Springs

Table Of Contents

[1. Dependency Injection 5](#_Toc503133923)

[2. Spring Factory Bean 5](#_Toc503133924)

[3. Spring Program Example – XmlBeanFactory 6](#_Toc503133925)

[a. Example – XmlBeanFactory 6](#_Toc503133926)

[4. Spring Application Context and property initialization 7](#_Toc503133927)

[5. Constructor Injection 9](#_Toc503133928)

[6. Injecting Objects 12](#_Toc503133929)

[7. Injecting Inner Beans, Alias and idref 15](#_Toc503133930)

[7.1 Aliases 15](#_Toc503133931)

[Example: Alias through the “alias” tag. 16](#_Toc503133932)

[Example: Alias through the “name” attribute of the “bean” tag. 17](#_Toc503133933)

[7.2 idref 18](#_Toc503133934)

[8. Initializing Collections 18](#_Toc503133935)

[9. Bean Autowiring 19](#_Toc503133936)

[10. Understanding Bean Scopes 23](#_Toc503133937)

[10.1 Types of Bean Scopes 24](#_Toc503133938)

[10.1.1 Singleton 24](#_Toc503133939)

[10.1.2 Prototype 27](#_Toc503133940)

[10.1.3 Web aware Context bean scope – Request 27](#_Toc503133941)

[10.1.3 Web aware Context bean scope – Session 27](#_Toc503133942)

[10.1.4 Web aware Context bean scope – Global Session 27](#_Toc503133943)

[11. Application Context Aware 27](#_Toc503133944)

[12. Bean Definition Inheritance 30](#_Toc503133945)

[12.1 Bean Definition Inheritance for Collections 31](#_Toc503133946)

[13. Life Cycle Call Backs 34](#_Toc503133947)

[13.1 Through InitializingBean and DisposableBean Interface 35](#_Toc503133948)

[13.2 Through spring.xml file confined to bean 36](#_Toc503133949)

[13.3 Through spring.xml file confined to beans (Global Declaration) 38](#_Toc503133950)

[14. Bean Post Processor 40](#_Toc503133951)

[15. BeanFactoryPostProcessor 42](#_Toc503133952)

[16. Coding To Interface 43](#_Toc503133953)

[17. Required Annotations. 44](#_Toc503133954)

[18. The Auto Wired Annotation 46](#_Toc503133955)

[Example: Autowired injection by type. 46](#_Toc503133956)

[Example: Autowired injection by name 47](#_Toc503133957)

[Example: Autowired injection by Qualifier name 48](#_Toc503133958)

[19. JSR 250 Annotations. 50](#_Toc503133959)

[@Resource Annotation 50](#_Toc503133960)

[@Resource by name (Default) 50](#_Toc503133961)

[@Resource by explicit name 51](#_Toc503133962)

[@PostConstruct Annotation 52](#_Toc503133963)

[@PreDestroy Annotation 53](#_Toc503133964)

[20. Component and Stereo Type Annotation 55](#_Toc503133965)

[21. Using Message Source to get Text from property file 57](#_Toc503133966)

[22. Event Handling 61](#_Toc503133967)

[23. Look Up : 63](#_Toc503133968)

[24. Ref 63](#_Toc503133969)

*SPRINGS*

# Dependency Injection

Java components should be as independent as possible from other classes. This increases the possibility to reuse the classes and also to test them independently of one another (unit testing).

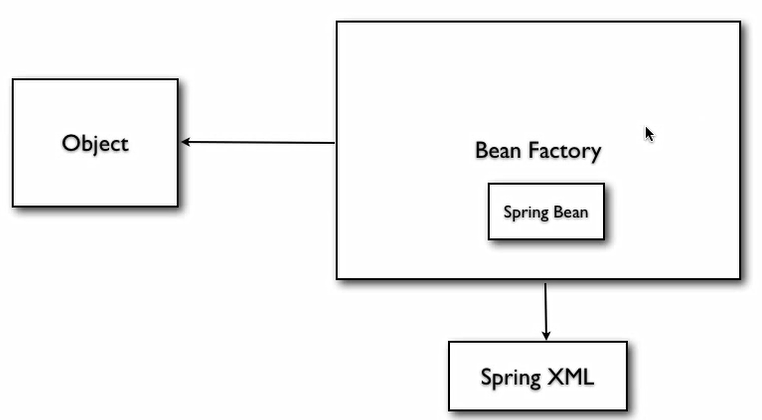
To decouple the java classes from one another the dependency should be injected rather than the class itself creating it. Say if Class A has dependency on class B and class A holds a field to accommodate an object of class B, the dependency can be injected by the following ways.

* Through the constructor of the class A which is called ***Constructor injection***.
* Through a setter in class A, this is called Setter injection.

The General concept between dependency injections is called ***Inversion of Control***. A Class should not be configured by itself rather it should be configured from outside. A design based on independent classes / components increases the re usability of code and independent unit testing. For example if class A expects a DAO (Data Access Object) for receiving the data from DB. We can easily create a test object that mocks the DB connection and inject this object into class A. Spring framework just simplifies the process by providing ways to configure and maintain the objects.

# Spring Factory Bean

The object provides the call to the bean factory and the bean factory in turn refers to the spring XML configuration file and creates the required bean which will then be assigned to the calling object.

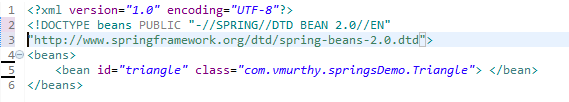


# Spring Program Example – XmlBeanFactory

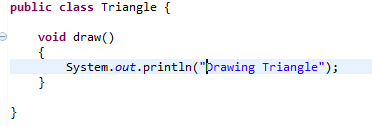
The XmlBeanFactory provides implementation of the BanFactory interface. Using the XmlBeanFactory the configuration information can be provided in the form of an XML file.

# Example – XmlBeanFactory

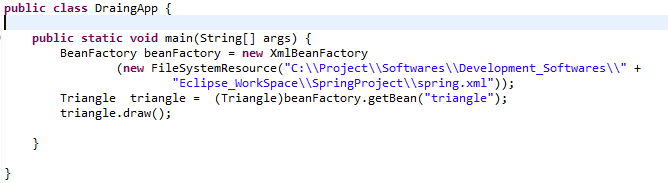
***File Name:*** spring.xml



***File Name:*** Triangle.java

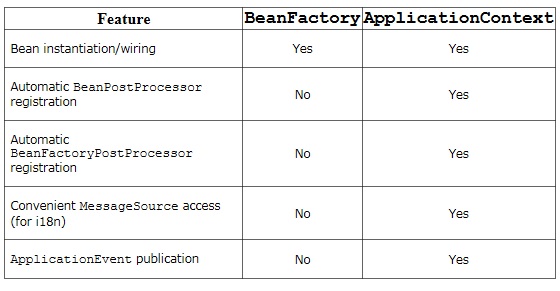


***File Name:*** DrawingApp.java



# Spring Application Context and property initialization

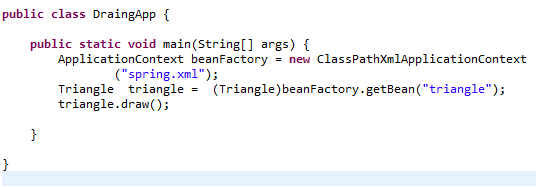
The ApplicationContext is built on top of the BeanFactory which provides additional functionality as mentioned in the below table.



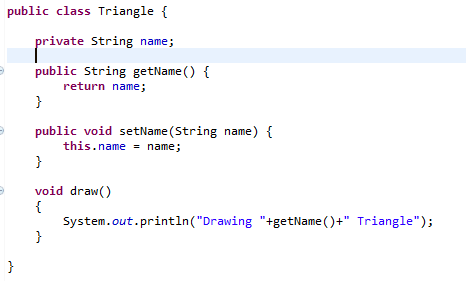
Property initialization: The fields of a class can be initialized to a default value through the property tag.

Example:

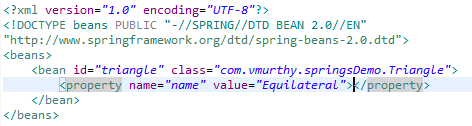
***File Name 1: DrawingApp.java***



***File Name 2: Triangle.java***



***File Name 3***: spring.xml

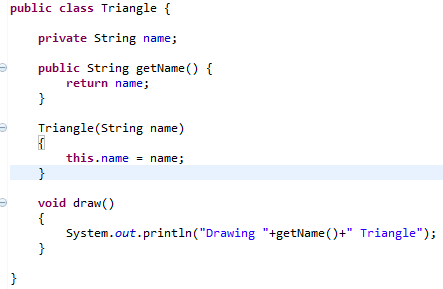


# Constructor Injection

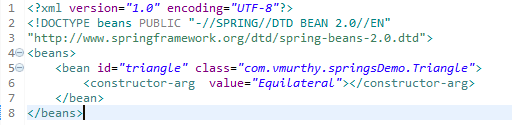
It can be defined as the process of injecting the values to an object through constructor. The XML tag the enables the feature is “constructor- arg”.

The below example illustrates a scenario where we have a single constructor with one string argument to initialize the name field of the Triangle class.

***File Name 1***: Triangle.java



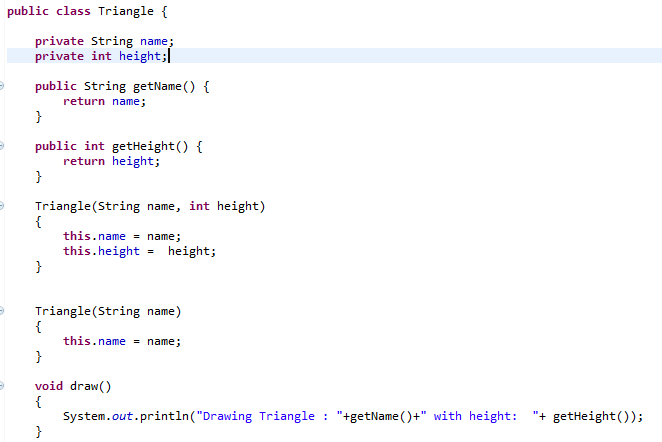
***File Name 2***: spring.xml



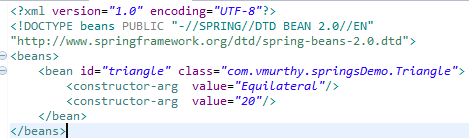
***File Name 3***: DrawingApp.java – remains unchanged.

The ***second*** scenario would be having a constructor with multiple arguments and providing the value for the same through the spring.xml file.

***File Name 1***: Triangle.java



***File Name 2***: spring.xml

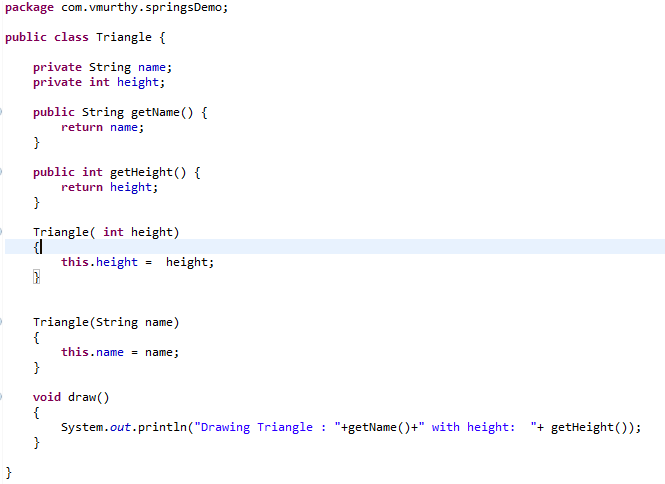


***File Name 3***: DrawingApp.java – remains unchanged.

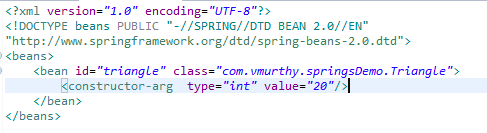
While defining the “*constructor-arg*” it can be observed that we are not defining the data type of the value being initialized as spring takes care of the type conversion implicitly until the constructor can be distinctly identified.

Based on the above mentioned example there might be ambiguity if there exists multiple constructor with one argument. Say one for initializing the height and the other one for the name. In such scenario the attribute “type” of the element “constructor-arg” will be used to provide the appropriate mapping criteria.

***File Name 1***: Triangle.java



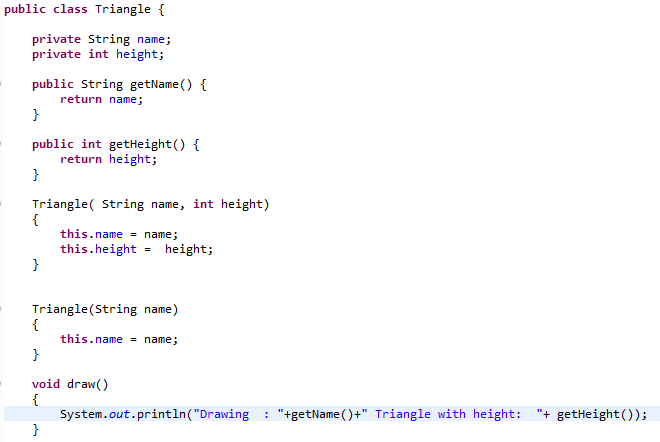
***File Name 2***: spring.xml



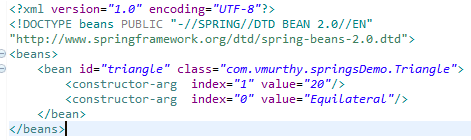
***File Name 3***: DrawingApp.java – remains unchanged.

If we are to define the placeholder value for the values contained in the “constructor-arg” element, it can be defined through the “index” attribute as described in the below mentioned example.

***File Name 1***: Triangle.java



***File Name 2***: spring.xml

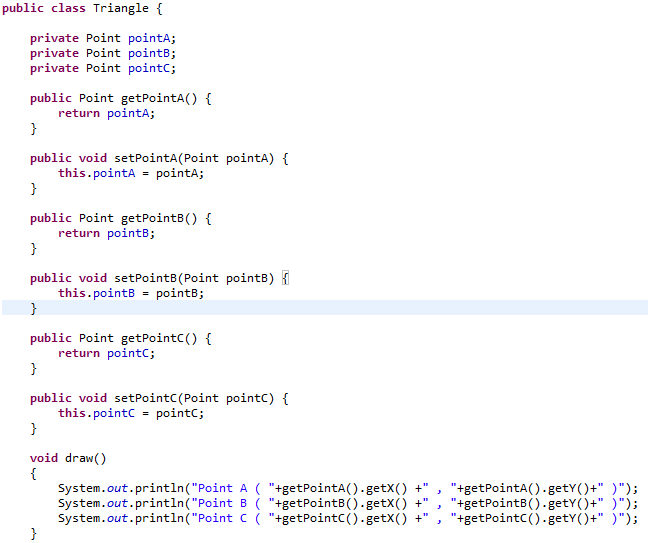


***File Name 3***: DrawingApp.java – remains unchanged.

# Injecting Objects

Through this mechanism we can inject one object into another object. It is achieved through the “*ref*” attribute of the “*property*” element in the spring cml configuration file.

***File Name 1***: Triangle.java

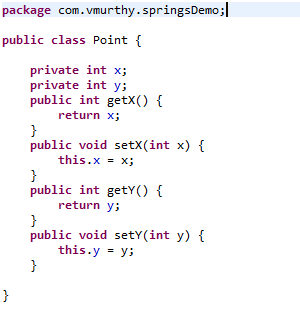


***File Name 2***: spring.xml



***File Name 3***: DrawingApp.java – remains unchanged.

***File Name 4***: Point.java



# Injecting Inner Beans, Alias and idref

Inner bean’s come into play if it is known that a given bean will be referred only by one object. In such case it would be ideal to embed the signature of a bean within another bean.

In the following example the pointB and pointC bean will be defined within its respective property tag in such definition the “id” attributes of the “bean” element and “ref” attribute of the property element is not required.

***File Name 1***: Triangle.java – remains unchanged

***File Name 2***: spring.xml



***File Name 3***: DrawingApp.java – remains unchanged.

***File Name 4***: Point.java– remains unchanged.

## 7.1 Aliases

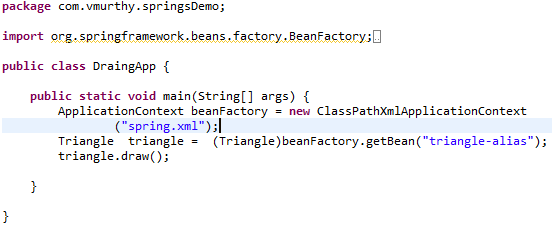
It is used to provide an alternate name for an existing bean. The alternate name can be defined through alias tag or through the “*name*” attribute of the “*bean*” tag.

### Example: Alias through the “alias” tag.

***File Name 1***: spring.xml



***File Name 2***: DrawingApp.java

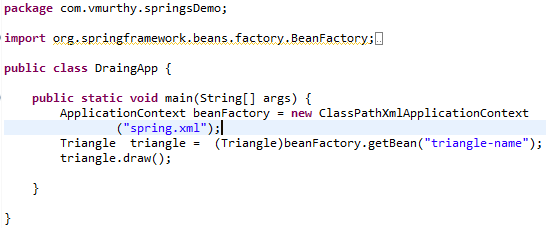


### Example: Alias through the “name” attribute of the “bean” tag.

***File Name 1***: spring.xml



***File Name 2***: DrawingApp.java.



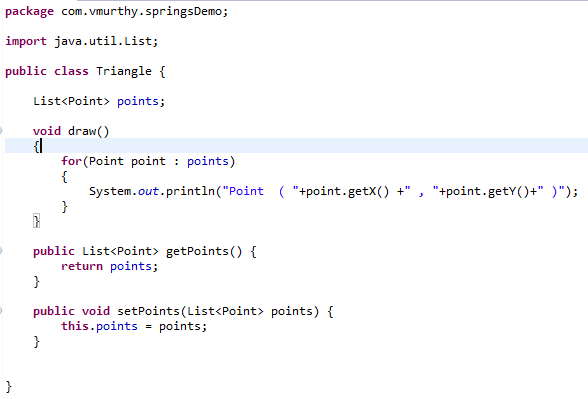
## 7.2 idref

By default the “*ref*” attribute of the property tag can be used to provide the feature of alias or as name attribute. In order to impose restrictions on the attribute to only serve the purpose of distinctly identifying a bean definition the “*ref*” attribute can be renamed as “*idref*”.

# Initializing Collections

Spring provides way to initialize the collections through the xml file. Collections such as List, Set and Map.

***File Name 1***: Triangle.java



***File Name 2***: spring.xml



***File Name 3***: DrawingApp.java – remains unchanged.

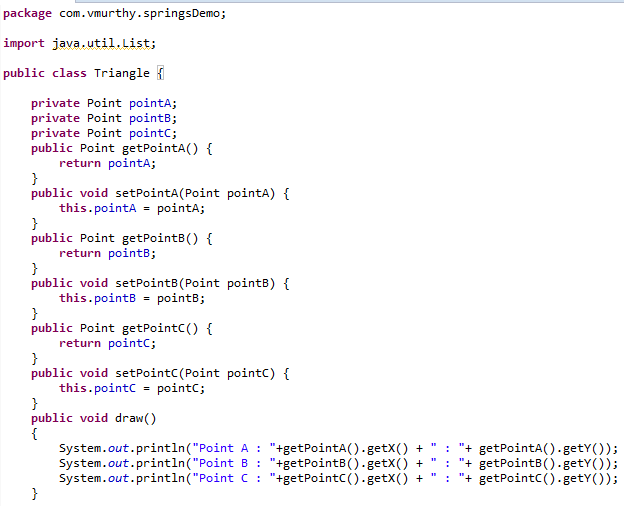
***File Name 4***: Point.java– remains unchanged.

# Bean Autowiring

Through bean autowiring it is possible for us to instruct spring pick up the values for the property for a given object. It can be based on name or by type or by constructor.

For Auto wiring through name, the identifier (i.e. id) of the bean and the name of the field should be same as one another. During the object creation Spring does a look-up for bean with similar name and creates an object of the same.

***File Name 1***: Triangle.java



***File Name 2***: spring.xml



***File Name 3:*** DrawingApp.java – remains unchanged.

***File Name 4:*** Point.java– remains unchanged.

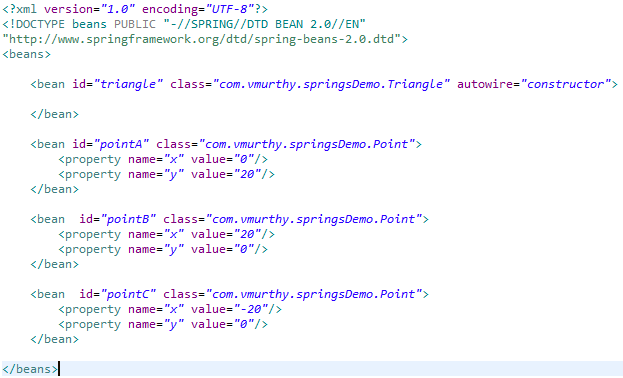
For Auto wiring through type, there should exist only one matching bean with the same type as that of the member variable.

***File Name 1:*** spring.xml



For Auto wiring through constructor there should exist only one matching bean with the same type as that of the arguments for the constructor.

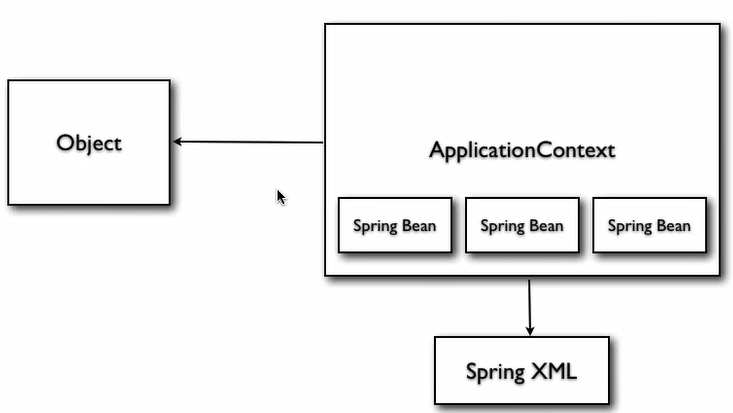
***File Name 1***: spring.xml



# Understanding Bean Scopes

The Application context does not create a bean when the “getBean()” call is initiated. The way it does is at the beginning of the initialization, the application context reads the configuration xml file and creates the required spring beans and later when the “getBean()” call is received the respective bean is returned which is the default behaviour.

And the above mentioned behaviour can be configured to create bean only when an appropriate call is received.



In order to configure the default flow of the Application context initialization there are different type of bean scope.

## 10.1 Types of Bean Scopes

### 10.1.1 Singleton

If a bean definition is declared as “Singleton” for the complete Application context there will exist only one copy of the bean. In other words the bean will be instantiated only once irrespective of the number of “getBean()” calls. This is the default behaviour if the bean scope is not explicitly defined. (one Instance per Application Context)

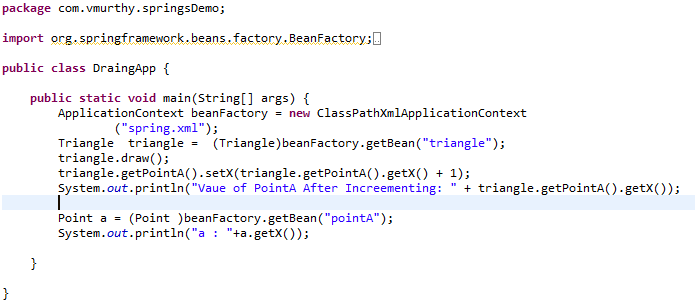
The singleton objects are not thread safe.

Example: If the parent bean is configured with scope as “singleton” and if we do not have any scope explicitly defined for the beans contained within the parent bean the scope is considered as ‘singleton’ by default.

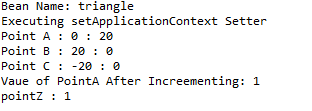
***File Name 1:*** spring.xml



***File Name 2***: DrawingApp.java



***Output:***

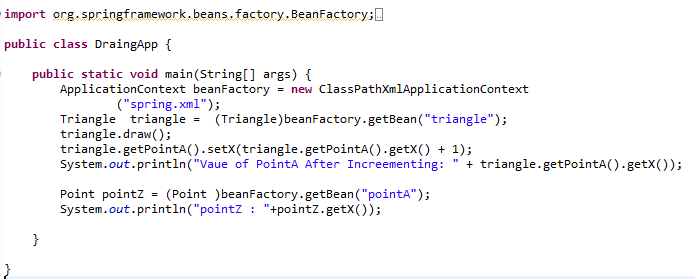


However we can define the triangle bean with scope as ‘singleton’ and “pointA” bean as ‘prototype’. With this kind of configuration for every call to retrieve the “Triangle” bean a new ‘pointA’ bean is created while the instantiation of the “triangle” bean is done only once.

***File Name 1***: spring.xml



***File Name 2***: DrawingApp.java



### 10.1.2 Prototype

This would alter the configuration to instantiate the bean object for every “getBean()” call. For example say if the “pointA” object discussed in the previous example is configured as “prototype” the “pointA” will be instantiated for every ref that is being made to the bean. If there are to property tag referring the “pointA” then the object will be instantiated twice.

### 10.1.3 Web aware Context bean scope – Request

A New bean is created per request.

### 10.1.3 Web aware Context bean scope – Session

A New bean is created per session.

### 10.1.4 Web aware Context bean scope – Global Session

A New bean per Http global session (portlet context)

***File Name 1***: spring.xml

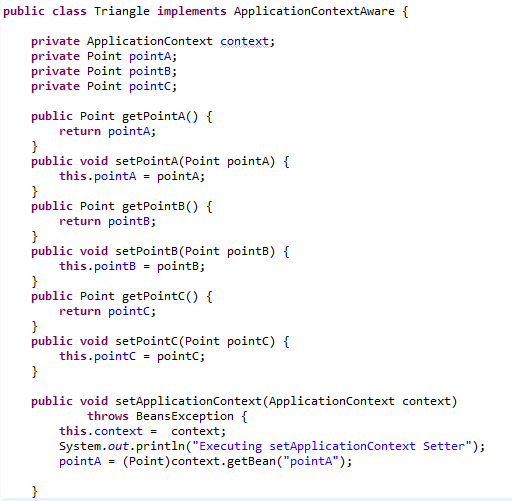


# Application Context Aware

In the example under discussion, the only class that has access to the Application context is the class in which it is instantiated and also the application context is instantiated only once. If there arises a need to allow access to the application context to few other classes it can be achieved through the “*ApplicationContextAware*” interface.

Example: If the Triangle bean is defined with the scope attribute as “Singleton”, by default the member variables of the Triangle will also have the singleton behaviour. If we want the scope of the member variables of “Triangle” bean as “prototype” then we may have to move the implementation to initialize the member variables through the “getBean()” method to the “Triangle” class.

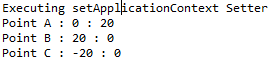
***File Name 1***: Triangle.java



***File Name 2***: spring.xml



***Output:***

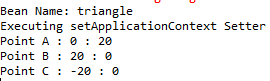


In the output of the above mentioned example it can be observed that the setApplicationContext is being called during the instantiation of the triangle class itself. Similar to ApplicationContextAware we have other classes. One another example is the BeanNameAware

***File Name 1***: Triangle.java



***Output:***



# Bean Definition Inheritance

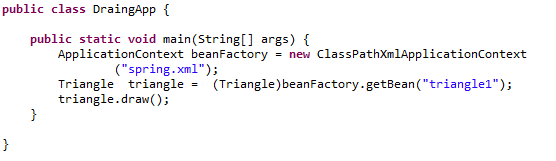
This is a strategy by which the definition of one bean can be acquired by another bean.

Example:

***File Name***: spring.xml



***File Name***: DrawingApp.java



***Output:***



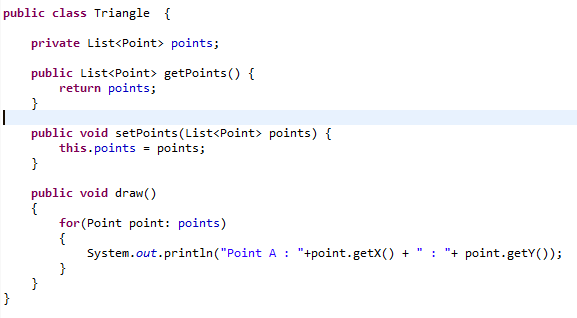
## 12.1 Bean Definition Inheritance for Collections

Inheritance of the collection from one bean to another one is possible through the use of “merge” attribute of the respective collection element. Say in the case of list the syntax is as mentioned below.

**Syntax:** *<list merge=“true” >*

Example:

***File Name***: Triangle.java



***File Name***: spring.xml

***Output:***



In the event if you don’t want a bean to be instantiated we can mark the corresponding bean as abstract which will instruct Spring, not to instantiate the bean.

***File Name***: spring.xml



Note: Most important of all if a bean is defined with attribute “abstract” and value “true”. A “getBean()” method call to abstract bean would result in an exception.

Exception: *Exception in thread "main" org.springframework.beans.factory.BeanIsAbstractException: Error creating bean with name 'parentTriangle': Bean definition is abstract*

# Life Cycle Call Backs

In Order to destroy all the beans the end of the application the application context should be registered to the shutdown hook. In order to do so the “ApplciationContext” interface used so far will be modified to an abstract class “AbstractApplicationContext”.

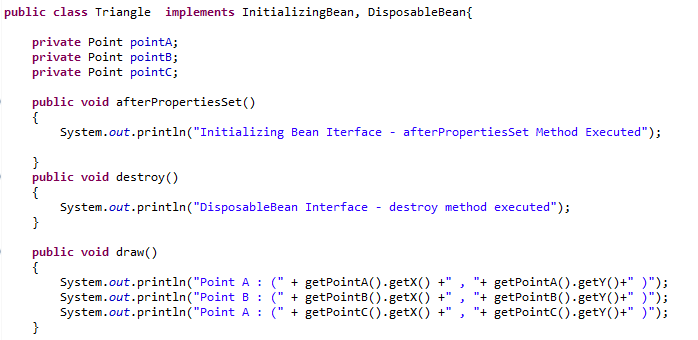
Through the life cycle call backs spring provides feature to call a method during the creation and destruction of a given bean. The invocation can be accomplished though many ways.

## 13.1 Through InitializingBean and DisposableBean Interface

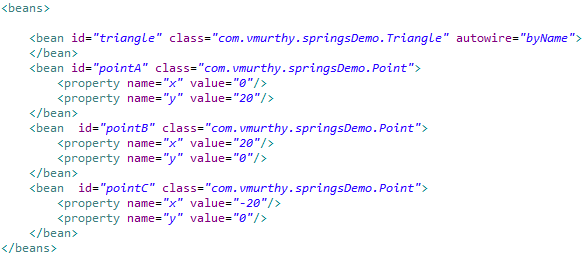
The respective class should implement the “InitializingBean” interface and override the method “afterPropertiesSet()” which will be executed during the creation of the bean. The interface “DisposableBean” should be implemented, overriding “destroy” method which will be executed before the bean is destroyed.

Example:

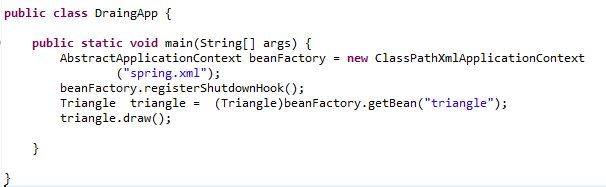
***File Name***: Triangle.java



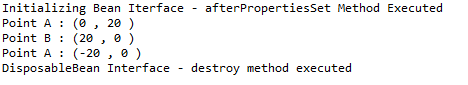
***File Name***: spring.xml



***File Name***: DrawingApp.java



***Output:***



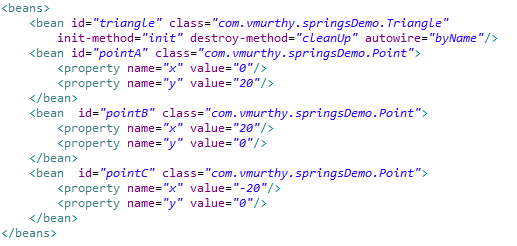
## 13.2 Through spring.xml file confined to bean

Since it is not a good practise to bind beans with spring itself the concept of init and destroy method can be achieved through the spring configuration xml file. The attribute “init-method” and “destroy-method” of the bean tag are used for the same.

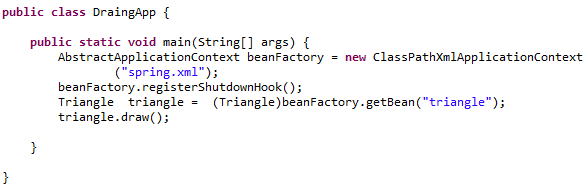
***File Name***: Triangle.java



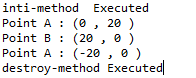
***File Name***: spring.xml



***File Name***: DrawingApp.java



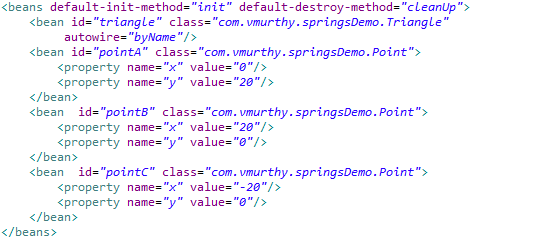
***Output:***



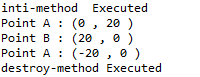
## 13.3 Through spring.xml file confined to beans (Global Declaration)

If the naming convention of the init and destroy method is same throughout the application. We can globally declare the init and destroy method in the “beans” tag. There by spring executes the defined method respectively if the bean class contains the configured method.

***File Name***: spring.xml

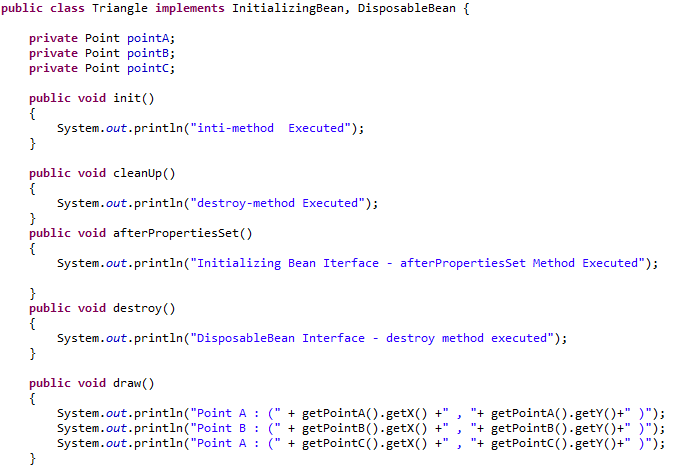


***Output:***

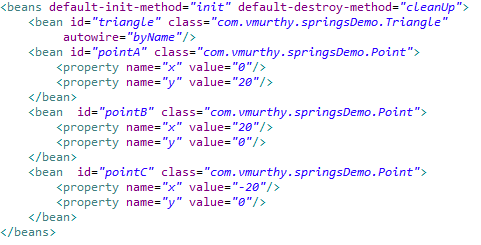


Here is the sequence of execution if the both the InitializingBean, DisposableBean and init method through spring configuration file exists.

***File Name***: Triangle.java



***File Name***: spring.xml



***Output:***

***Initializing Bean Iterface - afterPropertiesSet Method Executed***

inti-method Executed

Point A : (0 , 20 )

Point B : (20 , 0 )

Point A : (-20 , 0 )

Dec 31, 2017 3:56:37 PM ***org.springframework.context.support.ClassPathXmlApplicationContext doClose***

***INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@15eb0a9: startup date [Sun Dec 31 15:56:37 IST 2017]; root of context hierarchy***

DisposableBean Interface - destroy method executed

***destroy-method Executed***

# Bean Post Processor

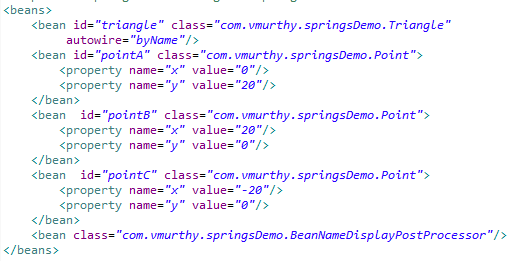
These are classes containing a set of methods, that are to be executed before and after a bean is created. Since the method are defined in a separate class the same set of methods are executed every time a bean is created.

First we create a class implementing the interface BeanPostProcessor which provides the following two methods.

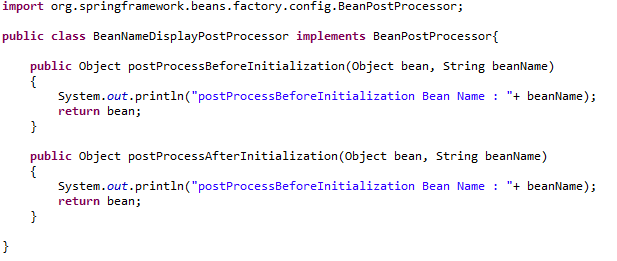
1. postProcessBeforeInitialization - Will be executed before the creation of the bean.
2. postProcessAfterInitialization – Will be executed after the creation of a bean.

Both the methods of the BeanPostProcessor interface takes in, two arguments, the object itself and the bean name. And at the end the object should be explicitly returned for the rest of the execution to take place.

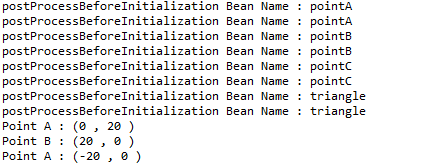
***File Name***: spring.xml



***File Name***: BeanNameDisplayPostProcessor.java



***Output:***



Since the BeanPostProcessor interface does not provide any methods for clean up activities in the event any such activity is required we can make use of the “DisposableBean” interface or “destroy-method” or “default-destroy-method”.

# BeanFactoryPostProcessor

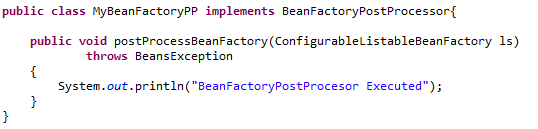
It is used to perform certain action way before the bean factory is created. In order to define the operation the class should implement “BeanFactoryPostProcessor” by overriding the method “postProcessBeanFactory”.

If we are to provide the values to the property of any bean through a property file it can be done through a predefined spring class named “PropertyPlaceHolderConfigurer”.

***File Name***: spring.xml



***File Name***: MyBeanPostProcessor.java



***Output:***

BeanFactoryPostProcesor Executed

Dec 31, 2017 6:18:56 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons

INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@16cd7d5: defining beans [triangle,pointA,pointB,pointC,com.vmurthy.springsDemo.MyBeanFactoryPP#0,org.springframework.beans.factory.config.PreferencesPlaceholderConfigurer#0]; root of factory hierarchy

Point A : (0 , 0 )

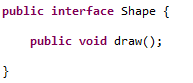
Point B : (20 , 0 )

Point A : (-20 , 0 )

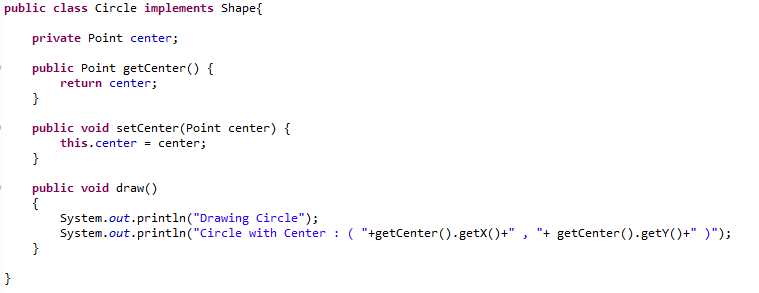
# Coding To Interface

As part of the coding standard it is always advisable to have an interface that serves as a template for all the implementing class.

***File Name***: Shape.java



***File Name***: Circle.java



***File Name***: spring.xml

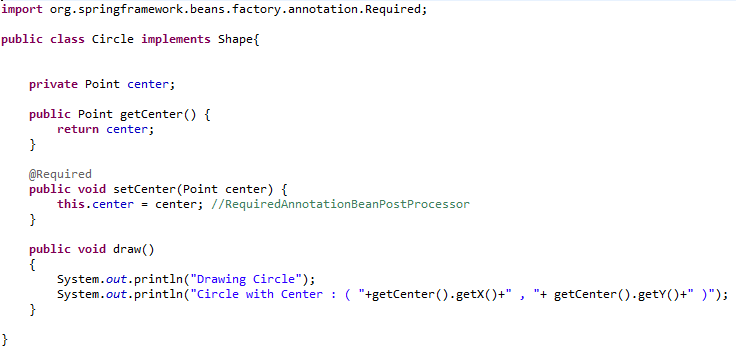


# Required Annotations

In the event if there is a member variable that is not initialized in the spring configuration XML file, during the run time there are possibility of Null Pointer exception. If the volume of the code is large it might become complex to trace the exact code segment which caused the exception.

Under such circumstance we can implement a validation to be done during the application initialization itself. The “@Required” annotation helps in defining that the given member variable should be initialized mandatorily during the bean creation.

***File Name***: Circle.java



***File Name***: spring.xml



In the above mentioned example the inbuilt BeanPostProcessor (“*RequiredAnnotationBeanPostProcessor*”) gets invoked after the bean gets initialized to validate if all the enforced constraints are satisfied. If there is any violation, an exception is generated at the initialization phase.

# The Auto Wired Annotation

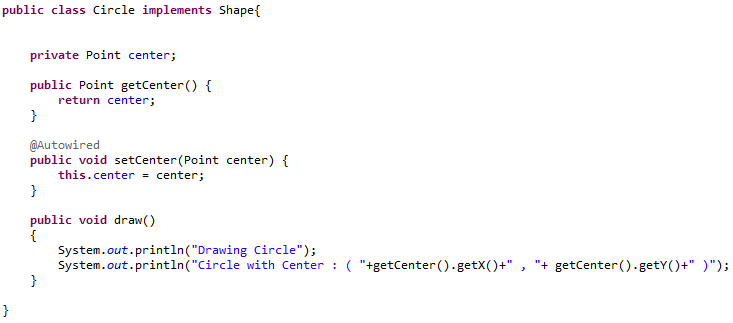
The process of defining the mapping between the member function to an object (i.e. property to a bean) can also be achieved by configuring the information in the java file, the *@Autowired* annotation serves the purpose.

The @Autowired does the process of injecting the right bean into the property by doing a lookup in the spring configuration XML file by type (default behaviour) in the event if there are multiple bean definition with the same type then spring performs a mapping by name.

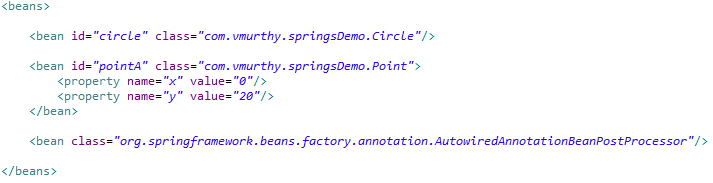
If both the lookup by type and by name fails then spring tries to perform dependency injection by qualifier name.

## Example: Auto wired injection by type.

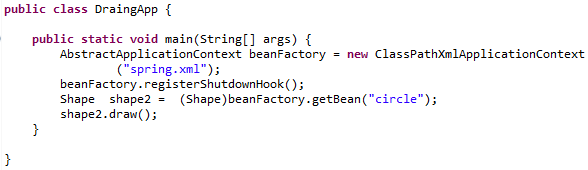
***File Name***: Circle.java



***File Name***: spring.xml

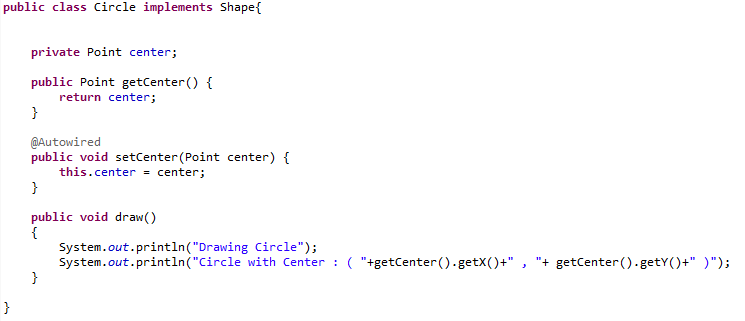


***File Name***: DrawingApp.java

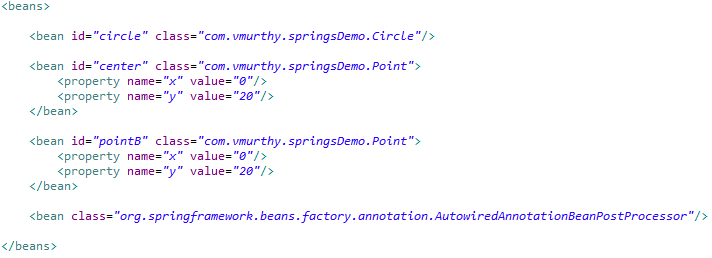


## Example: Auto wired injection by name

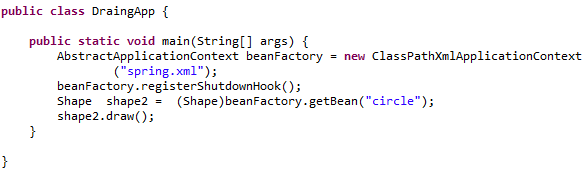
***File Name***: Circle.java



***File Name***: spring.xml



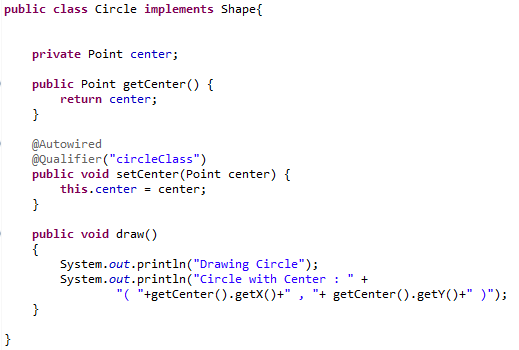
***File Name***: DrawingApp.java



## Example: Autowired injection by Qualifier name

The “*context:annotation-config*” tag implicitly imports the required bean post processor for the annotation used in the application.

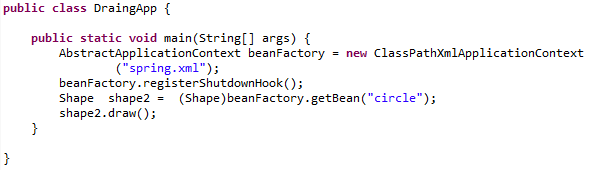
***File Name***: Circle.java



***File Name***: spring.xml

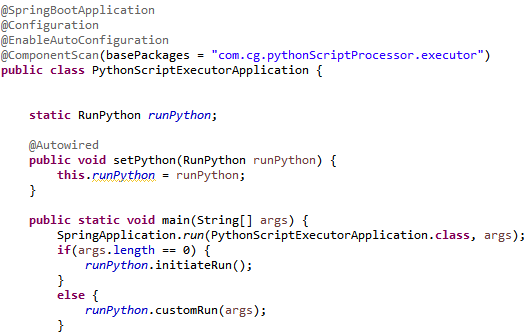


***File Name***: DrawingApp.java



# Static Member Auto wiring

The static members cannot be auto wired through the @AutoWired annotation on top of the member variable. Although it is not a good design for auto wiring a static variable in the event if you are to do so define setter for the corresponding filed and annotate the setter method with “@AutoWired” annotation.



# JSR 250 Annotations.

These are few Java Specification Request annotations which are available in the javax.annotation package

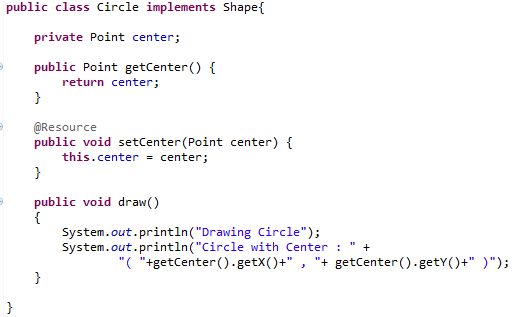
## @Resource Annotation

This annotation is very much similar to the @Autowired annotation provided by spring. By default the annotation does the mapping based on the type if not by name. It can be configured to perform the mapping through specific names using the name attribute.

Example : @Resource(name=“circle”)

### @Resource by name (Default)

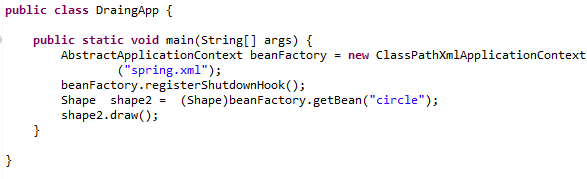
***File Name***: Cricle.java



***File Name***: spring.xml

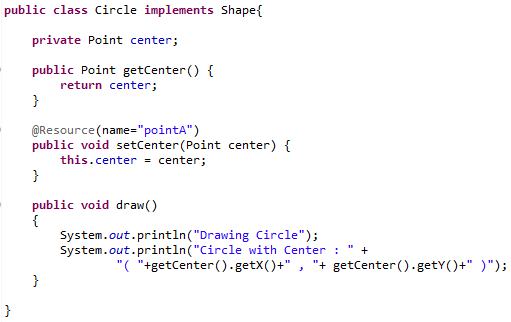


***File Name***: DrawingApp.java



### @Resource by explicit name

***File Name***: Circle.java



***File Name***: spring.xml



## @PostConstruct Annotation

The post construct annotation enables the method to be executed right after the creation of the bean.

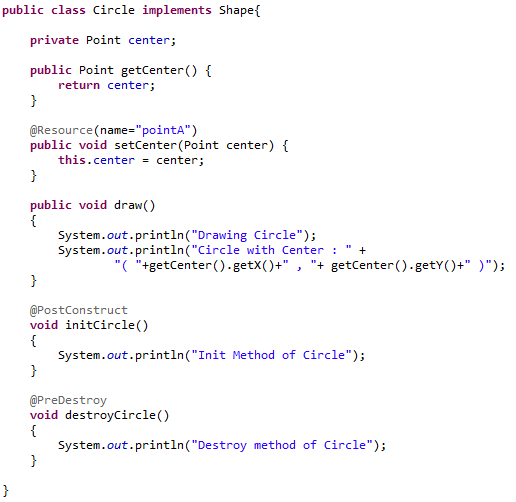
## @PreDestroy Annotation

The PreDestroy annotation enables the method to be executed prior to the destruction of the bean.

For the PreDestroy method to be invoked the application context instance should register shutdown hook.

Example:

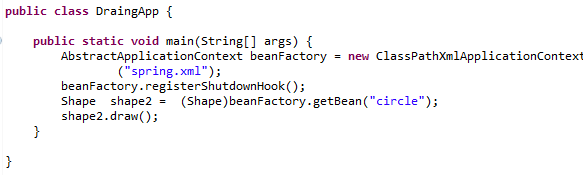
***File Name***: Circle.java



***File Name***: spring.xml



***File Name***: DrawingApp.java



# Component and Stereo Type Annotation

There a series of annotations that is termed as stereo type which provides a specific feature in an application. The @Component is one of the stereo type annotations which mark a class to be termed as a bean.

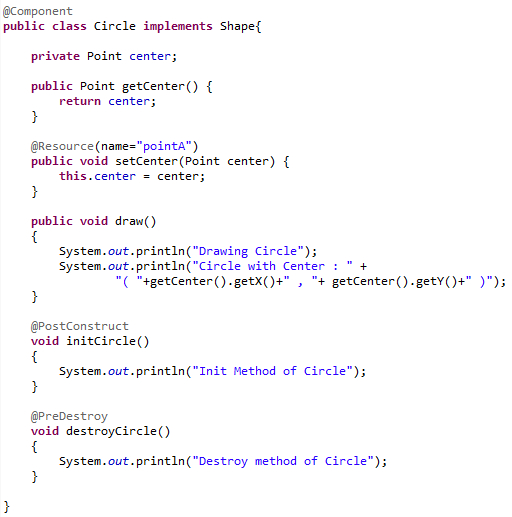
For example instead of defining the “*circle*” bean definition in the “spring.xml” file we can annotate the “*Circle*” class with “@Component” which would also be treated as equivalent to defining the bean in the “spring.xml” file.

The tag <“context:component-scan base-package=””/> is used to instruct spring to look for classes with @Component annotation.

The @Component annotation is not case sensitive. As the identifier of the class is “Circle” whereas the *getBean()* method is called to retrieve “circle” and the dependency injection is successful.

Example:

***File Name***: Circle.java



***File Name***: spring.xml



The Main drawback of the above mentioned approach is that since we are annotating above the class we will not be able to create multiple instance of the class with different values for the field values.

# Using Message Source to get Text from property file

The Application Context provides additional features such as messaging and internationalization over Bean Factory. The term messaging refers to the method through which we can print messages in an application like logger or error messages.

***First step :***

Create a property file have in key value pair where in a key represents a unique identifier associated to a message.

***Second Step:***

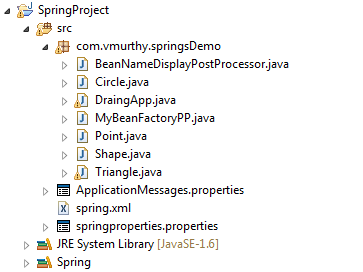
The message source is an interface which comes with a various implementation, in the below mentioned example we will be using the “*ResourceBundleMessageSource*” implementation which has to be defined in the spring configuration XML file which will initiate the process of performing the lookup for the property file and retrieve the appropriate message for a given call.

***Third Step:***

The message can be retrieved through the getMessage() method of the “ApplicationContext”. In the event if the message has to be displayed in any other call it can be achieved either by implementing the “*ApplicationContextAware*” interface or by creating a member variable of type “*MessageSource*” class which will be auto wired to the “*messsageSource*”  
“ bean in the spring configuration file.

*Below are the examples through application context and message source*

Location of the property file:



***File Name***: ApplicationMessages.properties



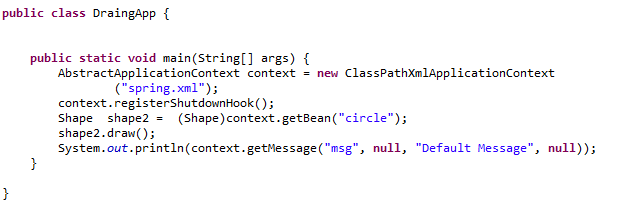
***File Name***: spring.xml



***File Name***: Circle.java



***File Name***: DrawingApp.java



***Output:***

Jan 7, 2018 12:06:20 AM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@15eb0a9: startup date [Sun Jan 07 00:06:20 IST 2018]; root of context hierarchy

Jan 7, 2018 12:06:20 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [spring.xml]

Jan 7, 2018 12:06:20 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons

INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1d80e6d: defining beans [pointA,pointB,messageSource,org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotationProcessor,circle]; root of factory hierarchy

*Init Method of Circle*

*Drawing Circle*

*Circle with Center (0,20)*

*Spring Application Executing!!!*

Jan 7, 2018 12:06:20 AM org.springframework.context.support.ClassPathXmlApplicationContext doClose

INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@15eb0a9: startup date [Sun Jan 07 00:06:20 IST 2018]; root of context hierarchy

*Destroy method of Circle*

# Event Handling

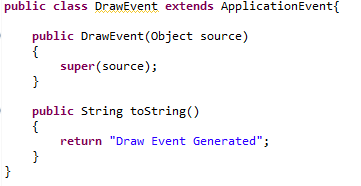
There are basically three entities involved in the event handling processes.

***First Entity***: The Event listener class having the ability to listen to the on-going events within the application and act upon when an event is triggered. In order to do the corresponding class should implement “*ApplicationListener*” interface overriding the method “*onApplicationEvent*”.

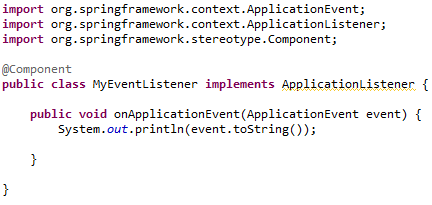
***Second Entity***: The Event publisher which is the snippet of code or the class which triggers the event. Which would be possible only if a given class implements “*ApplicationEventPublisherAware*” interface overriding the method “*setAapplicationEventPublisher*” and within the implementation of the overrided method create and initialize member variable of type “ApplicationEventPublisher”.

***Third Entity***: The event itself, which can be created by defining a class the extends the ApplicationEvent Class.

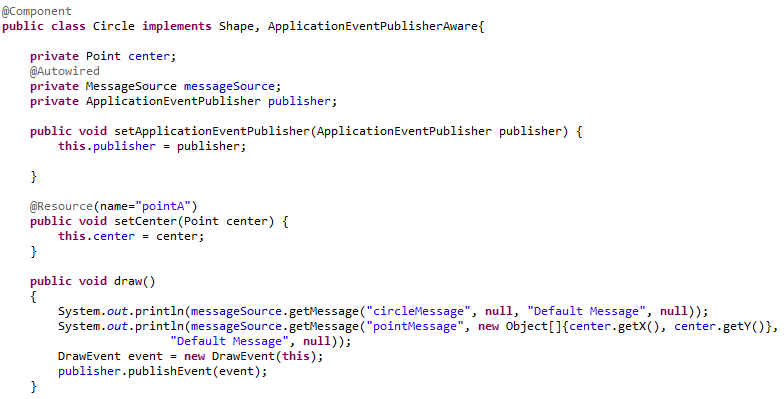
***File Name***: DrawEvent.java



***File Name***: MyEventListener.java



***File Name***: Circle.java



# Look Up :

1. Difference between POJO and Bean class -
2. XmlBeanFactory is just one implementation of the BeanFacotry,list the other commonly used implementation of the BeanFactory and the specific features provided by them.
3. Revisit Section 4 at the end of the topic to know he additional features of Application context.
4. Memorize the class names used in the “DrawingApp.java”.
5. Which implementation of Application Context is being used in EHS project. – FileSystemXmlApplicationContext – Mark the features of the same.
6. Think of a scenario where the Containing object needs to be singleton and the contained object to be prototype.
7. Practise writing spring.xml and DrawingApp.java file.
8. Marker Interface.
9. Design Pattern ; Adapter , Singleton,.
10. Data Structure : Annotations with example.
11. SQL : PL/ SQL , Joins.
12. Permutation Combinations.
13. Example of encapsulation
14. What is the difference between the BeanFactory and Application Context.

# Ref

***File Name***: Triangle.java

***File Name***: spring.xml

***File Name***: DrawingApp.java – remains unchanged.

***File Name***: Point.java

***Output:***