Spring Bean Life Cycle

Student s=new Student();

Class Address{ Student s;} class Student{….}

s=null;

**Let us understand Life Cycle of a bean in Spring**

The beans **life cycle** in spring is one of the most important features to understand.

In many of the real time applications, it is necessary to perform some of the operations before **initializing** a bean and it is necessary to perform some cleanup operations before the bean is**destroyed**by the container.

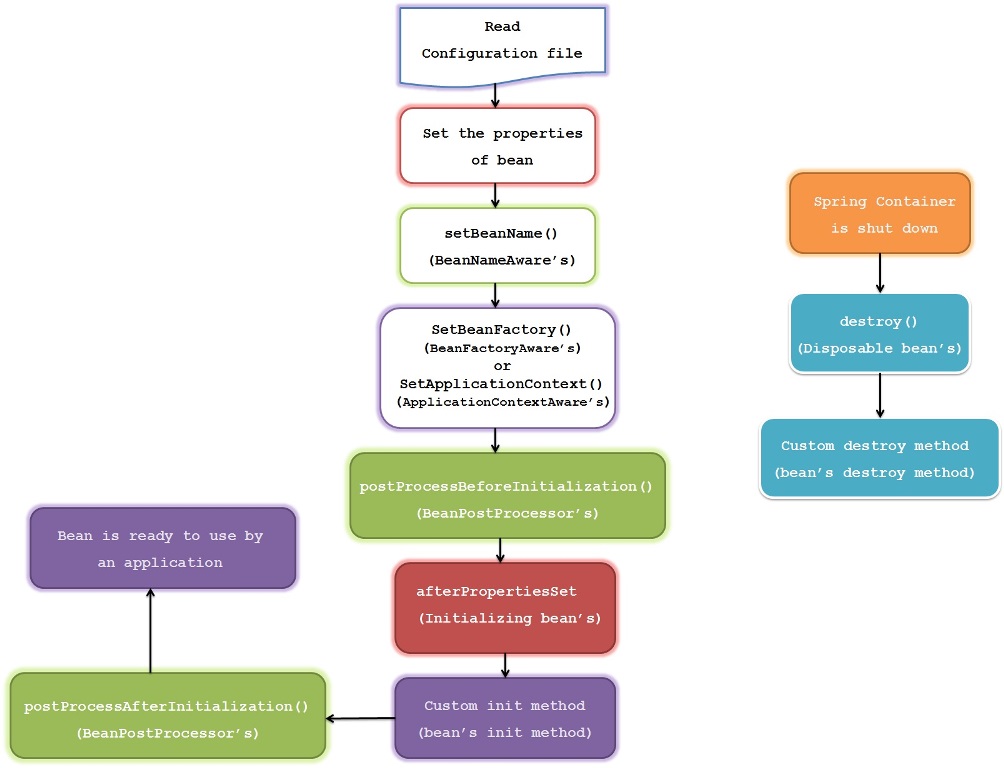
In Java, The life cycle of an object begins with **new** keyword.

When we create an object using **new**, that time it calls the series of hierarchical class **constructors**(call goes from bottom to top and hence execution from top to bottom)  
And finally makes the object available.

And when this object will not have any reference, it will be **garbage collected**. This is the simple life cycle of an object in Java.

But in spring, Bean’s life cycle is having few more things to do.

The spring bean’s life cycle is as shown below



1) Spring container looks for the definition of the **bean** in the spring configuration xml file

2) Spring **instantiate** the bean by calling no argument default constructor of that class,  
If there is **only parameterized constructor** in the class , then bean must be defined in spring xml file with **constructor injection** using which container will instantiate the bean otherwise it will throw bean creation exception.

2) Spring injects the values and references if any into bean’s properties.

3) If the bean implements **BeanNameAware** interface, Spring passes the bean’s ID to the **setBeanName ()** method and executes this method.

4) If the bean implements **BeanFactoryAware** interface, Spring calls the **setBeanFactory ()** method, passing in the bean factory itself and executes this method.

5) If the bean implements **ApplicationContextAware** interface, Spring will call the **setApplicationContext ()** method, passing in a reference to the current application context and executes this method.

6) If the bean implements the **BeanPostProcessor** interface, Spring calls their **postProcessBeforeInitialization ()** method

# Spring bean life cycle methods – BeanPostProcessor

**BeanPostProcessor** is used to perform some operations before and after creating a bean,this allows us to add some code before and after creating the bean.

**BeanPostProcessor** is applicable for all the beans, which means its methods will be executed for each bean we define in the xml.

We can use the **BeanPostProcessor** to execute some logic for all the beans in the application context before and after their initialization

**BeanPostProcessor** interface has 2 methods **postProcessBeforeInitialization()** and **postProcessAfterInitialization()** where former is called after the bean is created and before it is initialized And the latter is called after the bean initialization

import org.springframework.beans.BeansException;

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

public class MyBeanPostProcessor implements BeanPostProcessor {

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

System.out.println("Post Process After Initialization for the bean "+beanName);

return bean;

}

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

System.out.println("Post Process Before Initialization for the bean "+beanName);

return bean;

}

}

# InitializingBean and DisposableBean, Custom init-method and custom destroy-method

**InitializingBean** interface has one method **afterPropertiesSet()** which is called after all the properties of a bean are set.

**DisposableBean** interface has one method **destroy()** which is called before the bean gets destroyed or before it gets removed from the application context.

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class UserService implements InitializingBean,DisposableBean {

private User user;

public UserService() {

System.out.println("UserService no-arg constructor");

}

public void destroy() throws Exception {

System.out.println("UserService destroy method - closing opened resources");

}

public void afterPropertiesSet() throws Exception {

if(!user.getName().equals("Ram")){

user.setName("Ram");

}

System.out.println("UserService afterPropertiesSet method");

}

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

}

**Custom init-method and destroy-method**

Custom init method is called after the bean is initialized and custom destroy method is called before the bean is destroyed.

We can define the custom init and destroy methods in **2** ways.

One is for a **single** bean and other is for **all** the beans

To make it for single bean, we use the code like below

<bean id="personService" class="com.kb.beans.PersonService" init-method="customInit" destroy-method="customDestroy">

<property name="person" ref="person"/>

</bean>

Then we must define these **customInit** and **customDestroy** methods inside this bean class.

To make it for all the beans, we can write code like below

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

Then we must define these **customInit** and **customDestroy** methods inside each bean class  
In this case we must use the same method names for all the beans.

# @PostConstruct and @PreDestroy Annotations

These annotations are introduced in **Spring 2.5** to call the bean life cycle methods just like **init** and **destroy** methods.

**@PostConstruct** : is called after the bean has been initialized and before this bean is returned to the requested object.

**@PreDestroy** : is called just before the bean is removed from the container

**Create a PersonService class as below**

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class PersonService {

private Person person;

public PersonService() {

System.out.println("PersonService no-arg constructor");

}

@PostConstruct

public void Initialize() {

System.out.println("initializing the bean");

}

@PreDestroy

public void cleanUp() {

System.out.println("cleaning up the resources");

}

public Person getPerson() {

return person;

}

public void setPerson(Person person) {

this.person = person;

}

}

# Spring Aware Interfaces for beans

Sometimes it is required that our beans needs to get some information about **Spring container** and its **resources**.

For example, sometime bean need to know the **current Application Context** using which it can perform some operations like loading specific bean from the container in a programmatic way.

So to make the beans aware about this, spring provides lot of **Aware** interfaces.

All we have to do is, make our bean to implement the**Aware** interface and implement the **setter**method of it.

**org.springframework.beans.factory.Aware** is the root marker interface.

All the **Aware**interfaces which we use are the **sub interfaces** of the **Aware**interface.

**Some of the commonly used Aware interfaces are**

**1) ApplicationContextAware**  
Bean implementing this interface can get the **current application context** and this can be used to call any service from the application context

**2) BeanFactoryAware**  
Bean implementing this interface can get the **current bean factory** and this can be used to call any service from the bean factory

**3) BeanNameAware**  
Bean implementing this interface can get its **name**defined in the Spring container.

**4) MessageSourceAware**  
Bean implementing this interface can get the access to**message source** object which is used to achieve **internationalization**

**5) ServletContextAware**  
Bean implementing this interface can get the access to **ServeltContext**which is used to access servlet context parameters and attributes

**6) ServletConfigAware**  
Bean implementing this interface can get the access to **ServletConfig**object which is used to get the servlet config parameters

**7) ApplicationEventPublisherAware**  
Bean implementing this interface can publish the **application events** and we need to create listener which listen this event.

**8) ResourceLoaderAware**  
Bean implementing this interface can load the resources from the classpath or any external file.

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class ApplicationContextAwareImpl implements ApplicationContextAware{

private ApplicationContext applicationContext;

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {

System.out.println("setApplicationContext method called");

this.applicationContext = applicationContext;

}

public void displayPersonDetails(){

Person person = (Person)applicationContext.getBean("person");

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

System.out.println(applicationContext.isSingleton("person"));

}

}

### ****why do we need Dependency Injection****

**DI** is required since it provides **loose coupling.**

###### **Requirement :**

**Client need to open the door, whenever it opens he wants the alarm to be activated,and whenever he closes the door he wants alarm to be deactivated.**

package com.kb.di;

public class Door {

public void open()

{

}

public void close()

{

}

}

Now I need to add **Alarm functionality** to the **Door** class

**Create Alarm.java** – Here I am using sound alarm feature.

package com.kb.di;

public class SoundAlarm {

public SoundAlarm() {

System.out.println("SoundAlarm()");

}

public void activate() {

System.out.println("SoundAlarm activated");

}

public void deactivate() {

System.out.println("SoundAlarm deactivated");

}

}

package com.kb.di;

public class SoundAlarm {

public SoundAlarm() {

System.out.println("SoundAlarm()");

}

public void activate() {

System.out.println("SoundAlarm activated");

}

public void deactivate() {

System.out.println("SoundAlarm deactivated");

}

}

package com.kb.di;

public class Door {

SoundAlarm s=new SoundAlarm();

public void open()

{

S. activate();

}

public void close()

{

s.deactivate();

}

}

package com.kb.di;

public class VisualAlarm {

public VisualAlarm() {

System.out.println("VisualAlarm()");

}

public void activate() {

System.out.println("VisualAlarm activated");

}

public void deactivate() {

System.out.println("VisualAlarm deactivated");

}

}

Create alarm interface

Create soundAlarm

Create visualAlarm class

@Autowired---beanfactory

@Resource--annotation

@Inject—javax.inject

**BeanNameAware , MessageSourceAware and ResourceLoaderAware**

Bean implementing BeanNameAware interface can get its name defined in the Spring container.

Bean implementing MessageSourceAware interface can get the access to message source object which is used to achieve internationalization

Bean implementing ResourceLoaderAware interface can load the resources from the classpath or any external file.

package com.beans;

import java.io.IOException;

import java.util.Locale;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.context.MessageSource;

import org.springframework.context.MessageSourceAware;

import org.springframework.context.ResourceLoaderAware;

import org.springframework.core.io.Resource;

import org.springframework.core.io.ResourceLoader;

public class SpringAwareInterfaceImpl implements BeanNameAware,MessageSourceAware,ResourceLoaderAware {

private MessageSource messageSource;

private ResourceLoader resourceLoader;

public void setResourceLoader(ResourceLoader resourceLoader) {

this.resourceLoader = resourceLoader;

System.out.println("setResourceLoader called");

}

public void setMessageSource(MessageSource messageSource) {

this.messageSource=messageSource;

System.out.println("setMessageSource called");

}

public void setBeanName(String beanName) {

System.out.println("setBeanName called");

System.out.println("Bean name : "+beanName);

}

public void readMessagesSpecificToLocale(){

String englishGreet = messageSource.getMessage("greeting", null, Locale.US);

System.out.println("Greeting in English : " + englishGreet);

String germanGreet = messageSource.getMessage("greeting", null, Locale.GERMAN);

System.out.println("Greeting in German : " + germanGreet);

}

public void getFilePath() throws IOException{

Resource resource = resourceLoader.getResource("classpath:file.txt");

System.out.println("Absolute path of the file :"+resource.getFile().getPath());

//We can further read this file

}

}

In this class, we have implemented **BeanNameAware** interface and accessed the **name of the bean** using the same.

We have implemented **MessageSourceAware** interface using which we have achieved the **internationalization** by reading values from different **locales** properties file.

We have also implemented **ResourceLoaderAware** interface using which we have accessed the **file path** of the file available in the classpath.

**Create 2 files for US and German locale in the resources folder**

**messages\_en\_US.properties**  
greeting=hello

**messages\_de.properties**  
greeting=Hallo

The **greeting**value is accessed based on the **locale**using **MessageResourceAware**

**create a file in the resources folder**

**file.txt**  
hai