With a custom `Scope` implementation, you are not limited to programmatic registration of the scope. You can also do the `Scope` registration declaratively, using the `CustomScopeConfigurer` class:

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.support.SimpleThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

> **Note**
>
> When you place `<aop:scoped-proxy/>` in a `FactoryBean` implementation, it is the factory bean itself that is scoped, not the object returned from `getObject()`.

7.6 Customizing the nature of a bean

Lifecycle callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the former and `destroy()` for the latter to allow the bean to perform certain actions upon initialization and destruction of your beans.

> **Tip**
>
> The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring specific interfaces. For details see the section called "@PostConstruct and @PreDestroy".
>
> If you don't want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata.

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle

behavior Spring does not offer out-of-the-box, you can implement a `BeanPostProcessor` yourself. For more information, see Section 7.8, “Container Extension Points”.

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the `Lifecycle` interface so that those objects can participate in the startup and shutdown process as driven by the container’s own lifecycle.

The lifecycle callback interfaces are described in this section.

Initialization callbacks

The `org.springframework.beans.factory.InitializingBean` interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The `InitializingBean` interface specifies a single method:

```
void afterPropertiesSet() throws Exception;
```

It is recommended that you do not use the `InitializingBean` interface because it unnecessarily couples the code to Spring. Alternatively, use the `@PostConstruct` annotation or specify a POJO initialization method. In the case of XML-based configuration metadata, you use the `init-method` attribute to specify the name of the method that has a void no-argument signature. With Java config, you use the `initMethod` attribute of `@Bean`, see the section called “Receiving lifecycle callbacks”. For example, the following:

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }

}
```

…is exactly the same as…

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }

}
```

but does not couple the code to Spring.

Destruction callbacks

Implementing the `org.springframework.beans.factory.DisposableBean` interface allows a bean to get a callback when the container containing it is destroyed. The `DisposableBean` interface specifies a single method:

```
void destroy() throws Exception;
```

It is recommended that you do not use the `DisposableBean` callback interface because it unnecessarily couples the code to Spring. Alternatively, use the `@PreDestroy` annotation or specify a generic method that is supported by bean definitions. With XML-based configuration metadata, you

use the `destroy-method` attribute on the `<bean/>`. With Java config, you use the `destroyMethod` attribute of `@Bean`, see the section called “Receiving lifecycle callbacks”. For example, the following definition:

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

but does not couple the code to Spring.

> **Tip**
>
> The `destroy-method` attribute of a `<bean>` element can be assigned a special `(inferred)` value which instructs Spring to automatically detect a public `close` or `shutdown` method on the specific bean class (any class that implements `java.lang.AutoCloseable` or `java.io.Closeable` would therefore match). This special `(inferred)` value can also be set on the `default-destroy-method` attribute of a `<beans>` element to apply this behavior to an entire set of beans (see the section called “Default initialization and destroy methods”). Note that this is the default behavior with Java config.

Default initialization and destroy methods

When you write initialization and destroy method callbacks that do not use the Spring-specific `InitializingBean` and `DisposableBean` callback interfaces, you typically write methods with names such as `init()`, `initialize()`, `dispose()`, and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to `look` for named initialization and destroy callback method names on *every* bean. This means that you, as an application developer, can write your application classes and use an initialization callback called `init()`, without having to configure an `init-method="init"` attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named `init()` and destroy callback methods are named `destroy()`. Your class will resemble the class in the following example.

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }

}
```

```
<beans default-init-method="init">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

The presence of the `default-init-method` attribute on the top-level `<beans/>` element attribute causes the Spring IoC container to recognize a method called `init` on beans as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You configure destroy method callbacks similarly (in XML, that is) by using the `default-destroy-method` attribute on the top-level `<beans/>` element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name using the `init-method` and `destroy-method` attributes of the <bean/> itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created *first*, *then* an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the init method, because doing so would couple the lifecycle of the target bean with its proxy/interceptors and leave strange semantics when your code interacts directly to the raw target bean.

Combining lifecycle mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior: the `InitializingBean` and `DisposableBean` callback interfaces; custom `init()` and `destroy()` methods; and the `@PostConstruct` and `@PreDestroy` annotations. You can combine these mechanisms to control a given bean.

> **Note**
>
> If multiple lifecycle mechanisms are configured for a bean, and each mechanism is configured with a different method name, then each configured method is executed in the order listed below. However, if the same method name is configured - for example, `init()` for an initialization

> method - for more than one of these lifecycle mechanisms, that method is executed once, as explained in the preceding section.

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

- Methods annotated with `@PostConstruct`
- `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
- A custom configured `init()` method

Destroy methods are called in the same order:

- Methods annotated with `@PreDestroy`
- `destroy()` as defined by the `DisposableBean` callback interface
- A custom configured `destroy()` method

Startup and shutdown callbacks

The `Lifecycle` interface defines the essential methods for any object that has its own lifecycle requirements (e.g. starts and stops some background process):

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();

}
```

Any Spring-managed object may implement that interface. Then, when the `ApplicationContext` itself receives start and stop signals, e.g. for a stop/restart scenario at runtime, it will cascade those calls to all `Lifecycle` implementations defined within that context. It does this by delegating to a `LifecycleProcessor`:

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();

    void onClose();

}
```

Notice that the `LifecycleProcessor` is itself an extension of the `Lifecycle` interface. It also adds two other methods for reacting to the context being refreshed and closed.

> **Tip**
>
> Note that the regular `org.springframework.context.Lifecycle` interface is just a plain contract for explicit start/stop notifications and does NOT imply auto-startup at context refresh time. Consider implementing `org.springframework.context.SmartLifecycle` instead for fine-grained control over auto-startup of a specific bean (including startup phases). Also, please note that stop notifications are not guaranteed to come before destruction: On regular shutdown,

> all `Lifecycle` beans will first receive a stop notification before the general destruction callbacks are being propagated; however, on hot refresh during a context's lifetime or on aborted refresh attempts, only destroy methods will be called.

The order of startup and shutdown invocations can be important. If a "depends-on" relationship exists between any two objects, the dependent side will start *after* its dependency, and it will stop *before* its dependency. However, at times the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the `SmartLifecycle` interface defines another option, namely the `getPhase()` method as defined on its super-interface, `Phased`.

```
public interface Phased {

    int getPhase();

}
```

```
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);

}
```

When starting, the objects with the lowest phase start first, and when stopping, the reverse order is followed. Therefore, an object that implements `SmartLifecycle` and whose `getPhase()` method returns `Integer.MIN_VALUE` would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of `Integer.MAX_VALUE` would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it's also important to know that the default phase for any "normal" `Lifecycle` object that does not implement `SmartLifecycle` would be 0. Therefore, any negative phase value would indicate that an object should start before those standard components (and stop after them), and vice versa for any positive phase value.

As you can see the stop method defined by `SmartLifecycle` accepts a callback. Any implementation *must* invoke that callback's `run()` method after that implementation's shutdown process is complete. That enables asynchronous shutdown where necessary since the default implementation of the `LifecycleProcessor` interface, `DefaultLifecycleProcessor`, will wait up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named "lifecycleProcessor" within the context. If you only want to modify the timeout, then defining the following would be sufficient:

```
<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">
    <!-- timeout value in milliseconds -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

As mentioned, the `LifecycleProcessor` interface defines callback methods for the refreshing and closing of the context as well. The latter will simply drive the shutdown process as if `stop()` had been called explicitly, but it will happen when the context is closing. The 'refresh' callback on the other hand enables another feature of `SmartLifecycle` beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback will be invoked, and at that point the default lifecycle processor will check the boolean value returned by each `SmartLifecycle` object's

`isAutoStartup()` method. If "true", then that object will be started at that point rather than waiting for an explicit invocation of the context's or its own `start()` method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The "phase" value as well as any "depends-on" relationships will determine the startup order in the same way as described above.

Shutting down the Spring IoC container gracefully in non-web applications

> **Note**
>
> This section applies only to non-web applications. Spring's web-based `ApplicationContext` implementations already have code in place to shut down the Spring IoC container gracefully when the relevant web application is shut down.

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. Of course, you must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, you call the `registerShutdownHook()` method that is declared on the `ConfigurableApplicationContext` interface:

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

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

        ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext(
                new String []{"beans.xml"});

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...

    }
}
```

ApplicationContextAware and BeanNameAware

When an `ApplicationContext` creates an object instance that implements the `org.springframework.context.ApplicationContextAware` interface, the instance is provided with a reference to that `ApplicationContext`.

```
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;

}
```

Thus beans can manipulate programmatically the `ApplicationContext` that created them, through the `ApplicationContext` interface, or by casting the reference to a known subclass of this interface, such as `ConfigurableApplicationContext`, which exposes additional functionality. One use would be the programmatic retrieval of other beans. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the

`ApplicationContext` provide access to file resources, publishing application events, and accessing a `MessageSource`. These additional features are described in Section 7.15, “Additional Capabilities of the ApplicationContext”

As of Spring 2.5, autowiring is another alternative to obtain reference to the `ApplicationContext`. The "traditional" `constructor` and `byType` autowiring modes (as described in the section called “Autowiring collaborators”) can provide a dependency of type `ApplicationContext` for a constructor argument or setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the `ApplicationContext` is autowired into a field, constructor argument, or method parameter that is expecting the `ApplicationContext` type if the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see the section called “@Autowired”.

When an `ApplicationContext` creates a class that implements the `org.springframework.beans.factory.BeanNameAware` interface, the class is provided with a reference to the name defined in its associated object definition.

```
public interface BeanNameAware {

    void setBeanName(string name) throws BeansException;

}
```

The callback is invoked after population of normal bean properties but before an initialization callback such as `InitializingBean` *afterPropertiesSet* or a custom init-method.

Other Aware interfaces

Besides `ApplicationContextAware` and `BeanNameAware` discussed above, Spring offers a range of `Aware` interfaces that allow beans to indicate to the container that they require a certain *infrastructure* dependency. The most important `Aware` interfaces are summarized below - as a general rule, the name is a good indication of the dependency type:

Table 7.4. Aware interfaces

Name	Injected Dependency	Explained in…
`ApplicationContextAware`	Declaring `ApplicationContext`	the section called “ApplicationContextAware and BeanNameAware”
`ApplicationEventPublisherAware`	Event publisher of the enclosing `ApplicationContext`	Section 7.15, “Additional Capabilities of the ApplicationContext”
`BeanClassLoaderAware`	Class loader used to load the bean classes.	the section called “Instantiating beans”
`BeanFactoryAware`	Declaring `BeanFactory`	the section called “ApplicationContextAware and BeanNameAware”
`BeanNameAware`	Name of the declaring bean	the section called “ApplicationContextAware and BeanNameAware”

Name	Injected Dependency	Explained in…
`BootstrapContextAware`	Resource adapter `BootstrapContext` the container runs in. Typically available only in JCA aware ApplicationContexts	Chapter 32, *JCA CCI*
`LoadTimeWeaverAware`	Defined *weaver* for processing class definition at load time	the section called “Load-time weaving with AspectJ in the Spring Framework”
`MessageSourceAware`	Configured strategy for resolving messages (with support for parametrization and internationalization)	Section 7.15, “Additional Capabilities of the ApplicationContext”
`NotificationPublisherAware`	Spring JMX notification publisher	Section 31.7, “Notifications”
`PortletConfigAware`	Current `PortletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 25, *Portlet MVC Framework*
`PortletContextAware`	Current `PortletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 25, *Portlet MVC Framework*
`ResourceLoaderAware`	Configured loader for low-level access to resources	Chapter 8, *Resources*
`ServletConfigAware`	Current `ServletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 22, *Web MVC framework*
`ServletContextAware`	Current `ServletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 22, *Web MVC framework*

Note again that usage of these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As such, they are recommended for infrastructure beans that require programmatic access to the container.

7.7 Bean definition inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.