> as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually (attempt to) pre-instantiate the `abstract` bean.

7.8 Container Extension Points

Typically, an application developer does not need to subclass `ApplicationContext` implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

Customizing beans using a BeanPostProcessor

The `BeanPostProcessor` interface defines *callback methods* that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more `BeanPostProcessor` implementations.

You can configure multiple `BeanPostProcessor` instances, and you can control the order in which these BeanPostProcessors execute by setting the `order` property. You can set this property only if the `BeanPostProcessor` implements the `Ordered` interface; if you write your own `BeanPostProcessor` you should consider implementing the `Ordered` interface too. For further details, consult the javadocs of the `BeanPostProcessor` and `Ordered` interfaces. See also the note below on programmatic registration of `BeanPostProcessors`.

> **Note**
>
> BeanPostProcessors operate on bean (or object) *instances*; that is to say, the Spring IoC container instantiates a bean instance and *then* BeanPostProcessors do their work.
>
> BeanPostProcessors are scoped *per-container*. This is only relevant if you are using container hierarchies. If you define a `BeanPostProcessor` in one container, it will *only* post-process the beans in that container. In other words, beans that are defined in one container are not post-processed by a `BeanPostProcessor` defined in another container, even if both containers are part of the same hierarchy.
>
> To change the actual bean definition (i.e., the *blueprint* that defines the bean), you instead need to use a `BeanFactoryPostProcessor` as described in the section called "Customizing configuration metadata with a BeanFactoryPostProcessor".

The `org.springframework.beans.factory.config.BeanPostProcessor` interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both *before* container initialization methods (such as InitializingBean's *afterPropertiesSet()* and any declared init method) are called as well as *after* any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces or may wrap a bean with a proxy. Some Spring AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

An `ApplicationContext` *automatically detects* any beans that are defined in the configuration metadata which implement the `BeanPostProcessor` interface. The `ApplicationContext` registers

these beans as post-processors so that they can be called later upon bean creation. Bean post-processors can be deployed in the container just like any other beans.

Note that when declaring a BeanPostProcessor using an `@Bean` factory method on a configuration class, the return type of the factory method should be the implementation class itself or at least the `org.springframework.beans.factory.config.BeanPostProcessor` interface, clearly indicating the post-processor nature of that bean. Otherwise, the `ApplicationContext` won't be able to autodetect it by type before fully creating it. Since a BeanPostProcessor needs to be instantiated early in order to apply to the initialization of other beans in the context, this early type detection is critical.

> **Programmatically registering BeanPostProcessors**
>
> While the recommended approach for `BeanPostProcessor` registration is through `ApplicationContext` auto-detection (as described above), it is also possible to register them *programmatically* against a `ConfigurableBeanFactory` using the `addBeanPostProcessor` method. This can be useful when needing to evaluate conditional logic before registration, or even for copying bean post processors across contexts in a hierarchy. Note however that `BeanPostProcessors` added programmatically *do not respect the `Ordered` interface*. Here it is the *order of registration* that dictates the order of execution. Note also that `BeanPostProcessors` registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.

> **BeanPostProcessors and AOP auto-proxying**
>
> Classes that implement the `BeanPostProcessor` interface are *special* and are treated differently by the container. All `BeanPostProcessors` *and beans that they reference directly* are instantiated on startup, as part of the special startup phase of the `ApplicationContext`. Next, all `BeanPostProcessors` are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a `BeanPostProcessor` itself, neither `BeanPostProcessors` nor the beans they reference directly are eligible for auto-proxying, and thus do not have aspects woven into them.
>
> For any such bean, you should see an informational log message: "*Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)*".
>
> Note that if you have beans wired into your `BeanPostProcessor` using autowiring or `@Resource` (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates, and therefore make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with `@Resource` where the field/setter name does not directly correspond to the declared name of a bean and no name attribute is used, then Spring will access other beans for matching them by type.

The following examples show how to write, register, and use `BeanPostProcessors` in an `ApplicationContext`.

Example: Hello World, BeanPostProcessor-style

This first example illustrates basic usage. The example shows a custom `BeanPostProcessor` implementation that invokes the `toString()` method of each bean as it is created by the container and prints the resulting string to the system console.

Find below the custom `BeanPostProcessor` implementation class definition:

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean,
            String beanName) throws BeansException {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean,
            String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }

}
```

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

    <lang:groovy id="messenger"
            script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
    when the above bean (messenger) is instantiated, this custom
    BeanPostProcessor implementation will output the fact to the system console
    -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the `InstantiationTracingBeanPostProcessor` is simply defined. It does not even have a name, and because it is a bean it can be dependency-injected just like any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring dynamic language support is detailed in the chapter entitled Chapter 35, *Dynamic language support*.)

The following simple Java application executes the preceding code and configuration:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger);
    }

}
```

The output of the preceding application resembles the following:

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
org.springframework.scripting.groovy.GroovyMessenger@272961
```

Example: The RequiredAnnotationBeanPostProcessor

Using callback interfaces or annotations in conjunction with a custom `BeanPostProcessor` implementation is a common means of extending the Spring IoC container. An example is Spring's `RequiredAnnotationBeanPostProcessor` - a `BeanPostProcessor` implementation that ships with the Spring distribution which ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

Customizing configuration metadata with a BeanFactoryPostProcessor

The next extension point that we will look at is the `org.springframework.beans.factory.config.BeanFactoryPostProcessor`. The semantics of this interface are similar to those of the `BeanPostProcessor`, with one major difference: `BeanFactoryPostProcessor` operates on the *bean configuration metadata*; that is, the Spring IoC container allows a `BeanFactoryPostProcessor` to read the configuration metadata and potentially change it *before* the container instantiates any beans other than `BeanFactoryPostProcessors`.

You can configure multiple `BeanFactoryPostProcessors`, and you can control the order in which these `BeanFactoryPostProcessors` execute by setting the `order` property. However, you can only set this property if the `BeanFactoryPostProcessor` implements the `Ordered` interface. If you write your own `BeanFactoryPostProcessor`, you should consider implementing the `Ordered` interface too. Consult the javadocs of the `BeanFactoryPostProcessor` and `Ordered` interfaces for more details.

> **Note**
>
> If you want to change the actual bean *instances* (i.e., the objects that are created from the configuration metadata), then you instead need to use a `BeanPostProcessor` (described above in the section called "Customizing beans using a BeanPostProcessor"). While it is technically possible to work with bean instances within a `BeanFactoryPostProcessor` (e.g., using `BeanFactory.getBean()`), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects such as bypassing bean post processing.
>
> Also, `BeanFactoryPostProcessors` are scoped *per-container*. This is only relevant if you are using container hierarchies. If you define a `BeanFactoryPostProcessor` in one container, it will *only* be applied to the bean definitions in that container. Bean definitions in one container will not be post-processed by `BeanFactoryPostProcessors` in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is executed automatically when it is declared inside an `ApplicationContext`, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as `PropertyOverrideConfigurer` and `PropertyPlaceholderConfigurer`. A custom `BeanFactoryPostProcessor` can also be used, for example, to register custom property editors.

An `ApplicationContext` automatically detects any beans that are deployed into it that implement the `BeanFactoryPostProcessor` interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

> **Note**
>
> As with BeanPostProcessors , you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a `Bean(Factory)PostProcessor`, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the `Bean(Factory)PostProcessor` will be instantiated eagerly even if you set the `default-lazy-init` attribute to `true` on the declaration of your `<beans />` element.

Example: the Class name substitution PropertyPlaceholderConfigurer

You use the `PropertyPlaceholderConfigurer` to externalize property values from a bean definition in a separate file using the standard Java `Properties` format. Doing so enables the person deploying an application to customize environment-specific properties such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a `DataSource` with placeholder values is defined. The example shows properties configured from an external `Properties` file. At runtime, a `PropertyPlaceholderConfigurer` is applied to the metadata that will replace some properties of the DataSource. The values to replace are specified as *placeholders* of the form `${property-name}` which follows the Ant / log4j / JSP EL style.

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>

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

The actual values come from another file in the standard Java `Properties` format:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

Therefore, the string `${jdbc.username}` is replaced at runtime with the value 'sa', and the same applies for other placeholder values that match keys in the properties file. The `PropertyPlaceholderConfigurer` checks for placeholders in most properties and attributes of a bean definition. Furthermore, the placeholder prefix and suffix can be customized.

With the `context` namespace introduced in Spring 2.5, it is possible to configure property placeholders with a dedicated configuration element. One or more locations can be provided as a comma-separated list in the `location` attribute.

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

The `PropertyPlaceholderConfigurer` not only looks for properties in the `Properties` file you specify. By default it also checks against the Java `System` properties if it cannot find a property in the specified properties files. You can customize this behavior by setting the `systemPropertiesMode` property of the configurer with one of the following three supported integer values:

- *never* (0): Never check system properties
- *fallback* (1): Check system properties if not resolvable in the specified properties files. This is the default.
- *override* (2): Check system properties first, before trying the specified properties files. This allows system properties to override any other property source.

Consult the `PropertyPlaceholderConfigurer` javadocs for more information.

> **Tip**
>
> You can use the `PropertyPlaceholderConfigurer` to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:
>
> ```
> <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
>     <property name="locations">
>         <value>classpath:com/foo/strategy.properties</value>
>     </property>
>     <property name="properties">
>         <value>custom.strategy.class=com.foo.DefaultStrategy</value>
>     </property>
> </bean>
>
> <bean id="serviceStrategy" class="${custom.strategy.class}"/>
> ```
>
> If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the `preInstantiateSingletons()` phase of an `ApplicationContext` for a non-lazy-init bean.

Example: the PropertyOverrideConfigurer

The `PropertyOverrideConfigurer`, another bean factory post-processor, resembles the `PropertyPlaceholderConfigurer`, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overriding `Properties` file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean definition is *not* aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple `PropertyOverrideConfigurer` instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

Properties file configuration lines take this format:

```
beanName.property=value
```

For example:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

This example file can be used with a container definition that contains a bean called *dataSource*, which has *driver* and *url* properties.

Compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In this example…

```
foo.fred.bob.sammy=123
```

i. the `sammy` property of the `bob` property of the `fred` property of the `foo` bean is set to the scalar value `123`.

> **Note**
>
> Specified override values are always *literal* values; they are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference.

With the `context` namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element:

```
<context:property-override location="classpath:override.properties"/>
```

Customizing instantiation logic with a FactoryBean

Implement the `org.springframework.beans.factory.FactoryBean` interface for objects that *are themselves factories*.

The `FactoryBean` interface is a point of pluggability into the Spring IoC container's instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own `FactoryBean`, write the complex initialization inside that class, and then plug your custom `FactoryBean` into the container.

The `FactoryBean` interface provides three methods:

- `Object getObject()`: returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
- `boolean isSingleton()`: returns `true` if this `FactoryBean` returns singletons, `false` otherwise.
- `Class getObjectType()`: returns the object type returned by the `getObject()` method or `null` if the type is not known in advance.

The `FactoryBean` concept and interface is used in a number of places within the Spring Framework; more than 50 implementations of the `FactoryBean` interface ship with Spring itself.

When you need to ask a container for an actual `FactoryBean` instance itself instead of the bean it produces, preface the bean's id with the ampersand symbol (`&`) when calling the `getBean()` method of the `ApplicationContext`. So for a given `FactoryBean` with an id of `myBean`, invoking `getBean("myBean")` on the container returns the product of the `FactoryBean`; whereas, invoking `getBean("&myBean")` returns the `FactoryBean` instance itself.

7.9 Annotation-based container configuration

> **Are annotations better than XML for configuring Spring?**
>
> The introduction of annotation-based configurations raised the question of whether this approach is 'better' than XML. The short answer is *it depends*. The long answer is that each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without