Spring Bean Life Cycle Tutorial

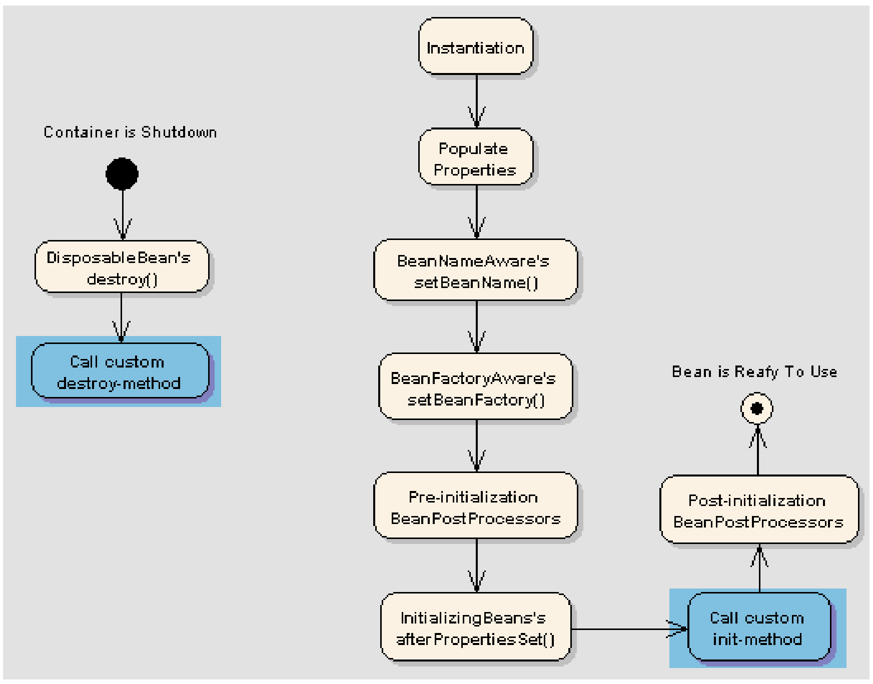
May 7, 2013 by Lokesh Gupta

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


Instantiation
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBeans's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is Reafy To Use
Container is Shutdown
DisposableBean's destroy()
Call custom destroy-method

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.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;
```

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="c
</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"></bean>
</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 implementation 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.

Happy Learning !!

About Lokesh Gupta

Founded HowToDoInJava.com in late 2012. I love computers, programming and solving problems everyday. A family guy with fun loving nature. You can find me on Facebook, Twitter and Google Plus.

Feedback, Discussion and Comments

Sreepad

March 3, 2017 at 2:53 pm

Hi Lokesh, Please tell me which are the default bean life cycle methods called by the Spring container ?

Reply

Sreepad

March 3, 2017 at 11:53 am

Hi Lokesh, I have one doubt. I have created a small project with 1 bean class, main class and context.xml file. What are the default interfaces implemented by bean behind the scenes if i wont implement any of them manually? Will it implements all the interfaces listed above or any particular mandatory interfaces it will implement ? Please help me in this, as i am stuck in this doubt without moving forward.

Reply

Lokesh Gupta

March 3, 2017 at 12:11 pm

By default, a bean does not implement any of above interfaces. Above interfaces are used for executing code blocks on specific lifecycle event on a bean, and so these are used as and when needed only.

Reply

Sreepad

March 3, 2017 at 1:33 pm

Thanks a lot Lokesh. I am clear now, as u said they are used when needed only. But,one doubt still persists. Spring container handles each and every beans life cycle defined in context.xml file. What are those mandatory life cycle methods(and these methods belongs to which interfaces ?) called by the Spring container. Example :- I have a single bean class, context file and Test class, and when i run Test class, which are the default methods

called on the bean.
Please bear my questions, which would help me lot :-).

Reply

Lokesh Gupta
March 3, 2017 at 4:55 pm

By default, you can have two methods "init-method" and "destroy-method". These methods then must be defined inside bean class. Spring framework internally uses reflection to invoke these methods.

Reply

Tofek khan
February 8, 2016 at 8:19 pm

Nice Explanation. But could you Please discuss more about "Other Aware Interfaces…" and implementation of all these type.

Thanks
Tofek

Reply

Lokesh Gupta
February 9, 2016 at 6:29 am

Sure.. I will cover them in future.

Reply

Shao Yong
March 25, 2017 at 6:27 pm

You should better look at Spring API docs

Reply

Yash
July 9, 2015 at 10:50 am

Hi,
Thanks for the above information .It is really helpful.

I had a requirement in my project, need to run the database scripts through liquibase. There is existing framework to check the change logs and run the change logs . So, just bean definition for this java class is enough to run the liquibase script changes for the entire project . This i have done as below
<bean id = "lbase" class="com.*.*.**.<> datasource="datasource" />

Now, I had another webservice client configuration class which handles the code to read the webservice url and username from database.And the table will be created and loaded with data by the above defined liquibase bean. I have defined this webservice configuration bean as

The order of defining in spring context file is firstly,liquibase related bean and then wsClientConfig bean is defined.
now, the doubt is -among these two, which bean is initialized first and which is next?
For me, error is coming as- no table or view exists while fetching the details of webservice fails.

Please help in handling this scenario. Thank you

Reply

Lokesh
July 9, 2015 at 11:00 am

Use "**depends-on**" attribute.

Reply

Yash
July 9, 2015 at 12:46 pm

I tried using depends-on but it still caused the same error.problem. after using depends-on, all the liquibase bean initialization is not completed before wsClientConfig bean ?

Reply

Lokesh
July 9, 2015 at 12:57 pm

Try this. Implement interface "InitializingBean" in wsClientConfig bean. And write your logic inside afterPropertiesSet() method.

Reply

KIRAN
July 8, 2015 at 6:03 pm

I have a doubt.Spring works on the top of the underline java its means JVM .Why spring beans required one more life cycle it means that already jvm will have a life cycle

Reply

Lokesh

July 8, 2015 at 6:43 pm

JVM does not manage it's bean's lifecyle. It's container for all objects, and it checks which objects are live or dead. JVM never tries to modify the state or behavior of an object. objects are just created inside JVM and cleanup by it when they are no longer referred from other objects. That's it !!

Spring bean life cycle is different thing. Here you want to change the state/behavior of bean (or application) whenever an object get's created or destroyed, or in some specific events. Please note that these events are not JVM triggered events. They are simple "callback methods" when spring's IOC framework create those bean instances or destroy them.

Reply

prakash mishra

February 16, 2015 at 6:38 pm

Hi, Lokesh Thanks a lot for this meaningful blog . i have one question which is related to beans life cycle i understood when public void afterPropertiesSet() has called but i have one confusion when public void destroy() will call and who call public void destroy() . and who is responsible to destroy spring bean?

Reply

Lokesh

February 17, 2015 at 10:32 am

It's spring IoC container classes which call destroy() method.

Reply

Ram

July 10, 2014 at 4:41 am

Hai am very new to spring .I am getting some problem in Spring bean lifecycle methods.Like
the out put is
constructor
setId
setName

```
This is init-method
Jul 10, 2014 9:58:35 AM org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@2a5330: defining beans [st]; root of
factory hierarchy
constructor
setId
setName
before
This is init-method
after
This is bussinessmethod
```

means constructor and setters methods are called two times.what happen will anyone suggest me

Reply

Lokesh

July 10, 2014 at 7:14 am

The reason is – spring aop is proxy-based, i.e. the container creates "raw" bean instance and proxy for it (created at runtime via subclassing original bean class) if the bean is affected by AOP rules. So, when proxy object is created, it calls superclass constructor/setters and it looks as two instances are created.

You notice that "before" and "after" methods are not called because they are run only when actual instance is created. It's contract provided by Spring's annotations @PostConstruct and @PreDestroy; OR "init-method" and "destroy-method" attributes of the bean element in configuration file.

Reply

Ram

July 13, 2014 at 3:42 pm

Thanks for your reply.I understand well.Small thing is here am not using Aop concept.i just tries it by implenting BeanPostProcessor interface.And one more thing is if i add bean scope is "singleton" then BeforeInitialization and afterInitialization methods are not called.Plz explain me abt is there any internal relation between bean scopes and BeanPostProcessor interface methods. in BeanLife management.plz reply me.

Reply

sam

October 10, 2013 at 6:53 pm

Hi, Thanks for your blog. I have one requirement in my current project like i need to start a new thread in my spring application. In that thread i need to call one of the beans methods. Would you please suggest.

Reply

Lokesh Gupta

October 10, 2013 at 10:01 pm

If the object (thread instance) that needs access to the container is a bean in the container, just implement the BeanFactoryAware or ApplicationContextAware interfaces.

If an object outside the container needs access to the container, you must use singleton pattern for the spring container. Create this singleton bean in application startup, and set the application context reference there in a variable. Now whenever you need a bean, get the context via singleton class, and lookup the bean from application context.

Reply

Santosh Srivastava

October 5, 2013 at 5:44 pm

Hi, Thanks a lot for this meaningful blog, I have in a problem please help me to solve it.
Problem: How to prevent serialization in java if the class has implemented Serializable interface. Problem might be like this,
Class A implements Seralizable{
}
class B extends A{
}

How to prevent class B to serialize if it have behavior of serialization if someone going to serialize. Please give me innovative solution.

Hope i will hear you soon.

Regard's
Santosh Srivastava

Reply

Lokesh Gupta

October 5, 2013 at 8:03 pm

Not possible. Reason is that in java super class can not see the internals of child classes. Only child classes can access super class's fields and methods. Anyway, what kind of design is this in which super class is serializable but child classes are not?? Please refer to "Liskov's Substitution Principle" in http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Anyways, you have following options:

1) Make super class final so that no one can extend it? [I know it is not what you expect].
2) Make super class's field transient, so that if B is serialized then it will have only its state saved.

Anybody came across this conversation, feel free to engage. :)

Reply

anurag
September 3, 2014 at 1:02 pm

In Java, if the super class of a class is implementing Serializable interface, it means that it is already serializable. Since, an interface cannot be unimplemented, it is not possible to make a class non-serializable. However, the serialization of a new class can be avoided. For this, writeObject () and readObject() methods should be implemented in your class so that a Not Serializable Exception can be thrown by these methods. And, this can be done by customizing the Java Serialization process. Below the code that demonstrates it

```
class MySubClass extends SomeSerializableSuperClass
{
    private void writeObject(java.io.ObjectOutputStream out) throws IOException {
    throw new NotSerializableException("Can not serialize this class");
    }
    private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException
        throw new NotSerializableException("Can not serialize this class");
    }
    private void readObjectNoData() throws ObjectStreamException; {
        throw new NotSerializableException("Can not serialize this class");
    }
}
```

Reply

Lokesh
September 3, 2014 at 1:14 pm

Hi Anurag, Thanks for providing a good solution for problem in hand. Seems logical to me as well.

Reply

Singham
September 25, 2013 at 2:24 pm

Hi
thnx for the post
can u tell me how bean factory will come to know (incase of global) .which class hold init and destroy method

Reply

Lokesh Gupta

September 25, 2013 at 10:04 pm

It checks via reflection.

Reply

Ask Questions & Share Feedback

Your email address will not be published. Required fields are marked *

Comment

*Want to Post Code Snippets or XML content? Please use **[java]** ... **[/java]** tags otherwise code may not appear partially or even fully. e.g.

```
[java]
public static void main (String[] args) {
...
}
[/java]
```

Name *

Email *

Website

Help me fight spammers. Solve this simple math. *

+ 1 = 7

POST COMMENT

Search Tutorials

Type and Press ENTER...

Spring Core Tutorial

Recommended

10 Life Lessons
Secure Hash Algorithms
Regular Expressions
How Web Servers work?
How Java I/O Works Internally?
Unit Testing Best Practices
Exception Handling Best Practices
Best Way to Learn Java
Java Best Practices
Java Interview Questions
REST API Tutorial

Developer Tools

JSON Formatter and Minifier
XML Formatter and Minifier
CSS Formatter and Minifier
HTML Formatter and Minifier

Meta Links

Advertise
Contact Us
Privacy policy
About Me

References

Java 8 API
Spring Framework Reference
RESTEasy Reference
Hibernate User Guide
Junit Wiki
Maven FAQs