Spring Bean Life Cycle

Spring bean factory is responsible for managing the life cycle of beans created through spring container. The life cycle of beans consist of **call back methods** which can be categorized broadly in two groups:

- Post initialization call back methods
- Pre destruction call back methods

Spring framework provides following **4 ways for controlling life cycle events** of bean:

1. InitializingBean and DisposableBean callback interfaces
2. Other Aware interfaces for specific behavior
3. custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

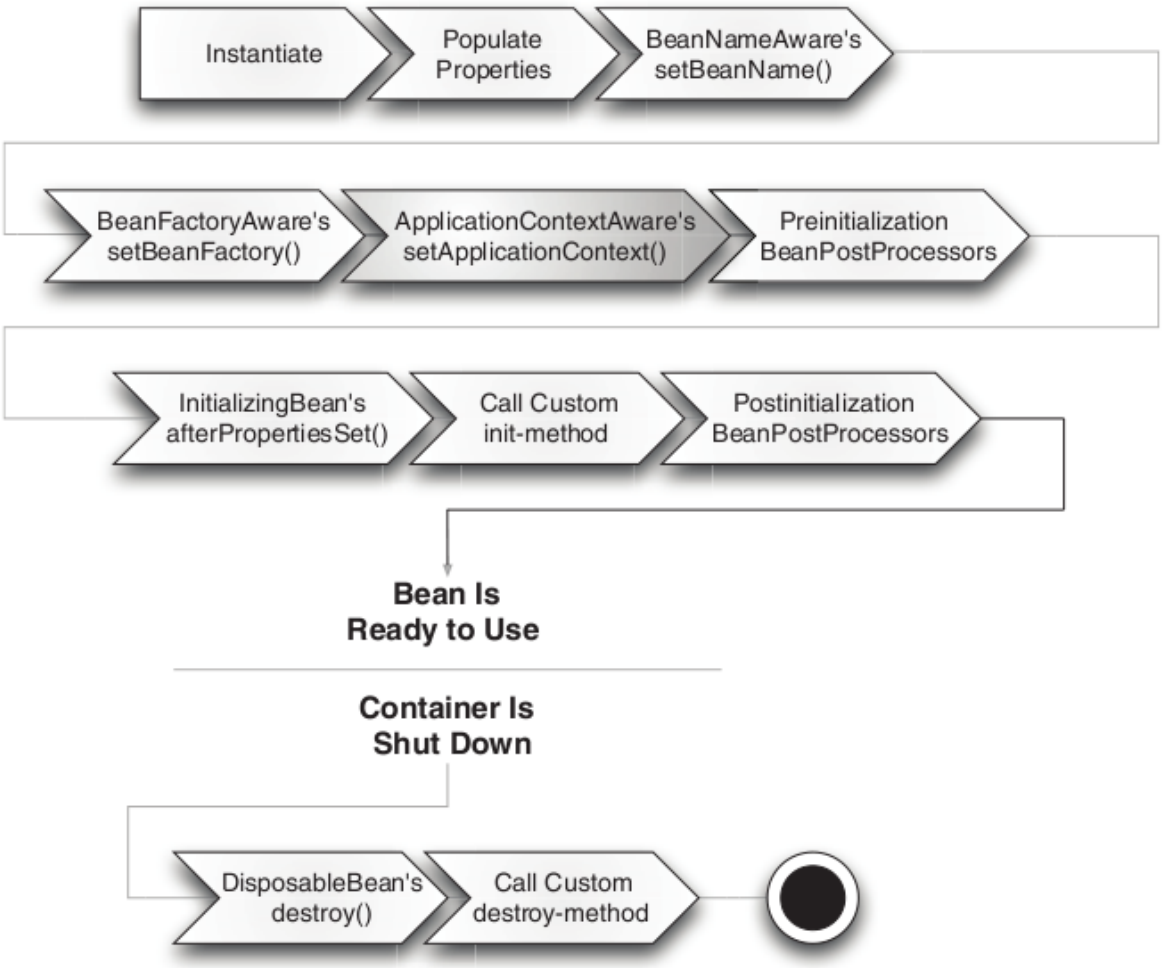

Figure 2.3 The lifecycle of a bean in a Spring application context extends the lifecycle of a factory-contained bean by adding a step to make the bean application context aware.

Spring Bean Life Cycle

Lets learn about them one by one.

InitializingBean and DisposableBean callback interfaces

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;
```

This is not a preferrable way to initialize the bean because it tightly couple your bean class with spring container. A better approach is to use "init-method" attribute in bean definition in applicationContext.xml file.

Similarly, implementing the org.springframework.beans.factory.DisposableBeaninterface 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;
```

A sample bean implementing above interfaces would look like this:

```
package com.howtodoinjava.task;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class DemoBeanTypeOne implements InitializingBean, DisposableBean
{
 //Other bean attributes and methods

 @Override
 public void afterPropertiesSet() throws Exception
 {
  //Bean initialization code
 }

 @Override
 public void destroy() throws Exception
 {
  //Bean destruction code
 }
}
```

Other Aware interfaces for specific behavior

Spring offers a range of Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean.

These interfaces can be summarized as :

Aware interface	Method to override	Purpose
ApplicationContextAware	void setApplicationContext(ApplicationContext applicationContext) throws BeansException;	Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in.
ApplicationEventPublisherAware	void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher);	Set the ApplicationEventPublisher that this object runs in.
BeanClassLoaderAware	void setBeanClassLoader(ClassLoader classLoader);	Callback that supplies the bean class loader to a bean instance.
BeanFactoryAware	void setBeanFactory(BeanFactory beanFactory) throws BeansException;	Callback that supplies the owning factory to a bean instance.
BeanNameAware	void setBeanName(String name);	Set the name of the bean in the bean factory that created this bean.
BootstrapContextAware	void setBootstrapContext(BootstrapContext bootstrapContext);	Set the BootstrapContext that this object runs in.
LoadTimeWeaverAware	void setLoadTimeWeaver(LoadTimeWeaver loadTimeWeaver);	Set the LoadTimeWeaver of this object's containing ApplicationContext.

MessageSourceAware	void setMessageSource(MessageSource messageSource);	Set the MessageSource that this object runs in.
NotificationPublisherAware	void setNotificationPublisher(NotificationPublisher notificationPublisher);	Set the NotificationPublisher instance for the current managed resource instance.
PortletConfigAware	void setPortletConfig(PortletConfig portletConfig);	Set the PortletConfig this object runs in.
PortletContextAware	void setPortletContext(PortletContext portletContext);	Set the PortletContext that this object runs in.
ResourceLoaderAware	void setResourceLoader(ResourceLoader resourceLoader);	Set the ResourceLoader that this object runs in.
ServletConfigAware	void setServletConfig(ServletConfig servletConfig);	Set the ServletConfig that this object runs in.
ServletContextAware	void setServletContext(ServletContext servletContext);	Set the ServletContext that this object runs in.

A sample implementation will look like this:

```
package com.howtodoinjava.task;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanClassLoaderAware;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.context.MessageSource;
import org.springframework.context.MessageSourceAware;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.context.weaving.LoadTimeWeaverAware;
import org.springframework.core.io.ResourceLoader;
import org.springframework.instrument.classloading.LoadTimeWeaver;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

public class BemoBeanTypeTwo implements ApplicationContextAware,
    ApplicationEventPublisherAware, BeanClassLoaderAware,
BeanFactoryAware,
    BeanNameAware, LoadTimeWeaverAware, MessageSourceAware,
    NotificationPublisherAware, ResourceLoaderAware
{
  @Override
  public void setResourceLoader(ResourceLoader arg0) {
    // TODO Auto-generated method stub
  }

  @Override
  public void setNotificationPublisher(NotificationPublisher arg0) {
    // TODO Auto-generated method stub

  }

  @Override
  public void setMessageSource(MessageSource arg0) {
    // TODO Auto-generated method stub
  }

  @Override
```

```
 public void setLoadTimeWeaver(LoadTimeWeaver arg0) {
  // TODO Auto-generated method stub
 }

 @Override
 public void setBeanName(String arg0) {
  // TODO Auto-generated method stub
 }

 @Override
 public void setBeanFactory(BeanFactory arg0) throws BeansException {
  // TODO Auto-generated method stub
 }

 @Override
 public void setBeanClassLoader(ClassLoader arg0) {
  // TODO Auto-generated method stub
 }

 @Override
 public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {
  // TODO Auto-generated method stub
 }

 @Override
 public void setApplicationContext(ApplicationContext arg0)
    throws BeansException {
  // TODO Auto-generated method stub
 }
}
```

Custom init() and destroy() methods in bean configuration file

The default init and destroy methods in bean configuration file can be defined in two ways:

- Bean local definition applicable to a single bean
- Global definition applicable to all beans defined in beans context

Local definition is given as below.

```
<beans>

  <bean id="demoBean" class="com.howtodoinjava.task.DemoBean" init-method="
customInit" destroy-method="customDestroy"></bean>

</beans>
```

Where as global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

```
<beans default-init-method="customInit" default-destroy-method="customDestroy
">

        <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bea
n>

</beans>
```

A sample implementation for this type of life cycle will be:

```
package com.howtodoinjava.task;

public class BemoBeanTypeThree
{
  public void customInit()
  {
    System.out.println("Method customInit() invoked...");
  }

  public void customDestroy()
  {
    System.out.println("Method customDestroy() invoked...");
  }
}
```

@PostConstruct and @PreDestroy annotations

Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.

- @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it's instance is returned to requesting object.
- @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.

A sample implemetation will look like this:

```
package com.howtodoinjava.task;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class BemoBeanTypeFour
{
  @PostConstruct
  public void customInit()
  {
    System.out.println("Method customInit() invoked...");
  }

  @PreDestroy
  public void customDestroy()
  {
    System.out.println("Method customDestroy() invoked...");
  }
}
```

So this is all about life cycle management of beans inside spring container. I hope that it has added some more knowledge in your kitty.

Spring IOC Container

Spring provides two type of Spring IOC container : BeanFactory and ApplicationContext.

Bean Factory

- Bean instantiation/wiring

Application Context

- Bean instantiation/wiring
- Automatic BeanPostProcessor registration
- Automatic BeanFactoryPostProcessor registration

- Convenient MessageSource access (for i18n)
- ApplicationEvent publication

What are Java Annotations:

- Java Annotations are metadata which are adding to Java elements. An annotation indicates that the declared element should be processed in some special way by a compiler, development tool, deployment tool, or during runtime.