Chapter 2: The Bean Factory and Application Context

Overview

At the heart of Spring is its lightweight *Inversion of Control* (IoC) container functionality. One or more instances of a container are used to configure and wire together application and framework objects, and manage their lifecycles. The key principles of IoC ensure that the great majority of these objects do not have to have dependencies on the container itself, and can generally deal with related objects purely in terms of (Java) *interfaces* or abstract *superclasses*, without having to worry about how those other objects are implemented, or how to locate them. The IoC container is the basis for many of the features that will be introduced in the chapters that follow.

In this chapter, you will learn about the configuration and usage of Spring's bean factories and application contexts, as the embodiment of Spring's IoC container functionality. You will learn about the `BeanFactory` and `ApplicationContext` interfaces, along with related interfaces and classes, used when the container needs to be created or accessed in a programmatic fashion. This chapter will focus on the `BeanFactory` and `ApplicationContext` variants configured in a declarative fashion via XML. These generally have the most built-in functionality in terms of configuration and usage, and are the ones used by Spring users in the vast majority of cases. Note however that Spring decouples container configuration and usage. The next chapter will show how you may also access the full power of the containers using programmatic mechanisms for configuration, as well as alternative declarative formats.

We'll be looking at:

- Inversion of control and *Dependency Injection*
- Basic object configuration and wiring of objects in the Spring container
- How dependencies are resolved, and explicit versus automatic wiring of dependencies
- Handling object lifecycles in the Spring container
- Abstracting access to services and resources
- Bean and bean factory post-processors for customizing bean and container behavior
- Programmatic usage of the `BeanFactory` and `ApplicationContext` interfaces

Inversion of Control and Dependency Injection

In Chapter 1, you learned about *Inversion of Control* and why it is important. Let's recap briefly what this term really means, and then look at a few examples as they apply to the Spring container.

Software code is normally broken up into logical components or services that interact with each other. In Java, these components are usually instances of Java classes, or objects. Each object must use or work with other objects in order to do its job. For an object *A*, it can be said that the other objects that object *A* deals with are its *dependencies*. Inversion of Control refers to the generally desirable architectural pattern of having an outside entity (the container in this case) wire together objects, such that objects are given their dependencies by the container, instead of directly instantiating them themselves.

Take a look at some examples.

> **Note** Most larger examples in this chapter, even when not listed in complete form, are available separately in fully compilable form so that you may experiment with them. See the website.

Assume we have a historical weather data service, coded in a traditional, non-IoC style:

```
public class WeatherService {

  WeatherDAO weatherDao = new StaticDataWeatherDAOImpl();

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

public interface WeatherDAO {
  WeatherData find(Date date);
  WeatherData save(Date date);
  WeatherData update(Date date);
}

public class StaticDataWeatherDAOImpl implements WeatherDAO {

  public WeatherData find(Date date) {
    WeatherData wd = new WeatherData();
    wd.setDate((Date) date.clone());
    ...
    return wd;
  }

  public WeatherData save(Date date) {
    ...
  }
  public WeatherData update(Date date) {
    ...
  }
}
```

Following good practice, we will have a test case to verify that this code works. Based on JUnit, it might look as follows:

```
public class WeatherServiceTest extends TestCase {
```

```
  public void testSample1() throws Exception {
    WeatherService ws = new WeatherService();
    Double high = ws.getHistoricalHigh(
            new GregorianCalendar(2004, 0, 1).getTime());
    // ... do more validation of returned value here...
  }
}
```

Our weather service deals with a weather data DAO (Data Access Object) in order to get historical data. It deals with the DAO through a `WeatherDAO` interface, but in this example, the weather service directly instantiates a specific known type of weather DAO that implements that interface, `StaticDataWeatherDAOImpl`. Additionally, our driver application, `WeatherServiceTest`, directly uses a specific `WeatherService` class, with no possibility of specialization. This example is coded in non-IoC style. While the weather service does deal with the weather DAO through an interface, the weather service is directly instantiating a specific DAO type, and is controlling the lifecycle; thus it has a dependency on both the DAO interface and a specific implementation class. Additionally, the test sample representing the client of the service directly instantiates a specific weather service type, instead of dealing with it through an interface. In a real application, further *implicit* dependencies are likely —for example, on a particular persistence framework —and with this approach, they will be hard-coded in the caller code.

Now let's look at a simple example of how a Spring container can provide Inversion of Control. We first rework our weather service so that it is split up into an interface and implementation and allow the specific DAO instance to be set into that implementation object as a JavaBean property:

```
public interface WeatherService {
   Double getHistoricalHigh(Date date);
}

public class WeatherServiceImpl implements WeatherService {

  private WeatherDAO weatherDao;

  public void setWeatherDao(WeatherDAO weatherDao) {
    this.weatherDao = weatherDao;
  }

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

// same class as in previous example
public class StaticDataWeatherDAOImpl implements WeatherDAO {
    ...
}
```

We are going to use a Spring application context, specifically `ClasspathXmlApplicationContext`, to manage an instance of the weather service and ensure that it is given an instance of the weather DAO to work with. First we define a configuration file in XML format, `applicationContext.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
```

```
    <property name="weatherDao">
      <ref local="weatherDao"/>
    </property>
  </bean>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

We configure a `weatherService` object using the `bean` element, and specify that its `weatherDao` property should be set to an instance of the `weatherDao` bean, which we also define. Now we just need to modify the test to create the container and obtain the weather service from it:

```
public class WeatherServiceTest extends TestCase {
  public void testSample2() throws Exception {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
        "ch03/sample2/applicationContext.xml");
    WeatherService ws = (WeatherService) ctx.getBean("weatherService");

    Double high = ws.getHistoricalHigh(
        new GregorianCalendar(2004, 0, 1).getTime());
    // ... do more validation of returned value here...
  }
}
```

The classes are now coded and deployed in an IoC style. The weather service does not know or care about the implementation details of the actual weather DAO, on which it has a dependency solely through the `WeatherDAO` interface. Additionally, splitting the original `WeatherService` class into a `WeatherService` interface and `WeatherServiceImpl` implementation class shields clients of the weather service from implementation details such as how the weather DAO is provided to the weather service, through a JavaBeans style setter method in this case. It has the added benefit of allowing the actual weather service implementation itself to be transparently changed. At this point we may switch the actual `WeatherDAO` implementation and/or the `WeatherService` implementation, with only configuration file changes and clients of these components being unaware of the changes. This is an example of using interfaces when code in one layer uses code from another layer, as mentioned in Chapter 1.

Different Forms of Dependency Injection

The term *Dependency Injection* describes the process of providing a component with the dependencies it needs, in an IoC fashion, because the dependencies can be said to effectively be injected into the component. The form of dependency injection we saw in the previous example is called *Setter Injection* because JavaBean setter methods are used to supply the dependency to the object needing them. Please refer to `http://java.sun.com/products/javabeans` for more information about JavaBeans and the JavaBeans specification.

Now let's look at a variant of this, called *Constructor Injection*, where dependencies are instead supplied to an object via that object's own constructor:

```
public interface WeatherService {
   Double getHistoricalHigh(Date date);
}
public class WeatherServiceImpl implements WeatherService {

  private final WeatherDao weatherDao;

  public WeatherServiceImpl(WeatherDao weatherDao) {
    this.weatherDao = weatherDao;
  }

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
```

```
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

// WeatherDAO unchanged
public interface WeatherDAO {
  ...
}
// StaticDataWeatherDAOImpl unchanged
public class StaticDataWeatherDAOImpl implements WeatherDAO {
  ...
}
```

`WeatherServiceImpl` now has a constructor that takes `WeatherDAO`, instead of a setter method that takes this value. The application context configuration is modified accordingly. The test class is oblivious to the changes and does not have to be modified at all:

```
<beans>
  <bean id="weatherService" class="ch02.sample3.WeatherServiceImpl">
    <constructor-arg>
      <ref local="weatherDao"/>
    </constructor-arg>
  </bean>
  <bean id="weatherDao" class="ch02.sample3.StaticDataWeatherDaoImpl">
  </bean>
</beans>

// WeatherServiceTest unchanged
public class WeatherServiceTest extends TestCase {
  ...
}
```

Method Injection, the final form of dependency injection we are going to look at, is the most rarely used. In this form, the container is responsible for implementing methods at runtime. For example, an object might define a protected abstract method, and the container might implement it at runtime to return an object resulting from a container lookup. The aim of method injection is, again, to avoid dependencies on the container APIs, and reduce coupling.

One of the best uses of method injection is to handle the case where a singleton, stateless object, needs to use a non-singleton, stateful, or non-threadsafe object. Consider our weather service, of which there is only one instance needed because our implementation is stateless, and as such it is deployed with the default Spring container handling of being treated as a singleton, with only one instance ever being created, which is then cached and reused. What if the weather service needs to use `StatefulWeatherDAO`, a WeatherDAO implementation that is not threadsafe? On every invocation of `WeatherService.getHistoricalHigh()`, the weather service needs to use a fresh instance of the weather DAO (or at least make sure no other weather service invocation is using the same DAO at the same time). As you'll shortly see, we can easily tell the container to make the DAO a non-singleton object, so that each request for it returns a new instance, but we have a problem because the one instance of the weather service is injected with its dependencies only once. One option is for the weather service to be aware of the Spring container, and ask for a new DAO whenever it needs one. This, however, couples it to Spring where ideally it would not know anything about Spring at all. Instead, we can rely on Spring's *Lookup Method Injection* support, making the weather service access the weather DAO via a `getWeatherDAO()` JavaBean method, which can remain abstract or concrete as desired. Then in the factory definition, we tell the container to override this method and provide an implementation that returns the new DAO instance as another bean:

```
public abstract class WeatherServiceImpl implements WeatherService {

  protected abstract WeatherDao getWeatherDao();
```

```
  public Double getHistoricalHigh(Date date) {
    WeatherData wd = getWeatherDao().find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

<beans>
  <bean id="weatherService" class="ch02.sample4.WeatherServiceImpl">
    <lookup-method name="getWeatherDao" bean="weatherDao"/>
  </bean>
  <bean id="weatherDao" singleton="false"
        class="ch02.sample4.StatefulDataWeatherDaoImpl">
  </bean>
</beans>
```

Note that we have told the container to make the DAO a non-singleton, so that a new instance can actually be returned on each invocation of `getWeatherDao()`. Otherwise the same cached singleton instance would be returned each time. While this is legal, that's probably almost never what you want because the main benefit of using Lookup Method Injection, as in this case, is to inject prototypes. In this example, the `WeatherServiceImpl` class and `getWeatherDao` method are abstract, but we could override any getter method on any JavaBean. (In fact, candidate methods simply need to have no arguments; it is not necessary for them to follow JavaBeans naming conventions, although it's often a good idea.) The important point is that other code must use that getter method to access the DAO, not a field. This is, in any case, a good practice for any JavaBean property.

The use of Method Injection raises the question of how you would test the code without the container being around to inject the method, such as in a unit test. This may seem especially relevant for the case, such as this one, where the implementation class is abstract. One realistic and relatively simple strategy is for the purposes of the unit test to just subclass the abstract class, providing a test-specific implementation of the normally injected method; for example:

```
...
WeatherService ws = new WeatherServiceImpl() {
  protected WeatherDao getWeatherDao() {
    // return a value for the test
    ...
  }
};
```

The use of some advanced features such as Method Injection should always be subject to scrutiny by Spring users, who ultimately should decide which approach feels cleanest to them, but generally we feel this approach is preferred to putting in a code-level dependency on the container into application code.

Deciding Between Setter Injection and Constructor Injection

When using existing classes, you will sometimes not even have a choice as to whether to use setter injection or constructor injection. If a class has only a multi-argument constructor, and no JavaBean properties, or alternately only a no-arg constructor and simple JavaBean properties, the choice has effectively been made for you. Additionally, some existing classes may actually force you to use both forms in combination, with a multi-arg constructor that needs to be used, followed by the setting of some optional JavaBean properties.

When you do have a choice as to which form to architect for or to use, there are a number of aspects to consider:

- Using JavaBean properties generally makes it easier to handle default or optional values, in the case that not all values are actually needed. In the constructor case, this usually leads to multiple

constructor variants, with one calling another internally. The many variations or long argument lists can become very verbose and unmanageable.

- JavaBean properties (as long as they are not private) are automatically inherited by subclasses, while constructors are never inherited. The latter limitation often leads to the need to create relatively boilerplate constructors in subclasses, which call superclass constructors. Most IDEs, however, do now include code completion aids that make creation of either constructors or JavaBean properties fairly effortless.
- JavaBean properties are arguably better at being self-documenting than constructor arguments, at the source level. When adding JavaDocs, properties require less work to document as there is no duplication.
- At runtime, JavaBean properties may be used for matching based on name because properties have names visible by reflection. However, in a compiled class file, constructor argument names are not retained, so that automatic matching based on name is not possible.
- JavaBean properties allow getting the current state (as well as setting it) if a getter method is provided. This is useful in a number of situations, such as when the state needs to be stored elsewhere.
- The JavaBeans `PropertyEditor` mechanism exists for performing automatic type conversion when needed. This is in fact used and supported by Spring.
- JavaBean properties can be mutable because a setter method can be called multiple times. This allows changing a dependency if the use case actually supports it. Dependencies passed in via constructors cannot be mutable unless also exposed as properties. If, on the other hand, complete immutability is required, then Constructor Injection allows you to also explicitly declare the field set from a constructor argument as `final`, whereas the best that a setter method can do is throw an exception if it is called more than once; there is no way to make the field storing the value from the setter method `final`, so that other class code may not modify it.
- Constructor arguments make it easier to ensure a valid object is constructed because all required values can be declared and thus must be passed in, and the class is free to use them in the order they are needed, in the process of initialization. With JavaBean properties, there is the possibility that some properties have not been set before the object is used, resulting in an invalid state. Additionally, with JavaBean properties, there is no way to specify and enforce that setters must be called in a certain order, which may mean that initialization may need to take place after property setting, via the use of an `init` method. If used, such an `init` method can also verify that all needed properties have been set. Spring can automatically call any declared initialization method after all properties have been set, or alternately beans may implement the `InitializingBean` interface, which declares an `afterPropertiesSet()` method that will automatically be called.
- When there are only a few constructors, this can sometimes be less verbose than many JavaBean property accessor methods. Most IDEs, however, do now include code completion aids, which make creation of either constructors or JavaBean properties fairly effortless.

Generally, the Spring team prefers the use of setter injection over constructor injection for most cases in practice, although this decision should not be a hard and fast one. The aspects mentioned should provide most of the factors used to make the decision in each particular situation. Typically, constructor injection seems to work well for simpler initialization scenarios, with perhaps a constructor taking only a few arguments, which are ideally (easier to match) complex types that are not duplicated. As the configuration complexity scales up, setter injection seems to become more manageable and less work. Note that other major IoC containers also support both constructor and setter injection, so the possibility of Spring lock-in should not be a concern when making this choice.

Of course, whatever form is being used, this should really affect only the implementation class and actual configuration of that class. Most other code in the application should be working against interfaces, and should be completely unaffected by configuration concerns.

The Container

The basic IoC container in Spring is called the bean factory. Any bean factory allows the configuration and wiring together of objects using dependency injection, in a consistent and workable fashion. A bean factory als provides some management of these objects, with respect to their lifecycles. The IoC approach allows significant decoupling of code from other code. Additionally, along with the use of reflection to access the objects, it ensures that application code generally doesn't have to be aware of the bean factory, and generally doesn't have to be modified to be used with Spring. Application code needed to configure objects, and to obta objects using approaches such as singletons or other approaches such as special factory objects, can be completely eliminated or significantly curtailed.

Important Typically, only a very small amount of *glue* code in a typical Spring-based application will actually be aware of the Spring container, or use Spring container interfaces. Even this small amount of glue code can often be eliminated by relying on existing framework code to load bean factories in a declarative fashion.

The Bean Factory

Different bean factory implementations exist to support somewhat varying levels of functionality, in practice mostly related to the configuration mechanism, with XML being by far the most common representation. All bean factories implement the `org.springframework.beans.factory.BeanFactory` interface, with instances able to be accessed through this interface if they need to be managed in a programmatic fashion. Sub-interfaces of `BeanFactory` exist that expose extra functionality. A partial listing of the interface hierarchy includes:

- `BeanFactory`: The basic interface used to access all bean factories. The simple `getBean(String name)` method allows you to get a bean from the container by name. The `getBean(String name, Class requiredType)` variant allows you to specify the required class of the returned bean, throwing a exception if it doesn't exist. Additional methods allow you to query the bean factory to see if a bean exists (by name), to find out the type of a bean (given a bean name), and to find out if there are any aliases for a bean in the factory (the bean is known to the factory by multiple names). Finally, you may find out if a bea is configured as a singleton; that is, whether or not the first time a bean is requested it will be created and then reused for all subsequent requests, as opposed to a new instance being created each time.

- `HierarchicalBeanFactory`: Most bean factories can be created as part of a hierarchy, such that if yo ask a factory for a bean, and it doesn't exist in that particular factory, a parent factory is also asked for the bean, and that parent factory can ask a parent factory of its own, and so on. As far as the client is concerned, the entire hierarchy upwards of and including the factory being used can be considered as on merged factory. One reason to use such a hierarchical layout is to match the actual architectural layers o modules in an application. While getting a bean from a factory that is part of a hierarchy is transparent, th `HierarchicalBeanFactory` interface exists so that you may ask a factory for its parent.

- `ListableBeanFactory`: The methods in this sub-interface of `BeanFactory` allow listing or enumeratio of beans in a bean factory, including the names of all the beans, the names of all the beans of a certain type, and the number of beans in the factory. Additionally, several methods allow you to get a `Map` instan containing all beans of a certain type in the factory. It's important to note that while methods from the `BeanFactory` interface automatically take part in any hierarchy that a factory may be part of, the `ListableBeanFactory` interface applies strictly to one bean factory. The `BeanFactoryUtils` helper class provides essentially identical methods to `ListableBeanFactory`, which do take into account the entire bean factory hierarchy. These methods are often more appropriate for usage by application code.

- `AutowireCapableBeanFactory`: This interface allows you, via the `autowireBeanProperties()` an `applyBeanPropertyValues()` methods, to have the factory configure an existing external object and supply its dependencies. As such, it skips the normal object creation step that is part of obtaining a bean via `BeanFactory.getBean()`. When working with third-party code that insists on creating object instances itself, it is sometimes not an option for beans to be created by Spring, but still very valuable for Spring to inject bean dependencies. Another method, `autowire()`, allows you to specify a classname to the factory, have the factory instantiate that class, use reflection to discover all the dependencies of the class, and inject those dependencies into the bean, returning the fully configured object to you. Note that

the factory will not hold onto and manage the object you configure with these methods, as it would for singleton instances that it creates normally, although you may add the instance yourself to the factory as singleton.

- `ConfigurableBeanFactory`: This interface allows for additional configuration options on a basic bean factory, to be applied during the initialization stage.

Spring generally tries to use non-checked exceptions (subclasses of `RuntimeException`) for non-recoverable errors. The bean factory interfaces, including `BeanFactory` and its subclasses, are no different. most cases, configuration errors are non-recoverable, so all exceptions thrown by these APIs are subclasses the non-checked `BeansException`. Developers can choose where and when they handle these exceptions, even trying to recover from them if a viable recovery strategy exists.

The Application Context

Spring also supports a somewhat more advanced bean factory, called the *application context*.

> **Important** It is important to stress that an application context *is* a bean factory, with the `org.springframework.context.ApplicationContext` interface being a subclass of `BeanFactory`.

Generally, anything a bean factory can do, an application context can do as well. Why the distinction? It come down mostly to increased functionality, and usage style:

- **General *framework*-oriented usage style:** Certain operations on the container or beans in the containe which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context. This includes the automatic recognition and usage of special bean post-processor and bean factory post-processors, to be described shortly. Additionally, a number of Spring Framework facilities exist to automatically load application contexts, for example in the Web MVC layer, such that bea factories will mostly be created by user code, but application contexts will mostly be used in a declarative fashion, being created by framework code. Of course, in both cases, most of the user code will be managed *by* the container, and won't know anything *about* the container at all.
- `MessageSource` **support:** The application context implements `MessageSource`, an interface used to obtain localized messages, with the actual implementation being pluggable.
- **Support for application and framework events:** The context is able to fire framework or application events to registered listeners.
- `ResourceLoader` **support:** Spring's `Resource` interface is a flexible generic abstraction for handling lo level resources. An application context itself *is* a `ResourceLoader`, hence provides an application with access to deployment-specific `Resource` instances.

  > **Important** You may be wondering when it's most appropriate to create and use a bean factory versu an application context. In almost all cases you are better off using an application context because you will get more features at no real cost. The main exception is perhaps something like an applet where every last byte of memory used (or transferred) is significant, and a bean factory will save some memory because you can use the Spring library package, which brings in only bean factory functionality, without bringing in application context functionality. In this chapter and elsewhere, when bean factory functionality is discussed, you can safely assume that all such functionality is also shared by application contexts.

This chapter and most of this book describe container configuration and functionality using examples for the declarative, XML-configured variants (such as `XMLBeanFactory` or `ClassPathXmlApplicationContext`) of the bean factory and application context. It's important to realize that the container functionality is distinct from the configuration format. While the XML configuration format is used in the vast majority of cases by Spring users, there is a full programmatic API for configuring and accessing the containers, and other configuration formats can be built and supported in the same fashion as the XML variants. For example, a `PropertiesBeanDefinitionReader` class already exists for reading definitions in Java Properties file format into a bean factory.

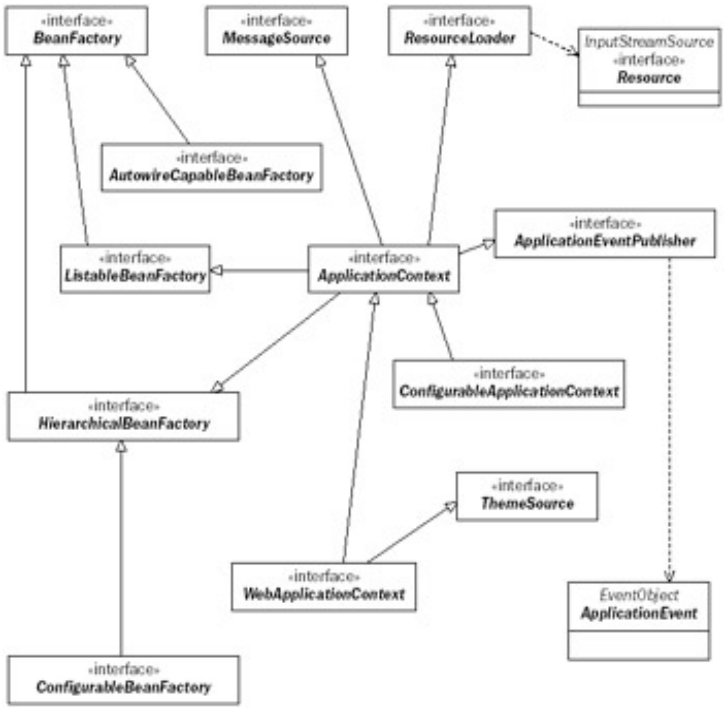

Figure 2-1

Starting the Container

From the previous examples, you already have an idea of how a container is programmatically started by user code. Let's look at a few variations.

We can load an `ApplicationContext` by pointing to the resource (file) on the classpath:

```
ApplicationContext appContext =
    new ClassPathXmlApplicationContext("ch03/sample2/applicationContext.xml");
// side note: an ApplicationContext is also a BeanFactory, of course!
BeanFactory factory = (BeanFactory) appContext;
```

Or we can point to a file system location:

```
ApplicationContext appContext =
    new FileSystemXmlApplicationContext("/some/file/path/applicationContext.xml");
```

We can also combine two or more XML file fragments. This allows us to locate bean definitions in the logical module they belong to while still producing one context from the combined definitions. As an example in the next chapter will show, this can be useful for testing as well. Here is one of the configuration samples seen before, but split up into two XML fragments:

`applicationContext-dao.xml:`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

`applicationContext-services.xml:`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref bean="weatherDao"/>
    </property>
  </bean>
</beans>
```

Note Careful readers will note that compared to a previous example we've slightly changed the way we referred to the `weatherDao` bean we use as a property of the weather service bean; later we'll describe bean references in much more detail.

Now to load and combine the two (or more) fragments, we just specify all of them:

```
ApplicationContext appContext = new ClassPathXmlApplicationContext(
    new String[] {"applicationContext-serviceLayer.xml", "applicationContext
dao.xml"});
```

Spring's `Resource` abstraction, which we will cover in depth later, allows us to use a `classpath*:` prefix to specify *all* resources matching a particular name that are visible to the class loader and its parents. If, for example, we have an application that is partitioned into multiple jars, all available on the class- path, and each jar contains its own application context fragment that is named `applicationContext. xml`, we easily specify that we want to create a context made up of all those fragments:

```
ApplicationContext appContext = new
ClassPathXmlApplicationContext("classpath*:ApplicationContext.xml");
```

Creating and loading an XML-configured bean factory is just as simple. The easiest mechanism is to use Spring's `Resource` abstraction for getting at a classpath resource:

```
ClassPathResource res =
    new ClassPathResource("org/springframework/prospering/beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
FilesystemResource res = new FilesystemResource("/some/file/path/beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

But we can also just use an `InputStream`:

```
InputStream is = new FileInputStream("/some/file/path/beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

Finally, for completeness, we show that we can easily split the step of creating the bean factory from parsing the bean definitions. We will not go into more depth here, but this distinction between bean factory behavior a parsing is what allows other configuration formats to be plugged in:

```
ClassPathResource res = new ClassPathResource("beans.xml");
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
```

```
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(res);
```

For application contexts, the use of the `GenericApplicationContext` class allows a similar separation of creation from parsing of definitions. Many Spring applications will never programmatically create a container themselves because they rely on framework code to do it. For example, you can declaratively configure Spring `ContextLoader` to automatically load an application context at web-app startup. You'll learn about this in the next chapter.

Using Beans from the Factory

Once the bean factory or application context has been loaded, accessing beans in a basic fashion is as simple as using `getBean()` from the `BeanFactory` interface:

```
WeatherService ws = (WeatherService) ctx.getBean("weatherService");
```

or other methods from some of the more advanced interfaces:

```
Map allWeatherServices = ctx.getBeansOfType(WeatherService.class);
```

Asking the container for a bean triggers the creation and initialization of the bean, including the dependency injection stage we've discussed previously. The dependency injection step can trigger the creation of other beans (the dependencies of the first bean), and so on, creating an entire graph of related object instances.

An obvious question might be what to do with the bean factory or application context itself, so that other code that needs it can get at it. As we're currently examining how the container is configured and how it works, we're mostly going to punt on this question here and defer it to later in this chapter.

Important Remember that excepting a small amount of glue code, the vast majority of application code written and assembled in proper IoC style does not have to be concerned at all with getting at the factory because the container will be hooking up these objects to other objects managed by the container. For the small amount of glue code that is needed to kick things off, the basic strategy is to put the factory in a known location, preferably somewhere that makes sense in the context of the expected usage and what code will actually need access to the factory. Spring itself provides support for declaratively loading an application context for web-app usage and storing it in the ServletContext. Additionally, it provides some quasi-singleton convenience classes that may be used for storing and getting at a factory, when a better fit or strategy does not exist for storing the factory in a particular app.

XML Bean Configuration

We've seen some sample XML format bean factory definition files, but have not gone into much detail so far. Essentially, a bean factory definition consists of just a top-level `beans` element containing one or more `bean` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref local="weatherDao"/>
    </property>
  </bean>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
```

```
</beans>
```

The valid elements and attributes in a definition file are fully described by the XML DTD (document type definition), `spring-beans.dtd`. This DTD, along with the Spring reference manual, should be considered th definitive source of configuration information. Generally, the optional attributes of the top- level `beans` elemen can affect the behavior of the entire configuration file and provide some default values for individual bean definition aspects, while (mostly) optional attributes and sub-elements of the child `bean` elements describe the configuration and lifecycle of individual beans. The DTD is includedas part of the Spring distribution, and you may also see it online at `www.springframework.org/dtd/spring-beans.dtd`.

The Basic Bean Definition

An individual bean definition contains the information needed for the container to know how to create a bean, some lifecycle details, and information about the bean's dependencies. Let's look at the first two aspects in thi section.

Identifier

For a top-level bean definition, you will almost always want to provide one or more identifiers, or names, for th bean, so that other beans may refer to it, or you may refer to it when using the container programmatically. Try to use the `id` attribute to supply the ID (or in the case of multiple IDs, the primary one). This has the advantag that because this attribute has the XML IDREF type, when other beans refer to this one, the XML parser can actually help detect whether the reference is valid (exists in the same file), allowing earlier validation of your factory config. However, XML IDREFs have a few limitations with regards to allowed characters, in that they must start with a letter followed by alphanumeric characters or the underscore, with no whitespace. This is no usually an issue, but to circumvent these limitations, you may instead use the `name` attribute to provide the identifier. This is useful, for example, when the bean identifier is not completely under the user's control and it represents a URL path. Additionally, the `name` attribute actually allows a comma-separated list of IDs. When a bean definition specifies more than one ID, via the combination of the `id` attribute and/or the `name` attribute, t additional IDs after the first can be considered *aliases*. All IDs are equally valid when referring to the bean. Le look at some examples:

```
<beans>

  <bean id="bean1" class="ch02.sample5.TestBean"/>

  <bean name="bean2" class="ch02.sample5.TestBean"/>

  <bean name="/myservlet/myaction" class="ch02.sample5.TestBean"/>

  <bean id="component1-dataSource"
        name="component2-dataSource,component3-dataSource"
        class="ch02.sample5.TestBean"/>
</beans>
```

The third bean needs an ID that starts with /, so it may not use the `id` attribute, but must use `name`. Note that the fourth bean has three IDs, all equally valid. You may be wondering why you would ever want to provide more than one ID for a bean. One valid use case is when you want to split up your configuration by componer or modules, such that each component provides one XML file fragment that liststhe beans related to that component, and their dependencies. The names of these dependencies can (as one example) be specified wi a component-specific prefix, such as was used for the hypothetical `DataSource` in the previous code sample When the final bean factory or context is assembled from the multiple fragments (or, as will be described later a hierarchy of contexts is created), each component will end up referring to the same actual bean. This is a lo tech way of providing some isolation between components.

Bean Creation Mechanism

You also need to tell the container how to instantiate or obtain an instance of the bean, when it is needed. Th most common mechanism is creation of the bean via its constructor. The `class` attribute is used to specify th

classname of the bean. When the container needs a *new* instance of the bean, it will internally perform the equivalent of using the `new` operator in Java code. All the examples we have seen so far use this mechanism

Another mechanism is to tell the container to use a static *factory-method* to obtain the new instance. Legacy code over which you have no control will sometimes force you to use such a static factory method. Use the `class` attribute to specify the name of the class that contains the static factory method, and specify the name of the actual method itself via the `factory-method` attribute:

```
...
<bean id="testBeanObtainedViaStaticFactory"
      class="ch02.sample4.StaticFactory" factory-method="getTestBeanInstance"/>
...

public class StaticFactory {
  public static TestBean getTestBeanInstance() {
    return new TestBean();
  }
}
```

The static factory method may return an object instance of any type; the class of the instance returneddoesn't have to be the same as the class containing the factory method.

The third mechanism for the container to get a new bean instance is for it to call a non-static factory method o a different bean instance in the container:

```
...
<bean id="nonStaticFactory"  class="ch02.sample4.NonStaticFactory"/>

<bean id="testBeanObtainedViaNonStaticFactory"
      factory-bean="nonStaticFactory" factory-method="getTestBeanInstance"/>
...

public class NonStaticFactory {
  public TestBean getTestBeanInstance() {
    return new TestBean();
  }
}
```

When a new instance of `testBeanObtainedViaNonStaticFactory` is needed, an instance of `nonStaticFactory` is first created, and the `getTestBeanInstance` method on that is called. Note that in this case, we do not specify any value at all for the `class` attribute.

Once a new object instance is obtained, the container will treat it the same regardless of whether it was obtained via a constructor, via a static factory method, or via an instance factory method. That is, setter injection can be used and normal lifecycle callbacks will apply.

Singleton versus Non-Singleton Beans

An important aspect of the bean lifecycle is whether the container treats it as a singleton or not. Singleton beans, the default, are created only once by the container. The container will then hold and use the same instance of the bean whenever it is referred to again. This can be significantly less expensive in terms of resource (memory or potentially even CPU) usage than creating a new instance of the bean on each request. As such, it's the best choice when the actual implementation of the classes in question allow it; that is, the bea is stateless, or state is set only once at initialization time, so that it is *threadsafe*, and may be used by multiple threads at a time. Singleton beans are the default because in practice most services, controllers, and resourc that end up being configured in the container are implemented as threadsafe classes, which do not modify any state past initialization time.

A non-singleton, or *prototype* bean as it is also called, may be specified by setting the `singleton` attribute to

false. It's important to note that the lifecycle of a prototype bean will often be different than that of a singleton bean. When a container is asked to supply a prototype bean, it is initialized and then used, but the container does not hold onto it past that point. So while it's possible to tell Spring to perform some end-of-lifecycle operations on singleton beans, as we will examine in a subsequent section, any such operations need to be performed by user code for prototype beans because the container will no longer know anything about them a that point:

```
<bean id="singleton1" class="ch02.sample4.TestBean"/>

<bean id="singleton2" singleton="true" class="ch02.sample4.TestBean"/>

<bean id="prototype1" singleton="false" class="ch02.sample4.TestBean"/>
```

Specifying Bean Dependencies

Satisfying bean dependencies, in the form of other beans or simple values the first bean needs, is the meat a potatoes or core functionality of the container, so it's important to understand how the process works. You've already seen examples of the main types of dependency injection, *constructor injection* and *setter injection*, a know that Spring supports both forms. You've also seen how, instead of using a constructor to obtain an initia object instance to work with, Spring can use a factory method. With respect to supplying dependencies to the bean, using a factory method to obtain a bean instance can essentially be considered equivalent to getting tha instance via a constructor. In the constructor case, the container supplies (optional) argument values, which a the dependencies, to a constructor. In the factory-method case, the container supplies (optional) argument values, which are the dependencies, to the factory method. Whether the initial object instance comes through constructor or factory method, it is from that point on treated identically.

Constructor injection and setter injection are not mutually exclusive. If Spring obtains an initial bean instance v a constructor or factory method, and supplies argument values to the constructor or factory method, thus injecting some dependencies, it is still able to then use setter injection to supply further dependencies. This ca be useful, for example, when you need to use and initialize an existing class that has a constructor taking one or more arguments, that produces a bean in a known (valid) initial state, but relies on JavaBeans setter methods for some optional properties. Without using both forms of injection, you would not be able to properly initialize this type of object if any of the optional properties needed to be set.

Let's examine how the container initializes and resolves bean dependencies:

- The container first initializes the bean definition, without instantiating the bean itself, typically at the time o container startup. The bean dependencies may be explicitly expressed in the form of constructor arguments or arguments to a factory method, and/or bean properties.
- Each property or constructor argument in a bean definition is either an actual value to set, or a reference another bean in the bean factory or in a parent bean factory.
- The container will do as much validation as it can at the time the bean definition is initialized. When using the XML configuration format, you'll first get an exception from the XML parser if your configuration does not validate against the XML DTD. Even if the XML is valid in terms of the DTD, you will get an exception from Spring if it is able to determine that a definition is not logically valid; for example two properties may be mutually exclusive, with no way for the DTD to express this.
- If a bean dependency cannot be satisfied (that is, a bean dependency is another bean that doesn't actua exist), or a constructor-argument or property value cannot be set properly, you will get an error only when the container actually needs to get a new instance of the bean and inject dependencies. If a bean instanc is never needed, then there is the potential that the bean definition contains (dependency) errors you will not find out about (until that bean is eventually used). Partially in order to provide you with earlier error trapping, application contexts (but not bean factories) by default pre-instantiate singleton bean instances. The pre- instantiation stage simply consists of enumerating all singleton (the default state) beans, creatin an instance of each including injection of dependencies, and caching that instance. Note that the pre-instantiation stage can be overridden through the use of the `default-lazy-init` attribute of the top-level `beans` element, or controlled on an individual bean basis by using the `lazy-init` attribute of the `bean` element.

- When the container needs a new bean instance, typically as the result of a `getBean()` call or another bean referring to the first bean as a dependency, it will get the initial bean instance via the constructor or factory method that is configured, and then start to try injecting dependencies, the optional constructor or factory method arguments, and the optional property values.
- Constructor arguments or bean properties that refer to another bean will force the container to create or obtain that other bean first. Effectively, the bean that is referred to is a dependency of the referee. This ca trigger a chain of bean creation, as an entire dependency graph is resolved.
- Every constructor argument or property value must be able to be converted from whatever type or format was specified in, to the actual type that that constructor argument or property is expecting, if the two type are not the same. Spring is able to convert arguments supplied in string format to all the built-in scalar types, such as `int`, `long`, `boolean`, and so on, and allthe wrapper types such as `Integer`, `Long`, `Boolean`, and so on. Spring also uses JavaBeans `PropertyEditors` to convert potentially any type of String values to other, arbitrary types. A number of built-in `PropertyEditors` are automatically registere and used by the container. A couple of examples are the `ClassEditor PropertyEditor`, which converts a classname string into an actual Class object instance, which may be fed to a property expecti a Class, or the `ResourceEditor PropertyEditor`, which converts a string location path into Spring's Resource type, which is used to access resources in an abstract fashion. All the built-in property editors are described in the next chapter. As will be described later in this chapter in the section "Creating a Custom PropertyEditor," it is possible to register your own `PropertyEditors` to handle your own custor types.
- The XML configuration variant used by most bean factory and application context implementations also has appropriate elements/attributes allowing you to specify complex values that are of various Collection types, including Lists, Sets, Maps, and Properties. These Collection values may nest arbitrarily.
- Dependencies can also be implicit, to the extent that even if they are not declared, Spring is able to use reflection to see what constructor arguments a bean takes, or what properties it has, and build up a list o valid dependencies for the bean. It can then, using functionality called *autowiring*, populate those dependencies, based on either *type* or *name* matching. We'll ignore autowiring for the time being, and come back to it later.

Specifying Bean Dependencies, Detailed

Let's look in detail at the specifics of supplying bean properties and constructor arguments in XML format. Eac `bean` element contains zero (0) or more `constructor-arg` elements, specifying constructor or lookup meth arguments. Each `bean` element contains zero (0) or more `property` elements, specifying a JavaBean prope to set. Constructor arguments and JavaBean properties may be combined if needed, which the following example takes advantage of. It has a constructor argument that is a reference to another bean, and an `int` property, a simple value:

```
<beans>
  <bean id="weatherService" class="ch02.sample6.WeatherServiceImpl">
    <constructor-arg index="0">
      <ref local="weatherDao"/>
    </constructor-arg>
    <property name="maxRetryAttempts"><value>2</value></property>
  </bean>

  <bean id="weatherDao" class="ch02.sample6.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

From the XML DTD, we can see that a number of elements are allowed inside a `property` or `constructor arg` element. Each is used to specify some sort of value for a property or constructor argument:

```
(bean | ref | idref | list | set | map | props | value | null)
```

The `ref` element is used to set the value of a property or constructor argument to be a reference to another bean from the factory, or from a parent factory:

```
<ref local="weatherDao"/>
<ref bean="weatherDao"/>
<ref parent="weatherDao"/>
```

The `local`, `bean`, or `parent` attributes are mutually exclusive, and must be the ID of the other bean. When using the `local` attribute, the XML parser is able to verify at parse time that the specified bean exists. However, because this relies on XML's IDREF mechanism, that bean must be in the same XML file as the reference, and its definition must use the `id` attribute to specify the ID being referred to, not `name`. When the `bean` attribute is used, the specified bean may be located in the same or another XML fragment that is used t build up the factory definition, or alternately in any parent factory of the current factory. However, Spring itself (not the XML parser) will validate that the specified bean exists, and this will happen only when that dependency actually needs to be resolved, not at factory load time. The much less frequently used `parent` attribute specifies that the target bean must come from a parent factory to the current one. This may be used the rare cases when there is a name conflict between a bean in the current factory and a parent factory.

The `value` element is used to specify simple property values or constructor arguments. As previously mentioned, conversion happens from the source form, which is a string, to the type of the target property or constructor arg, which can be any of the built-in scalar types or related wrapper types, and can also be any typ for which a JavaBean `PropertyEditor` capable of handling it is registered in the container. So for example,

```
<property name="classname">
  <value>ch02.sample6.StaticDataWeatherDaoImpl</value>
</property>
```

will set a `String` property called `classname` to the string literal value `ch02.sample6.StaticDataWeatherDaoImpl`, but if `classname` is instead a property of type `java.lang.Class`, the factory will use the built-in (automatically registered) `ClassEditor` `PropertyEditor` to convert the string value to an actual `Class` object instance.

It is easy to register your own `PropertyEditors` to handle string conversion to any other custom types that need to be handled in the container config. One good example of where this is useful is for entering dates in string form, to be used to set Date properties. Because dates are very much locale sensitive, use of a `PropertyEditor` that expects a specific source string format is the easiest way to handle this need. How to register custom `PropertyEditors` will be demonstrated in the description of `CustomEditorConfigurer`, bean factory *post-processor*, later in this chapter. Also, it is a little-known fact that the JavaBeans `PropertyEditor` machinery will automatically detect and use any `PropertyEditors` that are in the same package as the class they are meant to convert, as long as they have the same name as that class with "Editor" appended. That is, for a class `MyType`, a `PropertyEditor` called `MyTypeEditor` in the same package would automatically be detected and used by the JavaBeans support code in the Java library, withou having to tell Spring about it.

Properties or constructor arguments that need to be set to Null values require special treatment because an empty `value` element is treated as just an empty string. Instead, use the special `null` element:

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

The `idref` element is a convenient way to catch errors when specifying the name of another bean as a string value. There are some Spring-specific helper beans in the framework that as property values take the name o another bean and perform some action with it. You would typically use the form

```
<property name="beanName"><value>weatherService</value></property>
```

to populate these properties. It would be nice if typos could somehow be caught, and in fact this is what `idre` does. Using the form

```
<property name="beanName"><idref local="weatherService"/></property>
```

allows the XML parser to participate in validation, as it will catch references to beans that don't actually exist. The resulting property `value` will be exactly the same as if the first value tag had been used.

The `list`, `set`, `map`, and `props` elements allow complex properties or constructor arguments of type `java.util.List`, `java.util.Set`, `java.util.Map`, and `java.util.Properties` to be defined and se Let's look at a completely artificial example, in which a JavaBean has a List, Set, Map, and Properties property set:

```
<beans>
  <bean id="collectionsExample" class="ch02.sample7.CollectionsBean">
    <property name="theList">
      <list>
        <value>red</value>
        <value>red</value>
        <value>blue</value>
        <ref local="curDate"/>
        <list>
          <value>one</value>
          <value>two</value>
          <value>three</value>
        </list>
      </list>
    </property>
    <property name="theSet">
      <set>
        <value>red</value>
        <value>red</value>
        <value>blue</value>
      </set>
    </property>
    <property name="theMap">
      <map>
        <entry key="left">
          <value>right</value>
        </entry>
        <entry key="up">
          <value>down</value>
        </entry>
        <entry key="date">
          <ref local="curDate"/>
        </entry>
      </map>
    </property>
    <property name="theProperties">
      <props>
        <prop key="left">right</prop>
        <prop key="up">down</prop>
      </props>
    </property>
  </bean>

  <bean id="curDate" class="java.util.GregorianCalendar"/>
</beans>
```

List, Map, and Set values may be any of the elements

```
(bean | ref | idref | list | set | map | props | value | null)
```

As shown in the example for the list, this means the collection types can nest arbitrarily. One thing to keep in mind is that the properties or constructor arguments receiving Collection types must be of the generic types

`java.util.List`, `java.util.Set`, or `java.util.Map`. You cannot depend on the Spring-supplied collection being of a specific type, for example `ArrayList`. This presents a potential problem if you need to populate a property in an existing class that takes a specific type because you can't feed a generic type to it. One solution is to use `ListFactoryBean`, `SetFactoryBean`, and `MapFactoryBean`, a set of helper *Factory Beans* available in Spring. They allow you to specify the collection type to use, for instance a `java.util.LinkedList`. Please see the Spring JavaDocs for more info. We will discuss factory beans shortly.

The final element allowed as a property value or constructor argument, or inside one of the collection elements is the `bean` element. This means a bean definition can effectively nest inside another bean definition, as a property of the outer bean. Consider a variation of our original setter injection example:

```
<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <bean class="ch02.sample2.StaticDataWeatherDaoImpl">
        ...
      </bean>
    </property>
  </bean>
</beans>
```

Nested bean definitions are very useful when there is no use for the inner bean outside of the scope of the outer bean. In the preceding example, the weather DAO, which is set as a dependency of the weather service, has been moved to be an inner bean. No other bean or external user will ever need the weather DAO, so there is no use in keeping it as a separate outer-scope bean definition. Using the inner bean form is more concise and clearer. There is no need for the inner bean to have an ID, although it is legal. *Note*: Inner beans are always prototypes, with the singleton flag being ignored. In this case, this is essentially irrelevant; because there is only one instance of the outer bean ever created (it's a singleton), there would only ever be one instance of the dependency created, whether that dependency is marked as singleton or prototype. However if a prototype outer bean needs to have a singleton dependency, then that dependency should not be wired as an inner bean, but rather as a reference to a singleton external bean.

Manual Dependency Declarations

When a bean property or constructor argument refers to another bean, this is a declaration of a dependency on that other bean. You will sometimes have a need to force one bean to be initialized before another, even if it is not specified as a property of the other. The most typical case for this is when a class does some static initialization when it is loaded. For example, database drivers typically register themselves with the JDBC `DriverManager`. The `depends-on` attribute may be used to manually specify that a bean is dependent on another, triggering the instantiation of that other bean first when the dependent bean is accessed. Here's an example showing how to trigger the loading of a database driver:

```
<bean id="load-jdbc-driver" class="oracle.jdbc.driver.OracleDriver "/>

<bean id="dataBaseUsingBean" depends-on="load-jdbc-driver" class="..." >
   ...
</bean>
```

Note that most database connection pools and Spring support classes such as `DriverManagerDataSource` trigger this kind of loading themselves, so this is for example only.

Autowiring Dependencies

We've so far seen explicit declarations of bean dependencies through property values and constructor arguments. Under many circumstances, Spring is able to use introspection of bean classes in the factory and perform *autowiring* of dependencies. In autowiring, you leave the bean property or constructor argument undeclared (in the XML file), and Spring will use reflection to find the type and name of the property, and then match it to another bean in the factory based on type or name. This can potentially save a significant amount

typing, at the possible expense of some loss of transparency. You may control autowiring at both the entire container level and the individual bean definition level. Because autowiring may have uncertain effects when r used carefully, all autowiring is off by default. Autowiring at the bean level is controlled via the use of the `autowire` attribute, which has five possible values:

- `no`: No autowiring at all for this bean. Bean properties and constructor arguments must be explicitly declared, and any reference to another bean must be done via the use of a `ref` element. This is the default handling for individual beans when the default is not changed at the bean factory level. This mode is recommend in most circumstances, especially for larger deployments, as explicitly declared dependencies are a form of explicit documentation and much more transparent.
- `byName`: Autowire by property name. The property names are used to find matching beans in the factory. a property has the name "weatherDao," then the container will try to set the property as a reference to another bean with the name "weatherDao." If there are no matching beans, then the property is left unset This handling treats unmatched properties as optional. If you need to treat unmatched properties as an error case, you may do so by adding the `dependencycheck="objects"` attribute to the bean definition as described later.
- `byType`: Autowire by matching type. This is similar to the approach of PicoContainer, another popular dependency injection container. For each property, if there is exactly one bean in the factory of the same type as the property, the property value is set as that other bean. If there is more than one bean in the factory matching the type, it is considered a fatal error and an exception is raised. As for `byName` autowiring, if there are no matching beans, then the property is left unset. If this needs to be treated as a error case, the `dependency-check="objects"` attribute may be used on the bean definition, as described later.
- `constructor`: Autowire the constructor by type. This works in essentially identical fashion to how `byTyp` works for properties, except that there must be exactly one matching bean, by type, in the factory for eac constructor argument. In the case of multiple constructors, Spring will try to be greedy and satisfy the constructor with the most matching arguments.
- `autodetect`: Choose `byType` or `constructor` as appropriate. The bean is introspected, and if there is a default no-arg constructor, `byType` is used, otherwise, `constructor` is used.

It is possible to set a different default autowire mode (than the normal `no`) for all beans in the factory by using the `default-autowire` attribute of the top-level `beans` element. Note also that you may mix autowiring and explicit wiring, with explicit `property` or `constructor-arg` elements specifying dependencies always taking precedence for any given property.

Let's look at how autowiring could work for our weather service and weather DAO. As you can see, we remov the `weatherDao` property definition in the bean definition, turn on autowiring by name, and Spring will still populate the property based on a name match. We could also have used autowiring by type because the property is of a type `WeatherDao`, and only one bean in the container matches that type:

```
<beans>
  <bean id="weatherService" autowire="byName"
        class="ch02.sample2.WeatherServiceImpl">
    <!-- no more weatherDao property declaration here -->
  </bean>

  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

It may be tempting to try to use autowiring extensively to try to save typing in the factory configurations, but we would caution you to be very careful when using this feature.

Important Removing explicitly declared dependencies also removes a form of documentation of those dependencies. Additionally, when using `byType` or even `byName`, there is the potential for surprises when more than one bean matches, or no bean matches. For larger, more complicated deployments especially, we recommend you stay away from autowiring, or use it very judiciously, as you may otherwise find you have actually increased the complexity even though you have reduced the amount of XML. Most IDEs now have DTD-aware XML editors built-in or available as plugins, which can save most of the typing when creating bean configurations, so the verbosity of explicit dependency declarations is not as much of a concer as it once would have been. What may work well in some situations is to rely on autowiring for simple low-level plumbing —say, a `DataSource` —and to use explicit wiring for more complicated aspects. This tends to reduce clutter without causing surprises or sacrificing muc explicitness.

Constructor Argument Matching

As a general rule, you should almost always use the optional `index` or `type` attributes with `constructor-arg` elements you specify. While these attributes are optional, without one or the other, the list of constructor arguments you specify is resolved to actual constructor arguments based on matching of types. When the arguments you specify are references to different types of complex beans, or complex types such as a Map, it easy for the container to do this matching, especially if there is only one constructor. However, when specifyin multiple arguments of the same type, or using the `value` tag, which can be considered to be (in source strin form) an untyped value, trying to rely on automatic matching can produce errors or unexpected results.

Consider a bean that has a constructor taking a numeric error code value and a String error message value. I we try to use the `<value>` tag to supply values for these arguments, we need to give the container a hint so i can do its job. We can either use the `index` attribute to specify the correct (0-based) argument index, matchi the actual constructor:

```
<beans>
  <bean id="errorBean" class="ch02.sampleX.ErrorBean">
    <constructor-arg index="0"><value>1000<value></constructor-arg>
    <constructor-arg index="1"><value>Unexpected Error<value></constructor-arg>
  </bean>
</beans>
```

or alternately, we can give the container enough information that it can do proper matching based on type, by using the `type` attribute to specify the type of the value:

```
<beans>
  <bean id="errorBean" class="ch02.sampleX.ErrorBean">
    <constructor-arg type="int"><value>1000<value></constructor-arg>
    <constructor-arg type="java.lang.String">
      <value>Unexpected Error<value>
    </constructor-arg>
  </bean>
</beans>
```

Validating Bean Dependencies

Often, some JavaBean properties on an object are optional. You're free to set them or not as needed for the particular use case, and there would be no easy way for the container to try to help you in catching errors due to a property that needs to be set but isn't. However, when you have a bean in which all properties, or all properties of a certain nature, need to be set, the container's dependency validation feature can help out. Wh enabled, the container will consider it an error if properties are not supplied either by explicit declaration or through autowiring. By default, the container will not try to validate that all dependencies are set, but you may customize this behavior with the `dependency-check` attribute on a bean definition, which may have the following values:

–

- `none`: No dependency validation. If a property has no value specified, it is not considered an error. This i the default handling for individual beans when the default is not changed at the bean factory level.
- `simple`: Validate that primitive types and collections are set, considering it an error if they are not set. Other properties may be set or not set.
- `objects`: Validate that properties that are not primitive types or collections are set, considering it an erro if they are not set. Other properties may be set or not set.
- `all`: Validate that all properties are set, including primitive types, collections, and complex types.

It is possible to set a different default dependency check mode (than the normal `none`) for all beans in the factory by using the `default-dependency-check` attribute of the top-level `beans` element.

Note also that the `InitializingBean` callback interface described in the next section may also be used to manually verify dependencies.

Managing the Bean Lifecycle

A bean in the factory can have a very simple or relatively complex lifecycle, with respect to things that happen to it. Since we're talking about POJOs, the bean lifecycle does not have to amount to anything more than creation and usage of the object. However, there are a number of ways that more complex life-cycles can be managed and handled, mostly centering around bean lifecycle callbacks that the bean and that third-party *observer* objects called *bean post-processors* can receive at various stages in the initialization and destructio phases. Let's examine the possible container-driven actions (described in the following table) that can happen in the lifecycle of a bean managed by that container.

Action	Description
Initialization begins as bean is instantiated	The new bean is instantiated via a constructor, or by calling a factory method, which is considered equivalent. This hap- pens as a result of a `getBean()` call on the factory, or the fact that another bean that was already being instantiated had a dependency on this one, triggering the creation of this one first.
Dependencies injected	Dependencies are injected into the instance, as previously discussed.
`setBeanName()` called	If the bean implements the optional interface, `Bean NameAware`, then that interface' s `setBeanName()` method is called to provide the bean with its primary ID as declared in the bean definition.
`setBeanFactory()` called	If the bean implements the optional `BeanFactoryAware` interface, then that interface' s `setBeanFactory()` method is called to provide the bean with a reference to the factory it is deployed in. Note that since application contexts are also bean factories, this method will also be called for beans deployed in an application context, however passing in the bean factory internally used by the context.
`setResourceLoader()` called	If the bean implements the optional `ResourceLoader Aware` interface, and it is deployed in an application con- text, then that interface' s `setResourceLoader()` method is called, with the application context as the `ResourceLoader`. (To be discussed in the next chapter.)

`setApplicationEventPublisher` called	If the bean implements the optional `ApplicationEventPublisherAware` interface, and it is deployed in an application context, then that interface's `setApplicationEventPublisher()` method is called, with the application context as the `ApplicationEvent Publisher`. (To be discussed in the next chapter.)
`setMessageSource()` called	If the bean implements the optional `MessageSourceAware` interface, and it is deployed in an application context, then that interface's `setMessageSource()` method is called, with the application context as the `MessageSource`. (To be discussed in the next chapter.)
`setApplicationContext()` called	If the bean implements the optional `ApplicationContext Aware()` interface, and is deployed in an application context, then that interface's `setApplicationContext()` method is called to provide the bean with a reference to the context.
Bean post-processors get "before-initialization" callback with bean	Bean post-processors, which will be discussed shortly, are special handlers that applications may register with the factory. Post-processors get a pre-initialization callback with the bean, which they may manipulate as needed.
`afterPropertiesSet()` called	If the bean implements the `InitializingBean` marker interface, then `afterPropertiesSet()` from this interface is called to allow the bean to initialize itself.
Declared `init` method called	If the bean definition defines an initialization method via the `init-method` attribute, then this method is called to allow the bean to initialize itself.
Bean post-processors get "after-initialization" callback with the bean instance as argument	Any bean post-processors get a post-initialization callback with the bean instance, which they may manipulate as needed, as an argument.
Bean is used	The bean instance is actually used. That is, it's returned to the caller of `getBean()`, used to set a property on the other bean that triggered its creation as a dependency, and so on. **Important note:** Only singleton beans are tracked past this point, with prototype beans being considered as owned by the client using the bean. As such, the container will orchestrate only the subsequent lifecycle events for singleton beans. Any prototype beans have to be fully managed by the client past this point, including calling any needed destroy method.
Bean destruction begins	As part of the bean factory or application context shutdown process, all cached singleton beans go through a destruction process, consisting of the actions following this one. Note that beans are destroyed in appropriate order related to their dependency relationship, generally the reverse of the initialization order.
Bean post-processors get "destroy" callback with bean	Any bean post-processors implementing the `DestructionAwareBeanPostProcessors` interface get a callback to manipulate the singleton bean for destruction.
`destroy()` called	If a singleton bean implements the `DisposableBean` marker interface, then that interface's `destroy()` method is called to allow the bean to do any needed resource cleanup.

Declared destroy method called	If the bean definition of a singleton bean defines a destroy method via the `destroy-method` attribute, then this method is called to allow the bean to release any resources that need to be released.

Initialization and Destruction Callbacks

The initialization and destroy methods mentioned previously may be used to allow the bean to perform any needed resource allocation or destruction. When trying to use an existing class with uniquely named init or destroy methods, there is not much choice other than to use the `init-method` and `destroy-method` attributes to tell the container to call these methods at the appropriate time, as in the following example, where we need to call `close()` on a DBCP-based `DataSource`:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>jdbc:oracle:thin:@db-server:1521:devdb</value>
  </property>
  <property name="username"><value>john</value></property>
  <property name="password"><value>password</value></property>
</bean>
```

Generally, even for new development we recommend the use of the `init-method` and `destroy-method` attributes to tell the container about init methods or destroy methods, as opposed to the other alternative of having the bean implement the Spring-specific interfaces `InitializingBean` and `DisposableBean`, with their corresponding `afterPropertiesSet()` and `destroy()` methods. The latter approach is more convenient as the interfaces are recognized by Spring and the methods are called automatically, but you are a the same time unnecessarily tying your code to the Spring container. If the code is tied to Spring for other reasons, then this is not as much of a concern, and use of the interfaces can make life simpler for the deploye Spring Framework code intended to be deployed as beans in the container uses the interfaces frequently.

As mentioned, non-singleton, prototype beans are not kept or managed by the container past the point of initialization. As such, it's impossible for Spring to call destroy methods on non-singleton beans, or involve the bean in any container-managed lifecycle actions. Any such methods must be called by user code. Additionally bean post-processors do not get a chance to manipulate the bean at the destruction stage.

BeanFactoryAware and ApplicationContextAware Callbacks

A bean that wants to be aware of and access its containing bean factory or application context for any sort of programmatic manipulation may implement the `BeanFactoryAware` and `ApplicationContextAware` interfaces, respectively. In the order listed in the lifecycle actions table earlier in the chapter, the container will call into the bean via the `setBeanFactory` and `setApplicationContext()` methods of these interfaces, passing the bean a reference to itself. Generally most application code should not need to know about the container, but this is sometimes useful in Spring-specific code, and these callbacks are used in many of Spring's own classes. One situation when application code may want a reference to the factory is if a singleto bean needs to work with prototype beans. Because the singleton bean has its dependencies injected only onc the dependency injection mechanism would not appear to allow the bean to get new prototype instances as needed. Therefore, accessing the factory directly allows it to do so. However, we feel that Lookup Method Injection, already mentioned, is a better mechanism to handle this use case for most situations because with that solution the class is completely unaware of Spring or the prototype bean name.

Abstracting Access to Services and Resources

While there are a number of more advanced bean factory and application context features we have not yet touched on, at this point we've seen almost all the lower-level building blocks necessary for successful IoC-sty programming and deployment. We've seen how application objects can do their work purely with other objects

that have been provided to them by the container, working through interfaces or abstract superclasses, and n caring about the actual implementation or source of those objects. You should already know enough to be off and running in typical usage scenarios.

What is probably not very clear is how those other collaborators are actually obtained, when they can be configured and accessed in such a diverse fashion. Let's walk through some examples, to see how Spring lets you avoid some potential complexity and manage things transparently.

Consider a variation of our weather DAO, which instead of working with static data, uses JDBC to access historical data from a database. An initial implementation might use the original JDBC 1.0 `DriverManager` approach to get a connection; shown here is the `find()` method:

```
public WeatherData find(Date date) {
  // note that in the constructor or init method, we have done a
  // Class.forName(driverClassName) to register the JDBC driver
  // The driver, username and password have been configured as
  // properties of the bean
  try {
    Connection con = DriverManager.getConnection(url, username, password);
    // now use the connection
    ...
```

When we deploy this DAO as a bean in a Spring container, we already have some benefits, as we can easily feed in whatever values are needed for the JDBC driver url, username, and password, via Setter or Construct Injection. However, we're not going to get any connection pooling, either in a standalone or J2EE container environment, and our connection is not going to be able to participate in any J2EE container-managed transactions, which work only through container-managed connections.

The obvious solution is to move to JDBC 2.0 `DataSources` for obtaining our connections. Once the DAO has a `DataSource`, it can just ask it for a connection, and not care how that connection is actually provided. The theoretical availability of a `DataSource` is not really a problem; we know that there are standalone connection pool implementations such as DBCP from Apache Jakarta Commons that can be used in a J2SE or J2EE environment, exposed through the `DataSource` interface, and that in most J2EE container environments a container-managed connection pool is also available, which will participate in container-managed transactions These are also exposed as `DataSources`.

However, trying to move to obtaining connections through the `DataSource` interface introduces additional complexity because there are different ways to create and obtain a `DataSource`. A DBCP `DataSource` is created as a simple JavaBean, which is fed some configuration properties, while in most J2EE container environments, a container-managed `DataSource` is obtained from JNDI and used, with a code sequence similar to the following:

```
try {
  InitialContext context = new InitialContext();
  DataSource ds = (DataSource) context.lookup("java:comp/env/jdbc/datasource");
  // now use the DataSource
  ...
}
catch (NamingException e) {
  // handle naming exception if resource missing
}
```

Other `DataSources` might have a completely different creation/access strategy.

Our DAO could perhaps know itself how to create or obtain each type of `DataSource`, and be configured for which one to use. Because we've been learning Spring and IoC, we know this is not a great idea, as it ties the DAO to the `DataSource` implementation unnecessarily, makes configuration harder, and makes testing more complicated. The obvious IoC solution is to make the `DataSource` just be a property of the DAO, which we can inject into the DAO via a Spring container. This works great for the DBCP implementation, which can

create the `DataSource` as a bean and inject it into the DAO:

```
public class JdbcWeatherDaoImpl implements WeatherDAO {

  DataSource dataSource;

  public void setWeatherDao(DataSource dataSource) {
    this.dataSource = dataSource;
  }
  public WeatherData find(Date date) {
    try {
      Connection con = dataSource.getConnection();
      // now use the connection
      ...
  }
  ...
}

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>jdbc:oracle:thin:@db-server:1521:devdb</value>
  </property>
  <property name="username"><value>john</value></property>
  <property name="password"><value>password</value></property>
</bean>

<bean id="weatherDao" class="ch02.sampleX.JdbcWeatherDaoImpl">
  <property name="DataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

Now how do we manage to swap out the use of the DBCP `DataSource` with the use of a `DataSource` obtained from JNDI? It would not appear to be that simple, given that we need to set a property of type `DataSource` on the DAO, but need to get this value to come from JNDI; all we know how to do so far in the container configuration is define JavaBean properties as values and references to other beans. In fact,that's still all we have to do, as we leverage a helper class called `JndiObjectFactoryBean`!

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/datasource</value>
  </property>
</bean>

<bean id="weatherDao" class="ch02.sampleX.JdbcWeatherDaoImpl">
  <property name="DataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

Examining Factory Beans

`JndiObjectFactoryBean` is an example of a *Factory Bean*. A Spring factory bean is based on a very simpl concept: Essentially it's just a bean that on demand produces another object. All factory beans implement the special `org.springframework.beans.factory.FactoryBean` interface. The "magic" really happens

because a level of indirection is introduced. A factory bean is itself just a normal JavaBean. When you deploy factory bean, as with any JavaBean, you specify properties and constructor arguments needed for it to do its work. However, when another bean in the container refers to the factory bean, via a `<ref>` element, or when manual request is made for the factory bean via `getBean()`, the container does not return the factory bean itself; it recognizes that it is dealing with the factory bean (via the marker interface), and it returns the output o the factory bean. So for all intents and purposes, each factory bean can, in terms of being used to satisfy dependencies, be considered to *be* the object it actually produces. In the case of `JndiObjectFactoryBean` for example, this is the result of a JNDI lookup, specifically a `DataSource` object in our example.

The `FactoryBean` interface is very simple:

```
public interface FactoryBean {

  Object getObject() throws Exception;

  Class getObjectType();

  boolean isSingleton();

}
```

The `getObject()` method is called by the container to obtain the output object. The `isSingleton()` flag indicates if the same object instance or a different one will be returned on each invocation. Finally, the `getObjectType()` method indicates the type of the returned object, or null if it is not known. While factory beans are normally configured and deployed in the container, they are just JavaBeans, so they are usable programmatically if desired.

Important This raises the question of how a deployed factory bean may be obtained via a `getBean()` ca if asking for a factory bean actually returns its output. This is possible using an "escaping" mechanism to tell the container that you want the factory bean, and not its output. This is to prepend an & to the bean ID, as follows:

```
FactoryBean facBean = (FactoryBean) appContext.getBean
("&dataSource");
```

It's easy to create your own factory beans. However, Spring includes a number of useful factory bean implementations that cover most of the common resource and service access abstractions that benefit from being handled in this manner. Just a partial list of these includes:

- `JndiObjectFactoryBean`: Returns an object that is the result of a JNDI lookup.
- `ProxyFactoryBean`: Wraps an existing object in a proxy and returns it. What the proxy actually does is configured by the user, and can include interception to modify object behavior, the performance of securit checks, and so on. The usage of this factory bean will be described in much more detail in the chapter or Spring AOP.
- `TransactionProxyFactoryBean`: A specialization of the `ProxyFactoryBean` that wraps an object with a transactional proxy.
- `RmiProxyFactoryBean`: Creates a proxy for accessing a remote object transparently via RMI. `HttpInvokerProxyFactoryBean`, `JaxRpcPortProxyFactoryBean`, `HessianProxyFactoryBean` and `BurlapProxyFactoryBean` produce similar proxies for remote object access over HTTP, JAX-RPC Hessian, and Burlap protocols, respectively. In all cases, clients are unaware of the proxy and deal only with the business interface.
- `LocalSessionFactoryBean`: Configures and returns a Hibernate `SessionFactory` object.Similar classes exist for JDO and iBatis resource managers.
- `LocalStatelessSessionProxyFactoryBean` **and** `SimpleRemoteStatelessSessionProxyFactoryBean`: Create a proxy object used for accessing loc or remote stateless Session Beans, respectively. The client just uses the business interface, without worrying about JNDI access or EJB interfaces.

- `MethodInvokingFactoryBean`: Returns the result of a method invocation on another bean. `FieldRetrievingFactoryBean` returns the value of a field in another bean.
- A number of JMS-related factory beans return JMS resources.

Key Points and the Net Effect

In previous sections, we saw how easy it is to hook up objects in an IoC fashion. However, before getting to th stage of wiring up objects to each other, they do have to be able to be created or obtained first. For some potential collaborators, even if once obtained and configured they are ultimately accessed via a standard interface, the fact that they are normally created or obtained through complicated or nonstandard mechanisms creates an impediment to even getting the objects in the first place. Factory beans can eliminate this impediment. Past the initial object creation and wiring stage, the products of factory beans, as embodied in proxies and similar wrapper objects, can serve in an adapter role, helping in the act of abstracting actual resource and service usage, and making dissimilar services available through similar interfaces.

As you saw, without the client (`weatherDao`) being at all aware of it or having to be modified, we swapped ou the original DBCP-based `DataSource`, created as a local bean, to a `DataSource` that came from JNDI. Ultimately, IoC enabled this swapping, but it was necessary to move resource access out of the application code, so that it could be handled by IoC.

> **Important** We hope that one of the main things this will impress upon you is that when you can easily abstract service and resource access like this, and switch from one mechanism to the other at will, there is really no reason to use deployment scenarios and technologies that don't make sense at the particular time you put them into effect. In any case, the client should be agnostic to the actual deployment and implementation scenario.

If you can transparently (to the client) access remote services via RMI, RPC over HTTP, or EJBs, why should you not deploy the solution that makes the most sense, and why would you couple the client to one implementation over another when you don't have to? J2EE has traditionally pushed the idea of exposing som resources such as `DataSources`, JMS resources, JavaMail interfaces, and so on via JNDI. Even if it makes sense for the resources to be exposed there, it never makes sense for the client to do direct JNDI access. Abstracting via something like `JndiObjectFactoryBean` means that you can later switch to an environment without JNDI, simply by changing your bean factory config instead of client code. Even when you do not need change the deployment environment or implementation technology in production, these abstractions make uni and integration testing much easier, as they enable you to employ different configurations for deployment and test scenarios. This will be shown in more detail in the next chapter. It is also worth pointing out that using `JndiObjectFactoryBean` has removed the need for some non-trivial code in the DAO —JNDI lookup —th has nothing to do with the DAO's business function. This demonstrates the de-cluttering effect of dependency injection.

Reusing Bean Definitions

Considering the amount of Java code related to configuration and object wiring that they replace, XML bean definitions are actually fairly compact. However, sometimes you will need to specify a number of bean definitions that are quite similar to each other.

Consider our `WeatherService`:

```
<bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>
```

In a typical application scenario where the backend is a database, we need to use service objects like this transactionally. As you will learn in detail in Chapter 6, "Transaction and DataSource Management," one of the easiest ways to accomplish this is to declaratively wrap the object so that it becomes transactional, with a Spring factory bean called `TransactionProxyFactoryBean`:

```
<bean id="weatherServiceTarget" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>

<!-- transactional proxy -->
<bean id="weatherService"
 class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

  <property name="target"><ref local="weatherServiceTarget"/></property>

  <property name="transactionManager"><ref local="transactionManager"/></property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Don't worry for now about the exact details of how `TransactionProxyFactoryBean` is configured or how it actually does its work. What is important to know is that it's a `FactoryBean` that, given a target bean as input produces as its output a transactional proxy object, implementing the same interfaces as the target bean, but now with transactional usage semantics. Because we want clients to use the wrapped object, the original unwrapped (non-transactional) bean has been renamed to `weatherServiceTarget`, and the proxy now has the name `weatherService`. Any existing clients using the weather service are unaware that they are now dealing with a transactional service.

Being able to wrap objects declaratively like this is very convenient (especially compared to the alternative of doing it programmatically at the code level), but in a large application with dozens or hundreds of service interfaces that need to be wrapped in an almost similar fashion, it seems somewhat wasteful to have so much essentially identical, boilerplate XML. In fact, the container's ability to allow both *parent* and *child* bean definitions is meant exactly for this sort of situation. Taking advantage of this, we can now use the approach of having an abstract parent, or *template*, transaction proxy definition:

```
<bean id="txProxyTemplate" abstract="true"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref local="transactionManager"/></ref>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Then we create each real proxy as a child definition that has only to specify properties that are different from the parent template:

```
<bean id="weatherServiceTarget" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>

<bean id="weatherService" parent="txProxyTemplate">
  <property name="target"><ref local="weatherServiceTarget"/></ref></property>
</bean>
```

In fact, it's possible to get an even cleaner and somewhat more compact form. Because no client will ever need to use the unwrapped weather service bean, it can be defined as an inner bean of the wrapping proxy:

```
<bean id="weatherService" parent="txProxyTemplate">
  <property name="target">
    <bean class="ch02.sample2.WeatherServiceImpl">
      <property name="weatherDao">
        <ref local="weatherDao"/>
      </property>
    </bean>
  </property>
</bean>
```

Another service that needs to be wrapped can just use a bean definition that derives from the parent template in a similar fashion. In the following example, the `transactionAttributes` property from the parent templa is also overridden, in order to add transaction propagation settings that are specific to this particular proxy:

```
<bean id="anotherWeatherService" parent="txProxyTemplate">
  <property name="target">
    <bean class="ch02.sampleX.AnotherWeatherServiceImpl">
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED </prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

Important AOP "autoproxying" can provide an even simpler way of capturing commonality between AOF advice on different bean definitions. See Chapter 4, "Spring and AOP," for more information.

Child Bean Definition Specifics

A child bean definition inherits property values and constructor arguments defined in a parent bean definition. also inherits a number of the optional attributes from the parent definition (if actually specified in the parent). A seen in the previous example, the `parent` attribute in the child bean definition is used to point to the parent.

The child bean definition may specify a class, or leave it unset to inherit the value from the parent. If the child definition does specify a class (that is different from the parent), the class must still be able to accept any constructor arguments or properties specified in the parent definition because they will be inherited and used.

From the parent, child definitions inherit any constructor argument values, property values, and method overrides, but may add new values as needed. On the other hand, any `init-method`, `destroy-method`, or `factory-method` values from the parent are inherited, but completely overridden by corresponding values in the child.

Some bean configuration settings are never inherited, but will *always* be taken from the child definition. These are: `depends-on`, `autowire`, `dependency-check`, `singleton`, and `lazy-init`.

Beans marked as abstract using the `abstract` attribute (as used in our example) may not themselves be instantiated. Unless there is a need to instantiate a parent bean, you should almost always specifically mark it as abstract. This is a good practice because a non-abstract parent bean, even if you do not specifically ask th container for it or refer to it as a dependency, may possibly still be instantiated by the container. Application contexts (but not bean factories) will by default try to pre-instantiate non-abstract singleton beans.

Important Note also that even without an explicit `abstract` attribute, bean definitions can be considere implicitly abstract if they do not specify a class or factory-method because there is not enough information to be able to instantiate them. Any attempt to instantiate an (explicitly or implicitly) abstract bean will result in an error.

Using Post-Processors to Handle Customized Beans and Containers

Bean *post-processors* are special *listeners,* which you may register (either explicitly or implicitly) with the container, that receive a callback from the container for each bean that is instantiated. Bean factory postprocessors are similar listeners that receive a callback from the container when the *container* has been instantiated. You will periodically come across a need to customize a bean, a group of beans, or the entire container configuration, which you will find is handled most easily by creating a post-processor, or using one o the number of existing post-processors included with Spring.

Bean Post-Processors

Bean post-processors implement the `BeanPostProcessor` interface:

```
public interface BeanPostProcessor {
  Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException;
  Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException;
}
```

The `DestructionAwareBeanPostProcessor` extends this interface:

```
public interface DestructionAwareBeanPostProcessor extends BeanPostProcessor {
  void postProcessBeforeDestruction(Object bean, String name)
      throws BeansException;
}
```

The bean lifecycle table shown previously listed at which points in the bean lifecycle each of these call-backs occurs. Let's create a bean post-processor that uses the `postProcessAfterInitialization` call-back tc list to the console every bean that has been initialized in the container.

```
public class BeanInitializationLogger implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException {
    return bean;
  }

  public Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException {

    System.out.println("Bean ‘" + beanName + "' initialized");
    return bean;
  }
}
```

Note This example doesn't do very much; a real-life post-processor would probably manipulate the actual bean instance, and would write to a log file, not the console.

Important In an application context, bean post-processors are automatically recognized and used by the container, when deployed like any other bean:

```
<beans>
  <bean id="weatherService"
class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref local="weatherDao"/>
    </property>
  </bean>
  <bean id="weatherDao"
class="ch02.sample2.StaticDataWeatherDaoImpl"/>
  <bean id="beanInitLogger"
class="ch02.sample8.BeanInitializationLogger"/>
</beans>
```

When the context specified by this configuration is loaded, the post-processor gets two callbacks and produce the following output:

```
Bean 'weatherDao' initialized
Bean 'weatherService' initialized
```

Using bean post-processors with a simple bean factory is slightly more complicated because they must be manually registered with the factory (in the spirit of the bean factory's more programmatic usage approach) instead of just being added as a bean in the XML config itself:

```
XmlBeanFactory factory = new XmlBeanFactory(
        new ClassPathResource("ch03/sample8/beans.xml"));
BeanInitializationLogger logger = new BeanInitializationLogger();
factory.addBeanPostProcessor(logger);
// our beans are singletons, so will be pre-instantiated, at
// which time the post-processor will get callbacks for them too
factory.preInstantiateSingletons();
```

As you can see, `BeanFactory.preInstantiateSingletons()` is called to pre-instantiate the singleton beans in the factory because only application contexts pre-instantiate singletons by default. In any case, the post-processor is called when the bean is actually instantiated, whether as part of pre-instantiation, or when the bean is actually requested, if pre-instantiation is skipped.

Bean Factory Post-Processors

Bean factory post-processors implement the `BeanFactoryPostProcessor` interface:

```
public interface BeanFactoryPostProcessor {
  void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
          throws BeansException;
}
```

Here's an example bean factory post-processor that simply gets a list of all the bean names in the factory and prints them out:

```
public class AllBeansLister implements BeanFactoryPostProcessor {

  public void postProcessBeanFactory(ConfigurableListableBeanFactory factory)
      throws BeansException {

    System.out.println("The factory contains the followig beans:");
```

```
    String[] beanNames = factory.getBeanDefinitionNames();
    for (int i = 0; i < beanNames.length; ++i)
      System.out.println(beanNames[i]);
  }
}
```

Using bean factory post-processors is relatively similar to using bean post-processors. In an application context, they simply need to be deployed like any other bean, and will be automatically recognized and used b the context. With a simple bean factory on the other hand, they need to be manually executed against the factory:

```
XmlBeanFactory factory = new XmlBeanFactory(
        new ClassPathResource("ch03/sample8/beans.xml"));
AllBeansLister lister = new AllBeansLister();
lister.postProcessBeanFactory(factory);
```

Let's now examine some useful bean post-processors and bean factory post-processors that come with Sprin

PropertyPlaceholderConfigurer

Often when deploying a Spring-based application, most items in a container configuration are not meant to be modified at deployment time. Making somebody go into a complex configuration and change a few values that *do* need to be customized can be inconvenient. It can also potentially be dangerous because somebody may make a mistake and invalidate the configuration by accidentally modifying some unrelated values.

`PropertyPlaceholderConfigurer` is a bean factory post-processor that when used in a bean factory or application context definition allows you to specify some values via special *placeholder* strings, and have thos placeholders be replaced by real values from an external file in Java Properties format. Additionally, the configurer will by default also check against the Java System properties if there is no match for the placeholde string in the external file. The `systemPropertiesMode` property of the configurer allows turning off the fallback, or making the System Properties have precedence.

One example of values that would be nice to externalize are configuration strings for a database connection pool. Here's our previous DBCP-based `DataSource` definition, now using placeholders for the actual values:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClassName"><value>${db.driverClassName}</value></property>
  <property name="url"><value>${db.url}</value></property>
  <property name="username"><value>${db.username}</value></property>
  <property name="password"><value>${db.password}</value></property>
</bean>
```

The real values now come from an external file in Properties format: `jdbc.properties`.

```
db.driverClassName=oracle.jdbc.driver.OracleDriver
db.url=jdbc:oracle:thin:@db-server:1521:devdb
db.username=john
db.password=password
```

To use a `PropertyPlaceholderConfigurer` instance to pull in the proper values in an application context the configurer is simply deployed like any other bean:

```
<bean id="placeholderConfig"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location"><value>jdbc.properties</value></property>
</bean>
```

To use it with a simple bean factory, it must be manually executed:

```
    XmlBeanFactory factory = new XmlBeanFactory(
            new ClassPathResource("beans.xml"));
PropertyPlaceholderConfigurer ppc = new PropertyPlaceholderConfigurer();
ppc.setLocation(new FileSystemResource("db.properties"));
ppc.postProcessBeanFactory(factory);
```

PropertyOverrideConfigurer

`PropertyOverrideConfigurer`, a bean factory post-processor, is somewhat similar to `PropertyPlaceholderConfigurer`, but where for the latter, values *must* come from an external Propertie files, `PropertyOverrideConfigurer` allows values from the external Properties to *override* bean property values in the bean factory or application context.

Each line in the Properties file must be in the format:

```
beanName.property=value
```

Where `beanName` is the ID of a bean in the container, and `property` is one of its properties. An example Properties file could look like this:

```
dataSource.driverClassName=oracle.jdbc.driver.OracleDriver
dataSource.url=jdbc:oracle:thin:@db-server:1521:devdb
dataSource.username=john
dataSource.password=password
```

This would override four properties in the bean "dataSource." Any properties (of any bean) in the container, nc overridden by a value in the Properties file, will simply remain at whatever value the container config specifies for it, or the default value for that bean if the container config does not set it either. Because just looking at the container config will not give you an indication that a value is going to be overridden, without also seeing the Properties file, this functionality should be used with some care.

CustomEditorConfigurer

`CustomEditorConfigurer` is a bean factory post-processor, which allows you to register custom JavaBean `PropertyEditors` as needed to convert values in String form to the final property or constructor argument values in a specific complex object format.

Creating a Custom PropertyEditor

One particular reason you might need to use a custom `PropertyEditor` is to be able to set properties of typ `java.util.Date`, as specified in string form. Because date formats are very much locale sensitive, use of a `PropertyEditor`, which expects a specific source string format, is the easiest way to handle this need. Because this is a very concrete problem that users are likely to have, rather than present some abstract example we're just going to examine how Spring's own existing `CustomDateEditor`,a `PropertyEditor` fo Dates, is coded and how you may register and use it. Your own custom `PropertyEditors` would be implemented and registered in similar fashion:

```
public class CustomDateEditor extends PropertyEditorSupport {

  private final DateFormat dateFormat;
  private final boolean allowEmpty;

  public CustomDateEditor(DateFormat dateFormat, boolean allowEmpty) {
    this.dateFormat = dateFormat;
```

```
    this.allowEmpty = allowEmpty;
  }

  public void setAsText(String text) throws IllegalArgumentException {
    if (this.allowEmpty && !StringUtils.hasText(text)) {
      // treat empty String as null value
      setValue(null);
    }
    else {
      try {
        setValue(this.dateFormat.parse(text));
      }
      catch (ParseException ex) {
        throw new IllegalArgumentException("Could not parse date: " +
                ex.getMessage());
      }
    }
  }

  public String getAsText() {
    return (getValue() == null ? "" : this.dateFormat.format((Date) getValue()));
  }
}
```

For full JavaDocs for this class, please see the Spring distribution. Essentially though, to implement your own `PropertyEditor`, the easiest way is, as this code does, to start by inheriting from the `java.beans.PropertyEditorSupport` convenience base class that is part of the standard Java library. This implements most of the property editor machinery other than the actual `setAsText()` and `getAsText(` methods, the standard implementations of which you must normally override. Note that `PropertyEditors` have state, so they are not normally threadsafe, but Spring does ensure that they are used in a threadsafe fashion by synchronizing the entire sequence of method calls needed to perform a conversion.

`CustomDateEditor` uses any implementation of the `java.text.DateFormat` interface to do the actual conversion, passed in as a constructor argument. When you deploy it, you can use a `java.text.SimpleDateFormat` for this purpose. It can also be configured to treat empty strings as null values, or an error.

Registering and Using the Custom PropertyEditor

Now let's look at an application context definition in which `CustomEditorConfigurer` is used to register `CustomDateEditor` as a `PropertyEditor` to be used for converting strings to `java.util.Date` objects A specific date format string is provided:

```
<bean id="customEditorConfigurer"
      class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>

      <-- register property editor for java.util.Date -->
      <entry key="java.util.Date">
        <bean class="org.springframework.beans.propertyeditors.CustomDateEditor">
          <constructor-arg index="0">
            <bean class="java.text.SimpleDateFormat">
              <constructor-arg><value>M/d/yy</value></constructor-arg>
            </bean>
          </constructor-arg>
          <constructor-arg index="1"><value>true</value></constructor-arg>
        </bean>
      </entry>
```

```
    </map>
  </property>
</bean>

<-- try out the date editor by setting two Date properties as strings -->
<bean id="testBean" class="ch02.sample9.StartEndDatesBean">
  <property name="startDate"><value>10/09/1968</value></property>
  <property name="endDate"><value>10/26/2004</value></property>
</bean>
```

The `CustomEditorConfigurer` can register one or more custom `PropertyEditors`, with this sample registering only one. Your own custom `PropertyEditors` for other types might not need any special configuration. `CustomDateEditor`, which has a couple of constructor arguments in this sample, took a `SimpleDateFormat` with a date format string as the first, and a Boolean indicating that empty strings should produce null values, as the second.

The example also showed how a test bean called "testBean" had two properties of type Date set via string values, to try out the custom property editor.

BeanNameAutoProxyCreator

`BeanNameAutoProxyCreator` is a bean post-processor. The AOP chapter will talk in more detail about how it's used, but it's worth knowing that it exists. Essentially, given a list of bean names, `BeanNameAutoProxyCreator` is able to wrap beans that match those names in the factory, as they are instantiated, with proxy objects that intercept access to those original beans, or modify their behavior.

DefaultAdvisorAutoProxyCreator

This bean post-processor is similar to `BeanNameAutoProxyCreator`, but finds beans to wrap, along with the information on how to wrap them (the Advice), in a more generic fashion. Again, it's worth reading about it in the AOP chapter.

PREV NEXT

Summary

This chapter has given you a good feel for what the terms *Inversion of Control* and *Dependency Injection* really mean, and how they are embodied in Spring's bean factories and application contexts. We've examined and used most of Spring's fundamental container functionality. It's upon the IoC container as a foundation that the rest of Spring builds, and good knowledge of how it works and is configured, along with all of its capabilities, is the key to effectively utilizing all of Spring.

You have seen how:

- Using the container allows the use of one consistent, predictable mechanism to access, configure, and wire together objects, instead of using programmatic or ad hoc mechanisms that couple classes to each other and make testing harder. Generally, application or class-specific custom factories and singletons can be completely avoided.
- The container encourages and makes easy the generally desirable practice of separating interface and implementation in application code.
- Post-processors add the ability to customize bean and container behavior in a flexible, externalized fashion.
- IoC principles, combined with the factory bean, afford a powerful means to abstract the act of obtaining or accessing services and resources.
- IoC and the container provide a powerful base upon which Spring and application code can build higher value-adding functionality, without generally being tied to the container.

The next chapter looks at some more advanced capabilities of the application context, and will offer some advanced usage scenarios of the container in general.