**SPRING FRAMEWORK**

**What is Spring:**

Spring is an open source framework to develop enterprise application in a declarative fashion, based on MVC design pattern.It creates the dependent objects and inject those into the bean.

**Inventor of Spring:**

Spring Framework is one of the most popular Java EE frameworks.

It is an open source and light weight framework created by **Rod Johnson** in June **2003**.

**Features of Spring:**

1. **Light weight:** Spring framework is light weight framework because of its POJO model implementation.

2. **Loose Coupling:** Because of dependency injection concept, spring objects are loosely coupled.

3. **Modular fashion:** Spring framework is designed in modular fashion. A programmer can use only needed modules and ignore the rest.

4. **Transaction management:** Spring framework provides transaction management interface for transaction management.

5. **Aspect Oriented Programming(AOP):**

It is an important part of Spring Framework. Aspect Oriented Programming is used for separating cross-cutting concerns

(for example logging, security etc.) from the business logic of the application.

6. **Integration with other frameworks:**

It just tries to integrate them with its framework which provides a solution to greater problems. Example IBATIS, Hibernate, Toplink etc

**How Spring Has Come:**

Earlier **Java Beans** was used in development of java applications and was intended to be used as reusable components.Complex enterprise applications that requires more functionality like security and transaction management java beans were not sufficient.

So to address the complexity of enterprise application,**EJB** was evolved.EJB provided all the intended functionality but failed to achieve the simplicity of application.EJB applications were very complex so harder to maintain.

So a framework that provides the EJB functionality and simplicity of application development was needed.**Spring** provides programming techniques like AOP and DI;these techniques used with POJOs result in implementing enterprise functionality without EJB complexity.

**Callback method:**

A callback method in java is a method which is called when an event occurs. Normally we can implement that by passing an implementation of a certain interface to the system which is responsible for triggering the event.

**Commonly used callback methods in spring:**

**Post-initialization callback methods:**

**Using InitializingBean interface:**

The InitializingBean interface provides afterPropertiesSet() method which can be used for any post-initialization task.

## Syntax:

|  |
| --- |
| public class TestBean implements InitializingBean {  public void afterPropertiesSet() {  // post-initialization task.  }  } |

**Using init-method attribute:**

In XML configuration metadata we can specify the name of the method which has a void no-argument signature in init-method attribute for any post-initialization task.

## Syntax:

|  |
| --- |
| <bean id="testBean" class="Test" init-method="init"/>  **In class definition:**  public class Test {  public void init() {  // post-initialization task.  }  } |

**Pre-destroy callback methods:**

**Using DisposableBean interface:**

The DisposableBean interface provides destroy() method which can be used for any pre-destroy task.

## Syntax:

|  |
| --- |
| public class TestBean implements DisposableBean {  public void destroy() {  // pre-destroy task.  }  } |

***Using destroy-method attribute:***

In XML configuration metadata we can specify the name of the method which has a void no-argument signature in destroy-method attribute for any pre-destroy task.

## Syntax:

|  |
| --- |
| <bean id="testBean" class="Test" destroy-method=" destroy"/>  **In class definition**:  public class Test {  public void destroy() {  // pre-destroy task.  }  } |

**Spring IoC Container**:

The Spring **IoC container** is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

**IOC(Inversion Of Controller):**   Giving control to the container to get instance of object is called **Inversion of Control** means instead of you are creating object using new operator, let the container do that for you.

**By DI, the responsibility of creating objects is shifted from our application code to Spring container, hence the phenomenon is called IOC**.

**Injection:**

Injection is a process of passing the dependency to a dependent object.

**DI(Dependency Injection):**  Way of injecting properties to an object is called **Dependency injection**.

Dependency Injection (DI) is a design pattern that implements inversion of control principle for resolving dependencies

**Types of dependency Injection:**

1. **Constructor-based Dependency Injection.**

|  |
| --- |
| <bean id="student" class="com.Student">  <constructor-arg index="0" value="27"/>  <constructor-arg index="1" value="jai"/>  </bean> |

**2. Setter-based Dependency Injection.**

Spring provides following two distinct types of containers:

1.**BeanFactory container**

**2.ApplicationContext container**

**Spring BeanFactory container:**

**BeanFactory** is a Lightweight container which loads bean definitions and manages your beans .The BeanFactory enables you to read bean definitions and access them using the bean factory.

When using just the BeanFactory you would create one and read in some bean definitions in the XML format as follows:

**How to Create XmlBeanFactory:**

ClassPathResource resource = new ClassPathResource("beans.xml");

BeanFactory factory = new XmlBeanFactory(resource);

//Get bean

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

In the Spring framework, **ApplicationContext** is the advanced container. It extends the BeanFactory interface and provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.The Spring container is responsible for instantiating,

configuring and assembling objects known as beans, as well as managing their lifecycle.

The Spring framework provides several implementations of the **ApplicationContext** interface — **ClassPathXmlApplicationContext** and

**FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

Among the many implementations of **ApplicationContext** are three that are

commonly used:

-> **ClassPathXmlApplicationContext**—Loads a context definition from an

XML file located in the class path, treating context definition files as class

path resources.

-> **FileSystemXmlApplicationContext**—Loads a context definition from an

XML file in the filesystem.

-> **XmlWebApplicationContext**—Loads context definitions from an XML file

contained within a web application.

**ApplicationContext context =**

**new FileSystemXmlApplicationContext("c:/foo.xml");**

Similarly, you can load an application context from within the application’s class

path using ClassPathXmlApplicationContext:

**ApplicationContext context =**

**new ClassPathXmlApplicationContext("foo.xml");**

The difference between these uses of **FileSystemXmlApplicationContext** and

**ClassPathXmlApplicationContext** is that **FileSystemXmlApplicationContext** will

look for foo.xml in a specific location, whereas **ClassPathXmlApplicationContext**

will look for foo.xml anywhere in the class path.

**ContextLoaderListener:**

The **ContextLoaderListener** is a servlet listener that loads additional configuration files into spring application context along with application context

created by DispatcherServlet.

<web-app>

<servlet>

<servlet-name>disp</servlet-name>

<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>disp</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value> /WEB-INF/applicationContext.xml, /WEB-INF/spring-security.xml </param-value>

</context-param>

</web-app>

We should tell to **ContextLoaderListener** which spring configuration file(s) it should load, otherwise the context loader will look for spring

configuration file at /WEB-INF/applicationContext.xml.

**Spring Bean Life Cycle:**

The most important feature of Spring is the bean based approach. The Spring bean is created, managed and dispensed by the Spring IoC container.

Each Spring bean has a lifecycle and understanding the spring bean lifecycle enables better coding.

The life cycle of a Spring bean is very easy to understand. When a bean is instantiated, it may be required to perform some initialization to get

it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required. There are also

other activities between initialization and destruction of the bean. These activities take place behind the scenes.The following are the stages

in a bean’s lifecycle.

1.**Instantiate -** The Spring container instantiates the bean.

2. **Populate properties-** Spring IoC container injects the bean’s properties.

3. **Set Bean Name-** Spring container sets the bean name. If the bean implements **BeanNameAware**, spring container passes the bean’s id to **setBeanName()** method.

4.**Set Bean Factory-**If the bean implements **BeanFactoryAware**, Spring container passes theBeanFactory to **setBeanFactory().**

5.**Pre Initialization-**This stage is also called the bean postprocess . If there are anyBeanPostProcessors, theSpring container calls the

**postProcesserBeforeInitialization ()** method.

6.**Initialize beans-** If the bean implements IntializingBean,its **afterPropertySet()**method is called. If the bean has init method declaration, the specified

initialization method is called.

7.**Post Initialization**- If BeanPostProcessors is implemented by the bean, the Spring container calls their **postProcessAfterinitalization()** method.

8. **Ready to Use-** Now the bean is ready to be used by the application.

9.**Destroy-** The bean is destroyed during this stage. If the bean implements DisposableBean, the Spring IoC container will call the destroy() method .

If a custom destroy () method is defined, the container calls the specified method.

**Bean lifecycle in spring framework:**

1. Spring container finds the bean definition from configuration file.  
2. Spring container instantiates the bean using Java Reflection API.  
3. Spring container applies the all specified properties using DI.  
4. If the bean class implements the BeanNameAware interface, then spring container calls the setBeanName() method by passing bean’s id.  
5. If the bean class implements the BeanClassLoaderAware interface, then spring container calls the setBeanClassLoader() method by passing an instance of the ClassLoader object that loaded this bean.  
6. If the bean class implements the BeanFactoryAware interface, then spring container calls setBeanFactory() method by passing an instance of BeanFactory object.  
7. If there are any BeanPostProcessors object associated with the BeanFactory than spring container calls their postProcessBeforeInitialization() method even before setting the properties for the bean.  
8. If the bean class implements the InitializingBean interface, then spring container calls the afterPropertiesSet() method after setting bean properties.  
9. If init-method is specified in configuration file for the bean then spring container calls the corresponding method in the bean class.  
10. If there are any BeanPostProcessors associated with the bean then spring container calls the postProcessAfterInitialization() method.  
11. If the bean class implements the DisposableBean interface, then spring container calls the destroy() method when the application no longer needs the bean reference.  
12. If destroy-method is specified in the Configuration file for the bean, then spring container calls the corresponding method in the bean class.



### **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:**

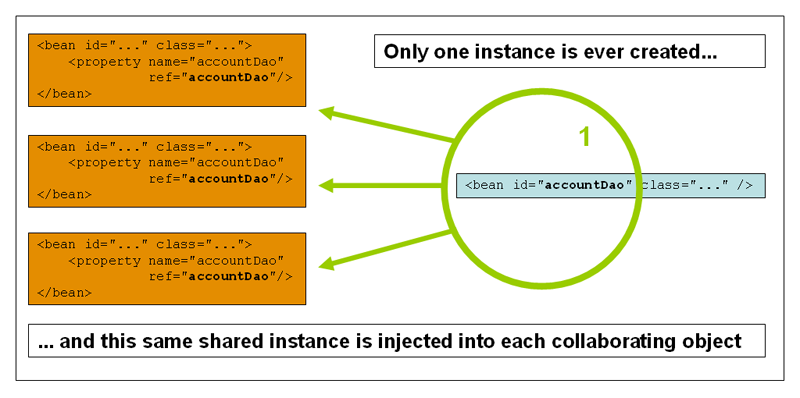
**Spring framework supports following five scopes**. Out of which three scopes are supported only in web ApplicationContext.

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

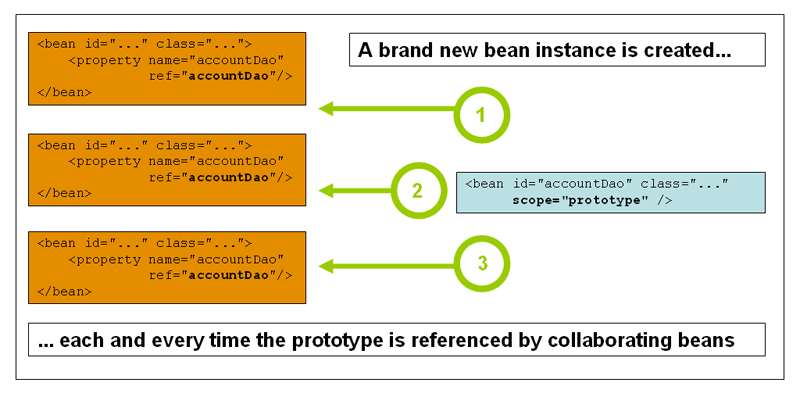
**Note:**Request, Session and Global Session scopes are valid in the context of a web-aware Spring ApplicationContext. This means that you can only use these scoped beans in a an application deployed to a web server. Spring can be used in applications that run in standard JVMs along with applications 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.HelloWorld" scope="prototype">

</bean>

<bean id="singletonscope" class="com.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.

**Spring BeanPostProcessor:**

---------------------------------

With the help of Spring **BeanPostProcessor**, you can easily process the instance created by the Spring IoC container. You can write your own

logic after a bean is created by the container like resource resolution, read some data from external file, validate the data member values etc.

Any bean implementing **BeanPostProcessor** is processed first by the container. You can also control the order in which more than one BeanPostProcessor

should get executed by setting order property. You can only set this property if the BeanPostProcessor implements the Ordered interface.

Important thing to remember

Any bean you write which implements BeanPostProcessor is not eligble for AOP proxies because AOP proxies are applied this way. So be careful when

writing codes, it is possible that AOP proxy may not yet have applied to bean instance. ApplicationContext automatically detects bean which

implements the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors and will be processed later by the container.

BeanPostProcessor proivdes two methods as follow : public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;

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

Sample Example on how to implement BeanPostProcessor

Bean Class package com.bpp;

public class HelloBPP {

private String name;

public void setName(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public void init() {

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

}

public void destroy() {

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

}

}

Bean implementing the BeanPostProcessor package com.bpp;

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

import org.springframework.beans.BeansException;

public class BppProcessor implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean,

String beanName) throws BeansException {

return bean;

}

public Object postProcessAfterInitialization(Object bean,

String beanName) throws BeansException {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

return bean;

}

}

Writing beans.xml (Bean Definition) <?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:lang="http://www.springframework.org/schema/lang"

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

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

http://www.springframework.org/schema/lang

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

<bean id="helloBPP" class="com.bpp.HelloBPP" init-method="init" destroy-method="destroy">

<property name="name" value="Ashish" />

</bean>

<bean class="com.bpp.BppProcessor"/>

</beans>

Finally calling the bean import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.bpp.HelloBPP;

public class Main {

public static void main(final String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

HelloBPP helloBPP = (HelloBPP) ctx.getBean("helloBPP");

System.out.println(helloBPP);

}

}

**Autowiring:**

Autowiring allows Spring to do the instantiation of the class.We don’t need to write explicit injection logic.At run time you will be able to access

all methods of the class without worrying about how you got the

class.

**Advantage of Autowiring:**

It requires the less code because we don't need to write the code to inject the dependency explicitly.

•Reduces the configuration for properties and constructor.

•Automatically gets updated when the configuration change.

**How Spring Autowiring Works?**

All spring beans are managed by spring container called “application context”. The autowiring happens at the time of application starts up. When any autowiring configuration is found either by xml configuration meta data or @Autowired annotation, Spring will look for a class that matches the property in the applicationContext, and inject it automatically.

**Note: There may be the situations when more than one property matched the class spring is looking for, in that case we have to provide the qualify information to help spring for identifying which one in should use.**

**Spring Autowiring Modes:**

* **no:**  No autowiring at all. Bean references must be defined via a reference variable.
* **byName:** Autowiring by property name. This will inspect the application context and look for a bean named exactly the same as the property which needs to be autowired.
* **byType:** Allows a property to be autowired if there is exactly one bean of the property type in the application context. If there is more than one, a fatal exception is thrown.
* **constructor:** This is analogous to byType, but applies to constructor arguments.

Spring framework provides the facility to inject collection values via constructor or setter method. We can use the following inside the constructor or property element.

Collection Injection  
 1. *List.*  
*2. Set.*  
*3. Map.*

*4. Properties*

## Syntax (constructor based dependency injection):

|  |
| --- |
| <bean id="testBeanId" **class**="Test">  <constructor-arg>  <list>  <value>value1</value>  <value>value2</value>  <value>value3</value>  </list>  </constructor-arg>  </bean> |

## Syntax (setter based dependency injection):

|  |
| --- |
| <bean id="testBeanId" **class**="Test">  <property name="testProperty">  <list>  <value>value1</value>  <value>value2</value>  <value>value3</value>  </list>  </property>  </bean> |

## Example Explanation:

We have created two beans “Student” and “Address”. Student class requires an Address class object. In spring configuration file we define Address bean objects and pass these objects as a list in constructor-arg.

## Example:

**Student.java**

|  |
| --- |
| **package** com;    **import** java.util.List;    **public** **class** Student {  **private** String name;  **private** String rollNo;  **private** String className;  **private** List<Address> address;    **public** Student(List<Address> address){  **this**.address = (List<Address>) address;  }    **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getRollNo() {  **return** rollNo;  }  **public** **void** setRollNo(String rollNo) {  **this**.rollNo = rollNo;  }  **public** String getClassName() {  **return** className;  }  **public** **void** setClassName(String className) {  **this**.className = className;  }  **public** List<Address> getAddress() {  **return** address;  }  } |

**Address.java**

|  |
| --- |
| **package** com;  **public** **class** Address {  **private** String addLine;  **private** String city;  **private** String state;  **private** String country;    **public** String getAddLine() {  **return** addLine;  }  **public** **void** setAddLine(String addLine) {  **this**.addLine = addLine;  }  **public** String getCity() {  **return** city;  }  **public** **void** setCity(String city) {  **this**.city = city;  }  **public** String getState() {  **return** state;  }  **public** **void** setState(String state) {  **this**.state = state;  }  **public** String getCountry() {  **return** country;  }  **public** **void** setCountry(String country) {  **this**.country = country;  }  } |

**applicationContext.java**

|  |
| --- |
| <?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="student" **class**="com.Student">  <property name="name" value="Jai"/>  <property name="rollNo" value="MCA/07/06"/>  <property name="className" value="MCA"/>  <constructor-arg>  <list>  <ref bean="address1"/>  <ref bean="address2"/>  </list>  </constructor-arg>  </bean>    <bean id="address1" **class**="com.Address">  <property name="addLine" value="Test address1"/>  <property name="city" value="Gurgaon"/>  <property name="state" value="Haryana"/>  <property name="country" value="India"/>  </bean>    <bean id="address2" **class**="com.Address">  <property name="addLine" value="Test address2"/>  <property name="city" value="Panipat"/>  <property name="state" value="Haryana"/>  <property name="country" value="India"/>  </bean>    </beans> |

**Test.java**

|  |
| --- |
| **package** com;    **import** java.util.List;    **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;    **public** **class** Test {  **public** **static** **void** main(String[] args) {  *//Get ApplicationContext using spring configuration file.*  ApplicationContext context =  **new** ClassPathXmlApplicationContext("applicationContext.xml");    *//Get Student bean object from ApplicationContext instance.*  Student student = (Student) context.getBean("student");    *//Process Student Object.*  System.out.println("Student info: ");  System.out.println("Name: " + student.getName());  System.out.println("RollNo: " + student.getRollNo());  System.out.println("Class: " + student.getClassName());    *//Get Address from Student Object.*  List<Address> studentAddressList = student.getAddress();    *//Declare program counter.*  **int** addressCounter = 1;    *//Iterate Address List.*  **for** (Address studentAddress : studentAddressList) {  *//Process Address Object.*  System.out.println("Student Address " +addressCounter+ ": ");  System.out.println("Address Line: "+studentAddress.getAddLine());  System.out.println("City: " + studentAddress.getCity());  System.out.println("State: " + studentAddress.getState());  System.out.println("Country: " + studentAddress.getCountry());  addressCounter++;  }  }  } |

## Output:

|  |
| --- |
| Student info:  Name: Jai  RollNo: MCA/07/06  **Class**: MCA  Student Address 1:  Address Line: Test address1  City: Gurgaon  State: Haryana  Country: India  Student Address 2:  Address Line: Test address2  City: Panipat  State: Haryana  Country: India |

spring-config.xml

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="stringId" class="java.lang.String">

</bean>

<bean id="student" class="com.Student">

<property name="studentProperties">

<props>

<prop key="Name">Kalai</prop>

<prop key="Course">JAVA</prop>

</props>

</property>

</bean>

</beans>

Student.java

package com;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

public class Student {

private Properties studentProperties = null;

public void setStudentProperties(Properties studentProperties) {

this.studentProperties = studentProperties;

}

public Properties getStudentProperties() {

return studentProperties;

}

}

StudentMain.java:

package com;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.FileSystemResource;

public class StudentMain {

public static void main(String a[]) {

BeanFactory factory = new XmlBeanFactory(new FileSystemResource(

"config\\spring-config.xml"));

Student student = (Student) factory.getBean("student");

Properties studentProperties = student.getStudentProperties();

Iterator iterator3 = studentProperties.values().iterator();

System.out.println("Properties example in Spring");

System.out.println();

while (iterator3.hasNext()) {

Object obj = iterator3.next();

System.out.println(obj);

System.out.println();

}

}

}

OUTPUT:

**DispatcherServlet and ContextLoaderListener:**

DispatcherServlet is basically the front controller in the Spring MVC application as it loads the spring bean configuration file and initializes all the beans that have been configured. If annotations are enabled, it also scans the packages to configure any bean annotated with @Component, @Controller, @Repository or @Service annotations.

ContextLoaderListener, on the other hand, is the listener to start up and shut down the WebApplicationContext in Spring root.

**Explain the role of InternalResourceViewResolver:**

InternalResourceViewResolver is one of the implementations of the ViewResolver interface that allows you to view the page directory and suffix locationsthrough the bean properties.