**INTERNITY FOUNDATION**

**TASK-15**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Spring - Bean Post Processors**

**Ans-** The **BeanPostProcessor** interface in the Spring framework defines callback methods that we can implement to perform custom logic on the bean instances. BeanPostProcessors operate on bean instances; the Spring IoC container instantiates a bean instance and then BeanPostProcessors do their work on those created instances. These instances are passed as argument to the callback methods. BeanPostProcessor callback methods are called on each bean created by the Springcontainer.

A bean post-processor typically checks for callback interfaces or may wrap a bean with a proxy.

An **ApplicationContext** automatically detects any beans that are defined with the implementation of the **BeanPostProcessor** interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation.

**BeanPostProcessor callback methods**

The org.springframework.beans.factory.config.BeanPostProcessor interface has the following two callback methods.

•***Object postProcessBeforeInitialization(Object bean, String beanName***)- This callback method is called before container initialization methods (such as InitializingBean’s afterPropertiesSet() and any declared init method) are called.

* ***Object postProcessAfterInitialization(Object bean, String beanName) -*** This method is called after any bean initialization callbacks.

**Example-**

package com.Amisha;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

public void init(){

System.out.println("Bean is going through init.");

}

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

**This is a very basic example of implementing BeanPostProcessor, which prints a bean name before and after initialization of any bean.**

**Content of InitHelloWorld.java file −**

package com.Amisha;

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

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("BeforeInitialization : " + beanName);

return bean;

}

public Object postProcessAfterInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("AfterInitialization : " + beanName);

return bean;

}

}

**OUTPUT-**

**BeforeInitialization : helloWorld**

**Bean is going through init.**

**AfterInitialization : helloWorld**

**Your Message : Hello World!**

**Bean will destroy now.**

**Spring - Bean Definition Inheritance**

**Ans-** The child beans definition inherits configuration data from the definition of the parent. Along with that the child definition has the ability to override some of the values while adding others when needed. The [**Java class**](https://data-flair.training/blogs/class-and-object-in-java/)inheritance has nothing to do with the[**Spring Bean definition**](https://data-flair.training/blogs/spring-bean/) but the concept of inheritance is applied.

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method,destroy method, and/or static factory method settings that you specify will override the corresponding parent settings.

***Example***

**We have a Payment class with two fields in it paymentType and amount. If we want to have a configuration where by default paymentType is "cash" then it can be done using bean definition inheritance.**

***Payment class***

**public class Payment {**

**private String paymentType;**

**private double amount;**

**public String getPaymentType() {**

**return paymentType;**

**}**

**public void setPaymentType(String paymentType) {**

**this.paymentType = paymentType;**

**}**

**public double getAmount() {**

**return amount;**

**}**

**public void setAmount(double amount) {**

**this.amount = amount;**

**}**

**@Override**

**public String toString() {**

**StringBuffer sb = new StringBuffer();**

**sb.append("Payment Type ").append(getPaymentType());**

**sb.append(" Amount ").append(getAmount());**

**return sb.toString();**

**}**

**}**

***XML Configuration***

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

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

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

**xmlns:aop="http://www.springframework.org/schema/aop"**

**xmlns:context="http://www.springframework.org/schema/context"**

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

**http://www.springframework.org/schema/beans/spring-beans-4.0.xsd**

**http://www.springframework.org/schema/context**

**http://www.springframework.org/schema/context/spring-context.xsd">**

**<!--<context:component-scan base-package="org.netjs.prog" /> -->**

**<bean id="basePaymentBean" class="org.netjs.prog.Payment">**

**<property name="paymentType" value="cash"/>**

**</bean>**

**<bean id="paymentBean" parent="basePaymentBean">**

**<property name="amount" value="50.15" />**

**<!-- paymentType value will be inherited from parent -->**

**</bean>**

**</beans>**

***Main Program:***

**import org.netjs.prog.Payment;**

**import org.springframework.context.support.ClassPathXmlApplicationContext;**

**public class App {**

**public static void main(String[] args) {**

**ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext**

**("appcontext.xml");**

**Payment bean = (Payment) context.getBean("paymentBean");**

**System.out.println("Values " + bean);**

**context.close();**

**}**

**}**

**Output**

**Values Payment Type cash Amount 50.15**

**==========================x-----END-----x=============================**