Spring Bean Post Processor

As per Spring specifications, The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

There are two methods of BeanPostProcessor

* **postProcessBeforeInitialization**
* **postProcessAfterInitialization**

So in essence the method **postProcessBeforeInitialization** defined in the BeanPostProcessor gets called (as the name indicates) before the initialization of beans and likewise the **postProcessAfterInitialization** gets called after the initialization of the bean.

Note : **So basically the BeanPostProcessor can be used to do custom instantiation logic for several beans whereas the others are defined on a per bean basis.**

**BeanPostProcessor is used to customize the beans’ instantiation process.**

BeanPostProcessor class has two methods.

1) postProcessBeforeInitialization - as name clearly says that it's used to make sure required actions are taken before initialization. e.g. you want to load certain property file/read data from the remote source/service.

2) postProcessAfterInitialization - any thing that you want to do after initialization before bean reference is given to application.

Sequence of the questioned methods in life cycle as follows :

1) BeanPostProcessor.postProcessBeforeInitialization()

2) init()

3) BeanPostProcessor.postProcessAfterInitialization()

4) destroy()

An working example is given below.

**Employee.java**

package com.ddlab.rnd.spring;

public class Employee {

private String name;

private int id;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void init() {

System.out.println("init() method called , Emp object is getting initialized ...");

}

public void destroy() {

System.out.println("destroy() method called , Emp object will be destroyed now ...");

}

}

**Project.java**

package com.ddlab.rnd.spring;  
  
public class Project {  
 private String name;  
 private int id;  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void init() {  
 System.out.println("init() method called , Project object is getting initialized ...");  
 }  
  
 public void destroy() {  
 System.out.println("destroy() method called , Project object will be destroyed now ...");  
 }  
}

**InitBeans.java**

package com.ddlab.rnd.spring;  
  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.beans.BeansException;  
  
public class InitBeans implements BeanPostProcessor {  
   
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("BeforeInitialization : " + beanName);  
 if( bean instanceof Employee ) {  
 Employee emp = (Employee) bean;  
 emp.setId(1);  
 }  
  
 if( bean instanceof Project ) {  
 Project project = (Project) bean;  
 project.setId(1);  
 }  
  
 return bean; // you can return any other object as well  
 }  
  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("AfterInitialization : " + beanName);  
 return bean; // you can return any other object as well  
 }  
  
}

**MainApp.java**

package com.ddlab.rnd.spring;  
  
import org.springframework.context.support.AbstractApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class MainApp {  
 public static void main(String[] args) {  
  
 AbstractApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
  
 Employee obj = (Employee) context.getBean("emp");  
 System.*out*.println(obj.getName());  
 System.*out*.println(obj.getId());  
 context.registerShutdownHook();  
 }  
}

**beans.xml**

<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="emp" class="com.ddlab.rnd.spring.Employee"  
 init-method="init" destroy-method="destroy">  
 <property name="name" value="Hello World!"/>  
 </bean>  
  
 <bean id="project" class="com.ddlab.rnd.spring.Project"  
 init-method="init" destroy-method="destroy">  
 <property name="name" value="Digital marketing"/>  
 </bean>  
  
 <bean class="com.ddlab.rnd.spring.InitBeans" />  
  
</beans>

**Output is given below**

BeforeInitialization : emp

init() method called , Emp object is getting initialized ...

AfterInitialization : emp

BeforeInitialization : project

init() method called , Project object is getting initialized ...

AfterInitialization : project

Hello World!

1

destroy() method called , Project object will be destroyed now ...

destroy() method called , Emp object will be destroyed now ...

Spring Bean Life Cycle

1. The Bean Container finds the definition of the Spring Bean in the Configuration file.
2. The Bean Container creates an instance of the Bean using Java Reflection API.
3. If any properties are mentioned, then they are also applied. If the property itself is a Bean, then it is resolved and set.
4. If the Bean class implements the BeanNameAware interface, then the setBeanName()method will be called by passing the name of the Bean.
5. If the Bean class implements the BeanClassLoaderAware interface, then the method setBeanClassLoader() method will be called by passing an instance of the ClassLoader object that loaded this bean.
6. If the Bean class implements the BeanFactoryAware interface, then the method setBeanFactory() will be called by passing an instance of BeanFactory object.
7. If there are any BeanPostProcessors object associated with the BeanFactory that loaded the Bean, then the method postProcessBeforeInitialization() will be called even before the properties for the Bean are set.
8. If the Bean class implements the InitializingBean interface, then the method afterPropertiesSet() will be called once all the Bean properties defined in the Configuration file are set.
9. If the Bean definition in the Configuration file contains a 'init-method' attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.
10. The postProcessAfterInitialization() method will be called if there are any Bean Post Processors attached for the Bean Factory object.
11. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
12. If the Bean definition in the Configuration file contains a 'destroy-method'attribute, then the corresponding method definition in the Bean class will be called.

# Difference between BeanNameAware and BeanFactoryAware

Consider the two interfaces BeanNameAware and BeanFactoryAware. **The first one makes the object aware of their bean name in a bean factory. The second interface gives the bean access to the Bean Factory that created it.**

BeanNameAware makes the object aware of its bean name. It is best used in pre annotation config spring (2.x). You could reference the bean from a locator by its name then.   
BeanFactoryAware gives the bean access to the beanfactory that created it. For the usefulness of this, you should check the documentation: