowiring.xml"));

        Target t = null;

        System.out.println("Using byName:\n");
        t = (Target) factory.getBean("targetByName");

        System.out.println("\nUsing byType:\n");
        t = (Target) factory.getBean("targetByType");

        System.out.println("\nUsing constructor:\n");
        t = (Target) factory.getBean("targetConstructor");

        System.out.println("\nUsing autodetect:\n");
        t = (Target) factory.getBean("targetAutodetect");

    }
}
```

In this code, you can see that the `Target` class has three constructors: a no argument constructor, a constructor that accepts a `Foo` instance, and a constructor that accepts a `Foo` and a `Bar` instance. In addition to these constructors, the `Target` bean has three properties: two of type `Foo` and one of type `Bar`. Each of these properties and constructors writes a message to `stdout` when it is called. The main method simply retrieves each of the `Target` beans declared in the `BeanFactory`, triggering the auto-wire process. Here is the output from running this example:

```
Using byName:
    Property foo set

Using byType:
Property foo set
Property foo2 set
Property myBarProperty set

Using constructor:
Target(Foo, Bar) called

Using autodetect:
Property foo set
Property foo2 set
Property myBarProperty set
```

From the output, you can see that when Spring uses `byName`, the only property that is set is the `foo` property, because this is the only property with a corresponding bean entry in the configuration file. When using `byType`, Spring sets the value of all three properties. The `foo` and `foo2` properties are set by the `foo` bean and the `myBarProperty` is set by the `bar` bean. When using `constructor`, Spring uses the two-argument constructor, because Spring can provide beans for both arguments and does not need to fall back to another constructor. In this case, `autodetect` functions just like `byType` because we defined a default constructor. If we had not done this, `autodetect` would have functioned just like `constructor`.

When to Use Auto-Wiring

In most cases, the answer to the question of whether you should use auto-wiring is definitely "No!" Auto-wiring can save you time in small applications, but in many cases, it leads to bad practices and is inflexible in large applications. Using `byName` seems like a good idea, but it may lead you to give your classes artificial property names so that you can take advantage of the auto-wiring functionality. The whole idea behind Spring is that you can create your classes how you like and have Spring work for you, not the other way around. You may be tempted to use `byType` until you realize that you can only have one bean for each type in your `BeanFactory`—a restriction that is problematic when you need to maintain beans with different configurations of the same type. The same argument applies to the use of `constructor` auto-wiring. This mode follows the same semantics as `byType` and to `autodetect`, which is simply `byType` and `constructor` bundled together.

In some cases, auto-wiring can save you time, but it does not really take that much extra effort to define your wiring explicitly, and you benefit from explicit semantics and full flexibility on property naming and on how many instances of the same type you manage. For any non-trivial application, steer clear of auto-wiring at all costs.

Checking Dependencies

When creating bean instances and wiring together dependencies, Spring does not, by default, check to see that every property on a bean has a value. In many cases, you do not need Spring

to perform this check, but if you have a bean that absolutely must have a value for all of its properties, then you can ask Spring to check this for you.

As pointed out in the documentation, this is not always effective, because you may provide a default value for some properties and you may just want to assert that only one particular property must also have a value; the dependency checking capabilities of Spring do not take these features into consideration. That said, it can be quite useful in certain circumstances to have Spring perform this check for you. In many cases, it allows you to remove checks from your code and have Spring perform them just once at startup.

Besides the default of no checking, Spring has three modes for dependency checking: `simple`, `objects`, and `all`. The `simple` mode checks to see whether all properties that are either Collections or a built-in type have a value. In this mode, Spring does not check to see whether or not properties of any other types are set. This mode can be quite useful for checking whether all the configuration parameters for a bean are set because they are typically either built-in values or Collections of built-in values.

The `objects` mode checks any property not covered by the `simple` mode, but it does not check properties that are covered by `simple`. So if you have a bean that has two properties, one of type `int` and the other of type `Foo`, then `objects` checks whether a value is specified for the `Foo` property but does not check for the `int` property.

The `all` mode checks all properties, essentially performing the checks of both the `simple` and `objects` mode. Listing 4-40 shows a `simple` class that has two properties: one `int` property and one property of the same type as the class itself.

Listing 4-40. *The SimpleBean Class*

```
package com.apress.prospring.ch4.depcheck;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class SimpleBean {

    private int someInt;

    private SimpleBean nestedSimpleBean;

    public void setSomeInt(int someInt) {
        this.someInt = someInt;
    }

    public void setNestedSimpleBean(SimpleBean nestedSimpleBean) {
        this.nestedSimpleBean = nestedSimpleBean;
    }
```

```
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/depcheck.xml"));

        SimpleBean simpleBean1 = (SimpleBean)factory.getBean("simpleBean1");
        SimpleBean simpleBean2 = (SimpleBean)factory.getBean("simpleBean2");
        SimpleBean simpleBean3 = (SimpleBean)factory.getBean("simpleBean3");
    }
}
```

The `main()` method in this code retrieves three beans, all of type `SimpleBean`, from the `BeanFactory`. Listing 4-41 shows the configuration for this `BeanFactory`.

Listing 4-41. *Configuring Dependency Checks*

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="simpleBean1" class="com.apress.prospring.ch4.depcheck.SimpleBean"
          dependency-check="simple">
        <property name="someInt">
            <value>16</value>
        </property>
    </bean>

    <bean id="simpleBean2" class="com.apress.prospring.ch4.depcheck.SimpleBean"
          dependency-check="objects">
        <property name="nestedSimpleBean">
            <ref local="nestedSimpleBean"/>
        </property>
    </bean>

    <bean id="simpleBean3" class="com.apress.prospring.ch4.depcheck.SimpleBean"
          dependency-check="all">
        <property name="someInt">
            <value>16</value>
        </property>
        <property name="nestedSimpleBean">
            <ref local="nestedSimpleBean"/>
        </property>
    </bean>

    <bean id="nestedSimpleBean"
          class="com.apress.prospring.ch4.depcheck.SimpleBean"/>
</beans>
```

As you can see from this configuration, each of the beans being retrieved from the `BeanFactory` in the Java app in Listing 4-40 has a different setting for the `dependency-check` attribute. The configuration currently ensures that all properties that need to be populated as per the `dependency-check` attribute have values, and as a result, the Java application runs without error. Try commenting out some of the `<property>` tags and see what happens then—Spring throws an `org.springframework.beans.factory.UnsatisfiedDependencyException`, indicating which property should have a value and does not have one.

Bean Inheritance

In some cases, you many need multiple definitions of beans that are the same type or implement a shared interface. This can become problematic if you want these beans to share some configuration settings but differ in others. The process of keeping the shared configuration settings in sync is quite error-prone, and on large projects, doing so can be quite time-consuming. To get around this, Spring allows you to define a `<bean>` definition that inherits its property settings from another bean in the same `BeanFactory`. You can override the values of any properties on the child bean as required, which allows you to have full control, but the parent bean can provide each of your beans with a base configuration. Listing 4-42 shows a simple configuration with two beans, one of which is the child of the other.

Listing 4-42. *Configuring Bean Inheritance*

```
<bean id="inheritParent" class="com.apress.prospring.ch4.inheritance.SimpleBean">
        <property name="name">
            <value>Rob Harrop</value>
        </property>
        <property name="age">
            <value>22</value>
        </property>
    </bean>

    <bean id="inheritChild" class="com.apress.prospring.ch4.inheritance.SimpleBean"
          parent="inheritParent">
        <property name="age">
            <value>35</value>
        </property>
    </bean>
```

In this code, you can see that the `<bean>` tag for the `inheritChild` bean has an extra attribute, `parent`, which indicates that Spring should consider the `inheritParent` bean the parent of the bean. Because the `inheritChild` bean has its own value for the `age` property, Spring passes this value to the bean. However, `inheritChild` has no value for the `name` property, so Spring uses the value given to the `inheritParent` bean. Listing 4-43 shows the code for the `SimpleBean` class used in a previous configuration.

Listing 4-43. *The SimpleBean Class*

```
package com.apress.prospring.ch4.inheritance;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class SimpleBean {

    public String name;

    public int age;

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                "./ch4/src/conf/beans.xml"));

        SimpleBean parent = (SimpleBean)factory.getBean("inheritParent");
        SimpleBean child = (SimpleBean)factory.getBean("inheritChild");

        System.out.println("Parent:\n" + parent);
        System.out.println("Child:\n" + child);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
        return    "Name: " + name + "\n"
                + "Age:" + age;
    }
}
```

As you can see, the `main()` method of the `SimpleBean` class grabs both the `inheritChild` and `inheritParent` beans from the `BeanFactory` and writes the contents of their properties to `stdout`. Here is the output from this example:

```
Parent:
Name: Rob Harrop
Age: 22
Child:
Name: Rob Harrop
Age: 35
```

As expected, the `inheritChild` bean inherited the value for its `name` property from the `inheritParent` bean, but was able to provide its own value for the `age` property.

Considerations for Using Bean Inheritance

Child beans inherit both constructor arguments and property values from the parent beans, so you can use both styles of injection with bean inheritance. This level of flexibility makes bean inheritance a powerful tool for building applications with more than a handful of bean definitions. If you are declaring a lot of beans of the same value with shared property values, then avoid the temptation to use copy and paste to share the values; instead, set up an inheritance hierarchy in your configuration.

When you are using inheritance, remember that bean inheritance does not have to match a Java inheritance hierarchy. It is perfectly acceptable to use bean inheritance on five beans of the same type. Think of bean inheritance as more like a templating feature than an inheritance feature. Be aware, however, that if you are changing the type of the child bean, then that type must be compatible with the type of the parent bean.

Summary

In this chapter, we covered a lot of ground with both the Spring core and IoC in general. We showed you examples of the different types of IoC and presented a discussion of the pros and cons of using each mechanism in your applications. We looked at which IoC mechanisms Spring provides and when and when not to use each within your applications. While exploring IoC, we introduced the Spring `BeanFactory`, which is the core component for Spring's IoC capabilities, and more specifically, we focused on the `XmlBeanFactory` that allows external configuration of Spring using XML.

This chapter also introduced you to the basics of Spring's IoC feature set including setter injection, constructor injection, auto-wiring, and bean inheritance. In the discussion of configuration, we demonstrated how you can configure your bean properties with a wide variety of different values, including other beans, using the `XmlBeanFactory`.

This chapter only scratches the surface of Spring and Spring's IoC container. In the next chapter, we look at some IoC-related features specific to Spring, and we take a more detailed look at other functionality available in the Spring core.