**Spring Bean Life Cycle**

We now know that the spring container creates beans and manages the dependencies. However, the steps involved between requesting a bean and handing it over to the requester is a long but interesting sequence of events.

http://www.tutorialscorner.com/img/Spring-bean-lifecycle.png

* **Instantiate** - The bean container creates an instance of the bean class.
* **Populate properties** - If the created bean has any properties then they are set up. If any of the property itself is a bean it is resolved and set.
* **Set Bean Name** - If the bean class implements the BeanNameAware interface then the method setBeanName(String name) is called on the created bean.
* **Set Bean Factory** - If the bean class implements the BeanFactoryAware interface then the setBeanFactory(BeanFactory beanFactory) method is called.
* **Pre initialization BeanPostProcessor** - If any BeanPostProcessors is associated with the beanFactory then the postProcessBeforeInitialization() method is called.
* If the bean class implements the interface InitializingBean then the method afterPropertiesSet() is called. Custom init method - If the bean definition in the xml configuration file contains an attribute 'init-method' then the method name provided in the value will be called.
* **Post initialization BeanPostProcessor** - If any BeanPostProcessors is associated with the beanFactory then the postProcessAfterInitialization() method is called.

Besides the above we have another step, the destroy-method, in bean lifecycle. If the bean definition in xml configuration file has an attribute destroy-method then the destroy method is called just before the bean is removed from the container.

## **InitializingBean and DisposbleBean callback interfaces**

* InitalizingBean interface is defined under org.springframework.beans.factory package and declares a single method where we can  be used to add any initialization related code. Any bean implementing InitalizingBean needs to provide an implementation of afterPropertiesSet() method. Signature of method is:

void afterPropertiesSet() throws Exception;

* Similarly DisposableBean interface is defined under the org.springframework.beans.factory and declares a single method which gets executed when bean is destroyed and can be used to add any cleanup related code. Any bean implementing DisposableBean needs to provide an implementation of destroy() method. Signature of method is :

void destroy() throws Exception;

This approach is simple to use but it’s not recommended because it will create tight coupling with the Spring framework in our bean implementations.



### **Example-**

Lets write an example to implement InitalizingBean and DisposableBean interface

**Solution:**

a) Write a PersonBean which implements InitializingBean and DisposableBean interface like below

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class PersonBean implements InitializingBean,DisposableBean{

private String name;

public PersonBean()

{

System.out.println("Constructor of person bean is called !! ");

}

@Override

public void destroy() throws Exception

{

System.out.println("destroy method of person bean is called !! ");

}

@Override

public void afterPropertiesSet() throws Exception

{

System.out.println("afterPropertiesSet method of person bean is called !! ");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

b) Create a beans.xml file in src directory to define the PersonBean

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="personBean" class="PersonBean" >

<property name="name" value="Dummy Person"/>

</bean>

</beans>

c) Create TestPersonBean class which will just loads the beans.xml and test the person bean life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestPersonBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

PersonBean bean = (PersonBean)context.getBean("personBean");

System.out.println(bean.getName());

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d)  Run the Program

You will see below output. Initialization and Destroy methods are getting called.

## Sample program output

## **Bean Name, Factory, Application context Aware interfaces**

Several times functionality requires infrastructure or we can say application context information in a bean. To achieve such functionalities ,Spring framework  provides  a range of Aware interfaces Each interface requires us to implement a method to inject the dependency in bean. Most commonly used are –

* **BeanFactoryAware** - This interface provides setBeanFactory() method  that supplies the owning bean factory instance to the bean. Signature of the method is

void setBeanFactory(BeanFactory beanFactory) throws BeansException

* **BeanNameAware**- This interface provides setBeanName() method which sets the name of the bean in the bean factory that created this bean. Signature of the method is-

void setBeanName(String name);

* **ApplicationContextAware** -This interface provides setApplicationContext() method  that supplies the owning application context instance to the bean. Signature of the method is

void setApplicationContext(ApplicationContext applicationContext) throws BeansException

### **Example**

Lets write an example to implement Aware interfaces

**Solution:**

a) Create  a class (AwareBean) which implements ApplicationContextAware, BeanNameAware and BeanFactoryAware

import java.util.Arrays;

import org.springframework.beans.BeansException;

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;

public class AwareBean implements ApplicationContextAware,BeanNameAware,BeanFactoryAware{

@Override

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

System.out.println("setBeanFactory method of Aware bean is called");

System.out.println("setBeanFactory:: Aware bean singleton="

+ beanFactory.isSingleton("awareBean"));

}

@Override

public void setBeanName(String beanName) {

System.out.println("setBeanName method of Aware bean is called");

System.out.println("setBeanName:: Bean Name defined in context="

+ beanName);

}

@Override

public void setApplicationContext(ApplicationContext applicationContext)

throws BeansException {

System.out.println("setApplicationContext method of Aware bean is called");

System.out.println("setApplicationContext:: Bean Definition Names="

+ Arrays.toString(applicationContext.getBeanDefinitionNames()));

}

}

**b)** Create a beans.xml file in src directory to define the AwareBean

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="awareBean" class="AwareBean" >

</bean>

</beans>

c)  Create TestAwareBean class which will just loads the beans.xml and test the aware life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAwareBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

AwareBean bean = (AwareBean)context.getBean("awareBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d) Run the Program

You will see below output.

Sample Program output

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

Implementing InitalizingBean and DisposableBean interface is simple to use but create tight coupling with the Spring framework in our bean implementations.

Alternatively we can **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

Note: Both post-init and pre-destroy methods should have no arguments but they can throw Exceptions

<beans>

<bean id="bean\_id" class="bean.class"

init-method="customInitmethod"

destroy-method="customDestroymethod">

</bean>

</beans>

We can configure the default init-method  and destroy-method which will be applied on all the beans .They are useful when we have a pattern of defining common method names such as init() and destroy() for all your beans consistently.

<beans default-init-method=”customDefaultInitMethod” default-destroy-method=”customDefaultDestroyMethod” >

<bean id="bean\_id" class="bean.class" >

</bean>

</beans>

### **Example**

Write and example to show the init-method and destroy-method

**Solution**

a)Write a class CustomLifeCycleMehodBean

public class CustomLifeCycleMethodBean {

private String name;

public CustomLifeCycleMethodBean()

{

System.out.println("Constructor of  bean is called !! ");

}

public void customDestroy() throws Exception {

System.out.println("custom destroy method of  bean is called !! ");

}

public void customInit() throws Exception {

System.out.println("custom Init  method of  bean is called !! ");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

b) Create a beans.xml file in src directory to define the CustomMethodLifeCycleBean

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="customLifeCycleMethodBean"

class="CustomLifeCycleMethodBean"

init-method="customInit"

destroy-method="customDestroy">

<property name="name" value="custom methods bean" ></property>

</bean>

</beans>

c) Create TestCustomMethodLifeCycleBean class which will just loads the beans.xml and test the custom methods life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCustomMethodLifeCycleBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

CustomLifeCycleMethodBean bean = (CustomLifeCycleMethodBean)context.getBean("customLifeCycleMethodBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d)Run the Program

You will see below output and custom life cycle methods are getting called

## output to show custom life cycle methods are being called

### **Example**

Write an example to demonstrate global init and destroy methods

**Solution**

a)Write a class CustomGlobalLifeCycleMehodBean

public class CustomGlobalLifeCycleMehodBean {

public CustomGlobalLifeCycleMehodBean()

{

System.out.println("Constructor of  bean is called !! ");

}

public void globalCustomDestroy() throws Exception {

System.out.println("global custom destroy method of  bean is called !! ");

}

public void globalCustomInit() throws Exception {

System.out.println("global custom Init  method of  bean is called !! ");

}

}

**b)** Create a beans.xml file in src directory to define the CustomGlobalMethodLifeCycleBean

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"

default-init-method="globalCustomInit"

default-destroy-method="globalCustomDestroy">

<bean id="customGlobalLifeCycleMethodBean"

class="CustomGlobalLifeCycleMehodBean" />

</beans>

c) Create TestCustomMethodLifeCycleBean class which will just loads the beans.xml and test the custom methods life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCustomGlobalMethodLifeCycleBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

CustomGlobalLifeCycleMehodBean bean = (CustomGlobalLifeCycleMehodBean)context.getBean("customGlobalLifeCycleMethodBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

**Spring Bean Scope:**

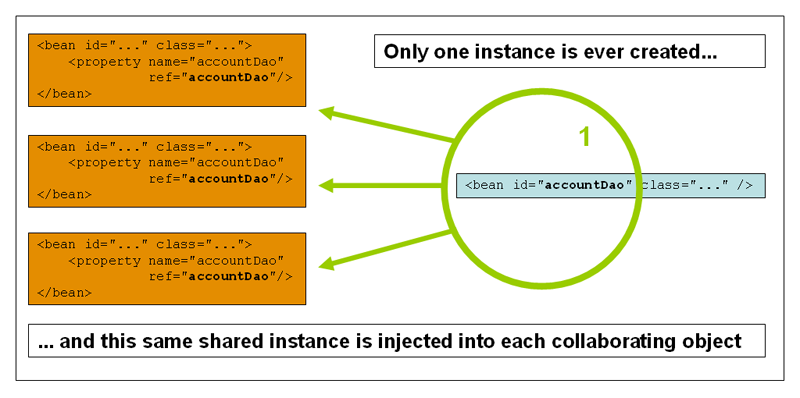
**Spring framework supports following five scopes**. Out of which three scopes are supported only in web ApplicationContext.

1. Singleton
2. Prototype
3. Request
4. Session
5. Global Session

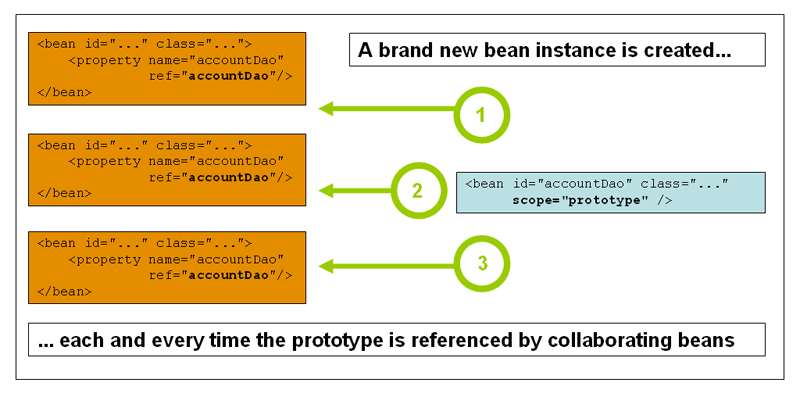
**Note:**Request, Session and Global Session scopes are valid in the context of a web-aware Spring ApplicationContext. This means that you can only use these scoped beans in a an application deployed to a web server. Spring can be used in applications that run in standard JVMs along with applications that run in servlet containers (Tomcat, etc). Request, Session and Global session however, only exists in web servers so it has no meaning if the application is running in a standard desktop environment

## Singleton and Prototype:

**The Singleton scopes the bean definition to a single instance per Spring IoC container (default)**. If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. You have to carefully understand that it is single for its own IoC container, not the JVM or your entire application. Because your application may have more than one IoC container.

[](http://d3t0dn7puh4fxw.cloudfront.net/wp-content/uploads/2013/03/singleton.png)

The **Prototype** scopes a single bean definition to have any number of object instances. If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made.

[](http://d3t0dn7puh4fxw.cloudfront.net/wp-content/uploads/2013/03/prototype.png)

## Singleton and Prototype Example:

**HelloWorld.java**

package com;

import java.util.Date;

public class HelloWorld {

private String message;

private Date date;

public Date getDate() {

return date;

}

public void setDate(Date date) {

this.date = date;

}

public void setMessage(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

}

MainApp.java

package com;

import java.util.Date;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("singletonscope");

System.out.println("\*\*\*\*\*\*\*\*\*SINGLETON SCOPE\*\*\*\*\*\*\*\*\*\*\*\*");

objA.setMessage("Message by object A");

objA.setDate(new Date());

System.out.println("Your Message : " + objA.getMessage());

System.out.println("Date : " + objA.getDate().toString());

HelloWorld objB = (HelloWorld) context.getBean("singletonscope");

System.out.println("Your Message : " + objB.getMessage());

System.out.println("Date : " + objB.getDate().toString());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

HelloWorld objC = (HelloWorld) context.getBean("prototypescope");

System.out.println("\*\*\*\*\*\*\*\*\*PROTOTYPE SCOPE\*\*\*\*\*\*\*\*\*\*\*\*");

objC.setMessage("Message by object C");

objC.setDate(new Date());

System.out.println("Your Message : " + objC.getMessage());

System.out.println("Date : " + objC.getDate().toString());

HelloWorld objD = (HelloWorld) context.getBean("prototypescope");

System.out.println("Your Message : " + objD.getMessage());

System.out.println("Your Date : " + objD.getDate());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

Step 2:

Create Beans configuration file Beans.xml under the src folder.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="prototypescope" class="com.javabeat.HelloWorld" scope="prototype">

</bean>

<bean id="singletonscope" class="com.javabeat.HelloWorld" scope="singleton">

</bean>

</beans>

Step 3:

As a final step, let us run the application. If everything is fine with your application, the following output is printed:

\*\*\*\*\*\*\*\*\*SINGLETON SCOPE\*\*\*\*\*\*\*\*\*\*\*\*

Your Message : Message by object A

Date : Fri Mar 29 17:39:21 IST 2013

Your Message : Message by object A

Date : Fri Mar 29 17:39:21 IST 2013

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*PROTOTYPE SCOPE\*\*\*\*\*\*\*\*\*\*\*\*

Your Message : Message by object C

Date : Fri Mar 29 17:39:21 IST 2013

Your Message : null

Your Date : null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Details of the above output:

Here we see that in the case of singleton scope, the second retrieval by objB will display the same message and Date which was set by objA,

even though its retrieved by a new getBean() method. In singleton scope, no matter how many times you retrieve it with getBean(), it will always

return the same instance.

In prototype scope, you will have a new instance for each getBean() method called. Hence for the second retrieval you see that both message

and date are null.