**Springs**

**Dependency injection or dependency inversion**

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled

**Problems of Dependency Lookup**

There are mainly two problems of dependency lookup.

* **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
* **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

For example we have one small application which will draw triangle or circle depending upon on the object you pass.

Application class

Traingle tr = new Traingle (); tr.draw();

Circle cr = new Circle(); cr.draw();

Circle

Draw () method

Traingle

Draw () method

Now in Application class we are creating the objects of Traingle and Circle, if we want remove this we can use polymorphism

Application Class

Shape sh = new Triangle(); sh.draw();

Shape sh = new Circle (); sh.draw();

Interface Shape

Draw () method

Circle

Draw () method

Triangle

Draw () method

Here we are using polymorphism but still we are creating a object at compile time and my Application class knows which type object we are creating. Here Application class is dependent on the Triangle and circle, spring framework gives us to remove this dependency.

Drawing Class

Private Shape shape;

Public void setShape(Shape shape){ this.shape= shape; }

Public void draw(Shape shape){ shape.draw(); }

Interface Shape

Draw () method

Different Class

Shape tr = new Triangle(); Drawing dr = new Drawing(); dr.setShape(tr); dr.draw();

Shape sh = new Circle(); dr.setShape(sh); dr.draw();

Circle

Draw () method

Triangle

Draw () method

Now if we want extend our application to draw point, rectangle etc then we no need to modify the Drawing class and just need to create a new class rectangle and we can call draw method with the existing code. With the minimal change we can extend your application.

### Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

## Setter Injection

This is the most popular and simple DI method, it will injects the dependency via a setter method.

##### **Example**

A helper class with a setter method.

**package** com.mkyong.output;

**import** com.mkyong.output.IOutputGenerator;

**public** **class** OutputHelper

{

IOutputGenerator outputGenerator;

**public** **void** setOutputGenerator(IOutputGenerator outputGenerator){

**this**.outputGenerator = outputGenerator;

}

}

A bean configuration file to declare the beans and set the dependency via setter injection (property tag).

**<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-2.5.xsd"**>**

**<bean** id="OutputHelper" class="com.mkyong.output.OutputHelper"**>**

**<property** name="outputGenerator"**>**

**<ref** bean="CsvOutputGenerator" **/>**

**</property>**

**</bean>**

**<bean** id="CsvOutputGenerator" class="com.mkyong.output.impl.CsvOutputGenerator" **/>**

**<bean** id="JsonOutputGenerator" class="com.mkyong.output.impl.JsonOutputGenerator" **/>**

**</beans>**

You just injects a ‘CsvOutputGenerator’ bean into ‘OutputHelper’ object via a setter method (setOutputGenerator).

## Constructor Injection

This DI method will injects the dependency via a constructor.

##### **Example**

A helper class with a constructor.

**package** com.mkyong.output;

**import** com.mkyong.output.IOutputGenerator;

**public** **class** OutputHelper

{

IOutputGenerator outputGenerator;

OutputHelper(IOutputGenerator outputGenerator){

**this**.outputGenerator = outputGenerator;

}

}

A bean configuration file to declare the beans and set the dependency via constructor injection (constructor-arg tag).

**<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-2.5.xsd"**>**

**<bean** id="OutputHelper" class="com.mkyong.output.OutputHelper"**>**

**<constructor-arg>**

**<bean** class="com.mkyong.output.impl.CsvOutputGenerator" **/>**

**</constructor-arg>**

**</bean>**

**<bean** id="CsvOutputGenerator" class="com.mkyong.output.impl.CsvOutputGenerator" **/>**

**<bean** id="JsonOutputGenerator" class="com.mkyong.output.impl.JsonOutputGenerator" **/>**

**</beans>**

You just injects a ‘CsvOutputGenerator’ bean into ‘OutputHelper’ object via a constructor.

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {

// ...

}

}

The following configuration works fine:

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

Let us check one more case where we pass different types to the constructor. Consider the following class:

package x.y;

public class Foo {

public Foo(int year, String name) {

// ...

}

}

The container can also use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute. For example:

<beans>

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="2001"/>

<constructor-arg type="java.lang.String" value="Zara"/>

</bean>

</beans>

Finally and the best way to pass constructor arguments, use the index attribute to specify explicitly the index of constructor arguments. Here the index is 0 based. For example:

<beans>

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="2001"/>

<constructor-arg index="1" value="Zara"/>

</bean>

</beans>

A final note, in case you are passing a reference to an object, you need to use **ref** attribute of <constructor-arg> tag and if you are passing a value directly then you should use **value**attribute as shown above.

# Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

**Bean Factory**

Spring container contains beans, its factory for beans. Container handles instantiation of objects and life cycle of objects and destractions of objects.

Bean factory reads the configuration file or spring.xml file and it will create the beans as specified in xml and whenever our programs ask for beans, bean factory will provide the beans.

**Autowiring**

If we specify the autowire as byName it will autowire i.e, it will inject the object by autowire like if my class A has the object B and we specify autowire is byName then it will automatically inject the object by setter method.

<bean id="b" **class**="org.sssit.B"></bean>

<bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

1. **public** **class** B {
2. B(){System.out.println("b is created");}
3. **void** print(){System.out.println("hello b");}
4. }
5. **public** **class** A {
6. B b;
7. A(){System.out.println("a is created");}
8. **public** B getB() {
9. **return** b;
10. }
11. **public** **void** setB(B b) {
12. **this**.b = b;
13. }
14. **void** print(){System.out.println("hello a");}
15. **void** display(){
16. print();
17. b.print();
18. }
19. }
20. **public** **class** Test {
21. **public** **static** **void** main(String[] args) {
22. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
23. A a=context.getBean("a",A.**class**);
24. a.display();
25. }
26. }

Output:

b is created

a is created

hello a

hello b

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

## byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byType"></bean>

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="b2" **class**="org.sssit.B"></bean>
3. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

In such case, it will throw exception.

**bean Scope:**

if the scope is singleton then container will create only one object per container

and it will pass the same object every time when we ask for the bean object.

if the scope is prototype then container will create new object everytime

when ever we ask for the bean object from the container.

The default scope is Singleton

## 7.1 Overview of Spring Bean Life Cycle

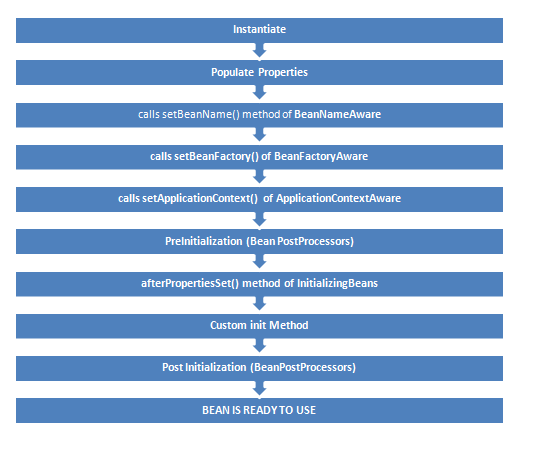
Life of traditional java objects starts on calling new operator which instantiates the object and finalize() method is getting called when the object is eligible for garbage collection. Life cycle of Spring beans are different as compared to traditional java objects.

Spring framework provides the following ways which can be used to control the lifecycle of  bean:

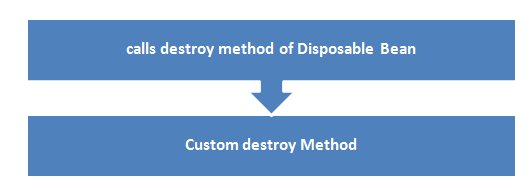
1. InitializingBean and DisposableBean callback interfaces
2. Bean Name, bean factory and Application Context  Aware interfaces for specific behavior
3. custom init() and destroy() methods in bean configuration file

For annotation based configurations -

@PostConstruct and @PreDestroy annotations

Below diagram shows the complete lifecycle methods (from instantiate to Ready To use )

Following diagram shows the method calling  at the time of destruction.



## ****7.2 InitializingBean and DisposbleBean callback interfaces****

* InitalizingBean interface is defined under org.springframework.beans.factory package and declares a single method where we can  be used to add any initialization related code. Any bean implementing InitalizingBean needs to provide an implementation of afterPropertiesSet() method. Signature of method is:

                  void afterPropertiesSet() throws Exception;

* Similarly DisposableBean interface is defined under the org.springframework.beans.factory and declares a single method which gets executed when bean is destroyed and can be used to add any cleanup related code. Any bean implementing DisposableBean needs to provide an implementation of destroy() method. Signature of method is :

                  void destroy() throws Exception;

         This approach is simple to use but it’s not recommended because it will create tight coupling with the Spring framework in our bean implementations.

### ****7.2.1 Example-****

Lets write an example to implement InitalizingBean and DisposableBean interface

**Solution:**

a) Write a PersonBean which implements InitializingBean and DisposableBean interface like below

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | 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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <?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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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

## ****7.3 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

### ****7.3.1 Example****

Lets write an example to implement Aware interfaces

**Solution:**

 a) Create  a class (AwareBean) which implements ApplicationContextAware, BeanNameAware and BeanFactoryAware

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | 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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?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> |

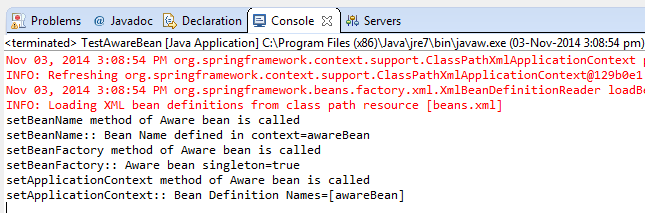
c)  Create TestAwareBean class which will just loads the beans.xml and test the aware life cycle

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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.



## 7.4 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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <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.

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4 | <beans default-init-method=”customDefaultInitMethod” default-destroy-method=”customDefaultDestroyMethod” >      <bean id="bean\_id" class="bean.class" >  </bean>  </beans> |

### ****7.4.1 Example****

Write and example to show the init-method and destroy-method

**Solution**

 a)Write a class CustomLifeCycleMehodBean

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <?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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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

### ****7.4.2 – Example****

Write an example to demonstrate global init and destroy methods

**Solution**

a)Write a class CustomGlobalLifeCycleMehodBean

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <?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

[?](http://www.wideskills.com/spring/spring-bean-lifecycle)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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();       }  } |

d)Run the Program

You will see below output and global custom life cycle methods are getting called

## output to show global custom life cycle methods are being called

## ****7.5 @PostConstruct and @PreDestroy annotations****

Spring 2.5 onwards we can use annotations to specify life cycle methods using @PostConstruct and @PreDestroy annotations.

Details on the annotation will be covered in upcoming chapters.

**Callback methods:**

Initialization callbacks:

The org.springframework.beans.factory.InitializingBean interface specifies a single method:

void afterPropertiesSet() throws Exception;So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

public class ExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example:

<bean id="exampleBean"

class="examples.ExampleBean" init-method="init"/>Following is the class definition:

public class ExampleBean {

public void init() {

// do some initialization work

}

}

**Destruction callbacks**

The org.springframework.beans.factory.DisposableBean interface specifies a single method:

void destroy() throws Exception;So you can simply implement above interface and finalization work can be done inside destroy() method as follows:

public class ExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work

}

}In the case of XML-based configuration metadata, you can use the destroy-method attribute to specify the name of the method that has a void no-argument signature. For example:

<bean id="exampleBean"

class="examples.ExampleBean" destroy-method="destroy"/>Following is the class definition:

public class ExampleBean {

public void destroy() {

// do some destruction work

}

}

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

**Default initialization and destroy methods:**

If you have too many beans having initialization and or destroy methods with the same name, you don't need to declare init-method and destroy-method on each individual bean. Instead framework provides the flexibility to configure such situation using default-init-method and default-destroy-method attributes on the <beans> element as follows:

<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="init"

default-destroy-method="destroy">

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

</beans>

**Bean Definitaion Inheritance**

if you want to Inherit from any class, you can sepecify it in the Springs.xml

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld">

<property name="message1" value="Hello World!"/>

<property name="message2" value="Hello Second World!"/>

</bean>

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia" parent="helloWorld">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Namaste India!"/>

</bean>

here HelloWorld is the parent class and HelloIndia can acess any varibales or methods

including private varibales from the HelloWorld Class.

You can also specify class as Abstarcat, below is the example

<bean id="beanTeamplate" abstract="true" class="fully qualified class name">

<property name="message1" value="Hello World!"/>

<property name="message2" value="Hello Second World!"/>

<property name="message3" value="Namaste India!"/>

</bean>

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia" parent="beanTeamplate">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Namaste India!"/>

</bean>

now you can not instantiate beanTemplate class becuase it is abtract class.

**Dependency Injection:**

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker">

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>

</property>

</bean>

spellChecker is the propert name or varibale in textEditor class and its type is

com.tutorialspoint.SpellChecker

**Injecting the collections:**

<!-- Definition for javaCollection -->

<bean id="javaCollection" class="com.tutorialspoint.JavaCollection">

<!-- results in a setAddressList(java.util.List) call -->

<property name="addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<!-- results in a setAddressSet(java.util.Set) call -->

<property name="addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

<!-- results in a setAddressMap(java.util.Map) call -->

<property name="addressMap">

<map>

<entry key="1" value="INDIA"/>

<entry key="2" value="Pakistan"/>

<entry key="3" value="USA"/>

<entry key="4" value="USA"/>

</map>

</property>

<!-- results in a setAddressProp(java.util.Properties) call -->

<property name="addressProp">

<props>

<prop key="one">INDIA</prop>

<prop key="two">Pakistan</prop>

<prop key="three">USA</prop>

<prop key="four">USA</prop>

</props>

</property>

</bean>

**Aspect Oriented Programing**

Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns.

A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

**Why use AOP?**

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

copy to clipboard1.class A{

2.public void m1(){...}

3.public void m2(){...}

4.public void m3(){...}

5.public void m4(){...}

6.public void m5(){...}

7.public void n1(){...}

8.public void n2(){...}

9.public void p1(){...}

10.public void p2(){...}

11.public void p3(){...}

12.}

class A{

public void m1(){...}

public void m2(){...}

public void m3(){...}

public void m4(){...}

public void m5(){...}

public void n1(){...}

public void n2(){...}

public void p1(){...}

public void p2(){...}

public void p3(){...}

}

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.

**Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

**Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

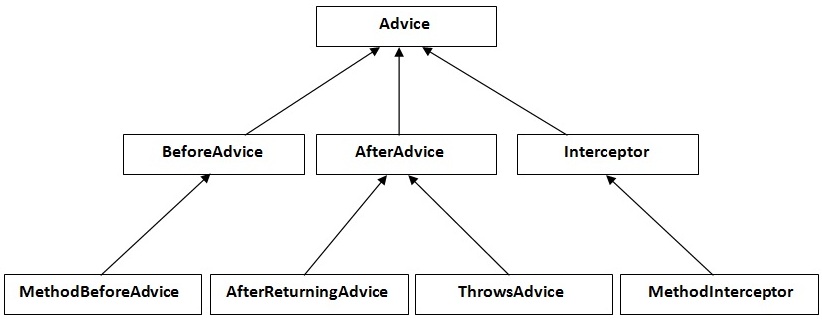
There are 4 types of advices supported in spring1.2 old style aop implementation.

1. Before Advice it is executed before the actual method call.

2. After Advice it is executed after the actual method call. If method returns a value, it is executed after returning value.

3. Around Advice it is executed before and after the actual method call.

4. Throws Advice it is executed if actual method throws exception.



If we implement BeforeAdvice it will log the information before calling the actual method and it is reverse for AfterAdvice.

1. **public** **class** BeforeAdvisor **implements** MethodBeforeAdvice{
2. @Override
3. **public** **void** before(Method method, Object[] args, Object target)**throws** Throwable {
4. System.out.println("additional concern before actual logic");
5. System.out.println("method info:"+method.getName()+" "+method.getModifiers());
6. System.out.println("argument info:");
7. **for**(Object arg:args)
8. System.out.println(arg);
9. System.out.println("target Object:"+target);
10. System.out.println("target object class name: "+target.getClass().getName());
11. }

}

Before method prototype is

**Public** **void** before(Method method, Object[] args, Object target)**throws** Throwable {

Method is actual method name i.e, in which method we are invoking, args are the method arguments and target is the class name in which we are invoking the method name.

1. <bean id="obj" **class**="com.javatpoint.A"></bean>
2. <bean id="ba" **class**="com.javatpoint.BeforeAdvisor"></bean>
4. <bean id="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean">
5. <property name="target" ref="obj"></property>
6. <property name="interceptorNames">
7. <list>
8. <value>ba</value>
9. </list>
10. </property>
11. </bean>

</beans>

Now here target object is obj (com.javatpoint.A) so before calling any method in class A then before method of BeforeAdvisor  class get invoked and then it will execute the actual method.

If we want to do the same after execution of the actual method then we just need to implement **AfterReturningAdvice** in the class BeforeAdvisor class and we need to override the afterReturning method in the class BeforeAdvisor.

**Spring MVC**

DispatcherServlet is the front controller and it is a servlet and entry point for Spring MVC Application

InternalViewResolver will take care of the forwarding the control to its view part of the application.

ModelAndView object is the model object and it will carry the data or objects to the view or jsp pages. ModelAndView objects are directly available at jsp pages, so we can fetch the information which we set at the controller classes.

HandlerMapping are used to map the request or actions to the controller classes, if we use annotations then it will automatically find the controller classes by scaning the classes.

We can get the variables which comes from url or set in the http get method (like <http://....userName=noor?Conutry=India>) we can get these variables from by using pathvariable attribute annotation.

Action method signature

@RequestMapping (“/welcome”)

Public ModelAndView helloWorld() { ............. }

If we want to get the pathVaribale

@RequestMapping (“/welcome/{userName}”) // if we give username in the curly brackets {} then it will take any name in the url like, <http://localhost.8080.com:Springs/welcome/Noor> you can give any name in the url like noor in the above example

Public ModelAndView helloWorld(@PathVariable(“username”) String Name) { // here you can use Name for manipulation ............. }

If we have so many pathVaribales then we can get the pathvaribales like

Public ModelAndView helloWorld(@PathVariable Map <Strings, Strings> pathvars) {

String Name = pathvars.get(“username”);

String LastName = pathvars.get(“lastname”);

............. }

Request Mapping on the method:

@RequestMapping (value=”helloworld” method=”RequestMethod.GET” ) //its like get method of servlet, if we specify post then its like post method of servlets.

Public ModelAndView helloWorld) { ............. }

Or we can also get the request variables using the @RequestParam annotation. If we have set of variables in the requestParameters then we can get all the request parameters using @RequestParameter Map<String, String> and we will get all the request parameters.

@RequestMapping (value=”getDetails” method=”RequestMethod.POST”)

Public ModelAndView getUserDetails(@RequestParam(“username”) String Name, @RequestParam(“LastName”) String LastName ) { ............ }

RequestParam is like RequestParameters in HttpServlet. It will give the values which set in the request.

Public ModelAndView getUserDetails(@RequestParam(valve = “username”, defaultValue=”XYZ”) String Name, @RequestParam(“LastName”) String LastName ) { ............ }

If we specify default value, when user does not give any value in the front end then it will automatically take the default value as XYZ.

And if we use @ModelAtribute in the method signature in our controller then spring automatically binds the data or the request parameters we set in jsp page to the beans. Like we have student form and in that we have two attribute studentName and studentPhno and your student bean should have exactly same variables in the student bean.

Public ModelAndView getUserDetails(@ModelAttribute(“student1”)Student student1 ) {

// now all the request parameters will be bind to the student object, and whatever you set in the Student object from the controller class it will be available in jsp page also.

If we want to set any message or any attribute which is available for all the methods in that particular controller class then we can use the @ModelAttribute just above the method. Like if we want to add any header message and which will be required in all the other methods then we can use @ModelAttribute annotation.

For example

@ModelAttribute((“student”) Student Student1)

Public void addHeader()

{ add the attribute in ModelAndView object. }

@initbinder annotation and @ResultsError Annotation need to check.

**Data Binding (Date and Collections)**

If we give the list of options to select for the user in the form like. Select Skill set

<select name=”Skill set” multiple>

<option value=”core Java” > Core Java </option>

<option value=”Springs” > Springs </option> if we specify the data type as ArrayList in the Student bean then it automatically binds the selected options to the ArrayList and we can get the same in response also.

We can also add the user defined objects. Like say if I have Student Address property and StudentAddress is a different class having county, state, pincode are the attributes of the StudentAddress class, then we will have use StudnetAdress attribute in the student class and provide setter and getter. You can get the StudentAddress Property like {Student.StudentAdress.country}

While data binding, if have int data type in the bean and if you give String in front end form then application will fail.

If we want know about the errors in the front end we can ResultError annotations.

**InitBinder**

@initBinder

Public void initBinder(webDataBinder binder){ binder.setDisallowedFields(new String[] {“studentMobile”} ) ; } now it will not bind the studentMobile field and if you print the response in your jsp page studentMobile field will be null

# Spring AOP Tutorial

1. [What is AOP?](http://www.javatpoint.com/spring-aop-tutorial)
2. [Where use AOP?](http://www.javatpoint.com/spring-aop-tutorial)
3. [AOP Concepts and Terminology](http://www.javatpoint.com/spring-aop-tutorial)
4. [Spring AOP Implementation](http://www.javatpoint.com/spring-aop-tutorial)

**Aspect Oriented Programming** (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

#### Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

1. **class** A{
2. **public** **void** m1(){...}
3. **public** **void** m2(){...}
4. **public** **void** m3(){...}
5. **public** **void** m4(){...}
6. **public** **void** m5(){...}
7. **public** **void** n1(){...}
8. **public** **void** n2(){...}
9. **public** **void** p1(){...}
10. **public** **void** p2(){...}
11. **public** **void** p3(){...}
12. }

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

**Understanding Scenario** I have to maintain log and send notification after calling methods that starts from m.

**Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

**Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

#### Where use AOP?

AOP is mostly used in following cases:

* to provide declarative enterprise services such as declarative transaction management.
* It allows users to implement custom aspects.

## AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

* Join point
* Advice
* Pointcut
* Introduction
* Target Object
* Aspect
* Interceptor
* AOP Proxy
* Weaving

#### Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

#### Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

* **Before Advice**: it executes before a join point.
* **After Returning Advice**: it executes after a joint point completes normally.
* **After Throwing Advice**: it executes if method exits by throwing an exception.
* **After (finally) Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return.
* **Around Advice**: It executes before and after a join point.

#### Pointcut

It is an expression language of AOP that matches join points.

#### Introduction

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

#### Target Object

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

#### Aspect

It is a class that contains advices, joinpoints etc.

#### Interceptor

It is an aspect that contains only one advice.

#### AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

#### Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

### AOP Implementations

AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

### Spring AOP

Spring AOP can be used by 3 ways given below. But the widely used approach is Spring AspectJ Annotation Style. The 3 ways to use spring AOP are given below:

1. [By Spring1.2 Old style (dtd based) (also supported in Spring3)](http://www.javatpoint.com/spring-aop-example)
2. [By AspectJ annotation-style](http://www.javatpoint.com/spring-aop-aspectj-annotation-example)
3. [By Spring XML configuration-style(schema based)](http://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)

Spring Transaction Management:

## Table of Contents

* [Spring overview](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#1)
* [Dependency Injection](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#2)
* [Spring Beans](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#3)
* [Spring Annotations](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#4)
* [Spring Data Access](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#5)
* [Spring Aspect Oriented Programming (AOP)](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#6)
* [Spring Model View Controller (MVC)](https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#7)

## Spring overview

### 1. What is Spring?

Spring is an open source development framework for [Enterprise Java](http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/). The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make Java EE development easier to use and promote good programming practice by enabling a [POJO-based programming model](http://www.javacodegeeks.com/2012/09/how-to-write-better-pojo-services.html).

### 2. What are benefits of Spring Framework?

* **Lightweight:**Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.
* **Inversion of control (IOC):** Loose coupling is achieved in Spring, with the [Inversion of Control technique](http://www.javacodegeeks.com/2011/08/what-is-dependency-inversion-is-it-ioc.html). The objects give their dependencies instead of creating or looking for dependent objects.
* **Aspect oriented (AOP):** [Spring supports Aspect oriented programming](http://www.javacodegeeks.com/2011/01/aspect-oriented-programming-spring-aop.html) and separates application business logic from system services.
* **Container:** Spring contains and manages the life cycle and configuration of application objects.
* **MVC Framework:** Spring’s web framework is a well-designed [web MVC framework](http://www.javacodegeeks.com/2011/02/spring-mvc-development-tutorial.html), which provides a great alternative to web frameworks.
* **Transaction Management:** Spring provides a consistent transaction management interface that can scale down to a local transaction and scale up to global transactions (JTA).
* **Exception Handling:** Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions.

### 3. Which are the Spring framework modules?

The basic modules of the Spring framework are :

* Core module
* Bean module
* Context module
* Expression Language module
* [JDBC module](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/)
* [ORM module](http://examples.javacodegeeks.com/enterprise-java/spring/jpaorm/spring-hibernate-mysql-and-maven-showcase/)
* OXM module
* Java Messaging Service(JMS) module
* Transaction module
* Web module
* Web-Servlet module
* Web-Struts module
* Web-Portlet module

### 4. Explain the Core Container (Application context) module

This is the basic Spring module, which provides the fundamental functionality of the Spring framework. BeanFactory is the heart of any spring-based application. Spring framework was built on the top of this module, which makes the Spring container.

### 5. BeanFactory – BeanFactory implementation example

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application’s configuration and dependencies from the actual application code.

The most commonly used BeanFactory implementation is the XmlBeanFactory class.

### 6. XMLBeanFactory

The most useful one is org.springframework.beans.factory.xml.XmlBeanFactory, which loads its beans based on the definitions contained in an XML file. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

### 7. Explain the AOP module

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between [Spring and other AOP frameworks](http://www.javacodegeeks.com/2014/02/applying-aspect-oriented-programming.html). This module also introduces metadata programming to Spring.

### 8. Explain the JDBC abstraction and DAO module

With the [JDBC abstraction and DAO module](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/) we can be sure that we keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. It provides a layer of meaningful exceptions on top of the error messages given by several database servers. It also makes use of Spring’s AOP module to provide transaction management services for objects in a Spring application.

### 9. Explain the object/relational mapping integration module

Spring also supports for using of an [object/relational mapping (ORM) too](http://www.javacodegeeks.com/2011/12/persistence-layer-with-spring-31-and_14.html)l over straight JDBC by providing the ORM module. Spring provides support to tie into several popular ORM frameworks, including [Hibernate](http://www.javacodegeeks.com/2010/05/jboss-42x-spring-3-jpa-hibernate.html), JDO, and [iBATIS SQL Maps](http://www.javacodegeeks.com/2012/02/mybatis-3-spring-integration-tutorial.html). Spring’s transaction management supports each of these ORM frameworks as well as JDBC.

### 10. Explain the web module

The [Spring web module](http://examples.javacodegeeks.com/enterprise-java/spring/mvc/spring-mvc-hello-world-example/) is built on the application context module, providing a context that is appropriate for web-based applications. This module also contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Jakarta Struts.

### 11. Explain the Spring MVC module

MVC framework is provided by Spring for building web applications. Spring can easily be integrated with other MVC frameworks, but [Spring’s MVC framework](http://www.javacodegeeks.com/2012/09/spring-adding-spring-mvc-part-1.html) is a better choice, since it uses IoC to provide for a clean separation of controller logic from business objects. With Spring MVC you can declaratively bind request parameters to your business objects.

### 12. Spring configuration file

Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

### 13. What is Spring IoC container?

The Spring IoC is responsible for creating the objects,managing them (with dependency injection (DI)), wiring them together, configuring them, as also managing their complete lifecycle.

### 14. What are the benefits of IOC?

IOC or dependency injection minimizes the amount of code in an application. It makes easy to test applications, since no singletons or JNDI lookup mechanisms are required in unit tests. Loose coupling is promoted with minimal effort and least intrusive mechanism. IOC containers support eager instantiation and lazy loading of services.

### 15. What are the common implementations of the ApplicationContext?

The **FileSystemXmlApplicationContext** container loads the definitions of the beans from an XML file. The full path of the XML bean configuration file must be provided to the constructor.  
The **ClassPathXmlApplicationContext** container also loads the definitions of the beans from an XML file. Here, you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.  
The **WebXmlApplicationContext:** container loads the XML file with definitions of all beans from within a web application.

### 16. What is the difference between Bean Factory and ApplicationContext?

Application contexts provide a means for resolving text messages, a generic way to load file resources (such as images), they can publish events to beans that are registered as listeners. In addition, operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context. The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable.

### 17. What does a Spring application look like?

* An interface that defines the functions.
* The implementation that contains properties, its setter and getter methods, functions etc.,
* [Spring AOP](http://examples.javacodegeeks.com/enterprise-java/spring/aop/spring-aop-example/)
* The Spring configuration XML file.
* Client program that uses the function

## Dependency Injection

### 18. What is Dependency Injection in Spring?

[Dependency Injection](http://www.javacodegeeks.com/2014/02/dependency-injection-options-for-java.html), an aspect of Inversion of Control (IoC), is a general concept, and it can be expressed in many different ways.This concept says that you do not create your objects but describe how they should be created. You don’t directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (the IOC container) is then responsible for hooking it all up.

### 19. What are the different types of IoC (dependency injection)?

* **Constructor-based dependency injection:** Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
* **Setter-based dependency injection:** Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

### 20. Which DI would you suggest Constructor-based or setter-based DI?

You can use both Constructor-based and Setter-based Dependency Injection. The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies.

## Spring Beans

### 21. What are Spring beans?

The [Spring Beans](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-3-bean-reference-example/) are Java Objects that form the backbone of a Spring application. They are instantiated, assembled, and managed by the Spring IoC container. These beans are created with the configuration metadata that is supplied to the container, for example, in the form of XML <bean/> definitions.

Beans defined in spring framework are singleton beans. There is an attribute in bean tag named "singleton" if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

### 22. What does a Spring Bean definition contain?

A Spring Bean definition contains all configuration metadata which is needed for the container to know how to create a bean, its lifecycle details and its dependencies.

### 23. How do you provide configuration metadata to the Spring Container?

There are three important methods to provide configuration metadata to the Spring Container:

* XML based configuration file.
* Annotation-based configuration
* [Java-based configuration](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-3-java-config-example/)

### 24. How do you define the scope of a bean?

When defining a <bean> in Spring, we can also declare a scope for the bean. It can be defined through the scope attribute in the bean definition. For example, when Spring has to produce a new bean instance each time one is needed, the bean’sscope attribute to be prototype. On the other hand, when the same instance of a bean must be returned by Spring every time it is needed, the the bean scope attribute must be set to singleton.

### 25. Explain the bean scopes supported by Spring

There are five scoped provided by the Spring Framework supports following five scopes:

* In **singleton** scope, Spring scopes the bean definition to a single instance per Spring IoC container.
* In **prototype** scope, a single bean definition has any number of object instances.
* In **request** scope, a bean is defined to an HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.
* In **session**scope, a bean definition is scoped to an HTTP session. This scope is also valid only in a web-aware Spring ApplicationContext.
* In **global-session** scope, a bean definition is scoped to a global HTTP session. This is also a case used in a web-aware Spring ApplicationContext.

The default scope of a Spring Bean is Singleton.

### 26. Are Singleton beans thread safe in Spring Framework?

No, singleton beans are not thread-safe in Spring framework.

### 27. Explain Bean lifecycle in Spring framework

* The spring container finds the bean’s definition from the XML file and instantiates the bean.
* Spring populates all of the properties as specified in the bean definition (DI).
* If the bean implements BeanNameAware interface, spring passes the bean’s id to setBeanName() method.
* If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
* If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization()method.
* If the bean implements IntializingBean, its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
* If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
* If the bean implements DisposableBean, it will call the destroy() method.

### 28. Which are the important beans lifecycle methods? Can you override them?

There are two important bean lifecycle methods. The first one is setup which is called when the bean is loaded in to the container. The second method is the teardown method which is called when the bean is unloaded from the container.  
The bean tag has two important attributes (init-method and destroy-method) with which you can define your own custom initialization and destroy methods. There are also the correspondive annotations(@PostConstruct and @PreDestroy).

### 29. What are inner beans in Spring?

When a bean is only used as a property of another bean it can be declared as an inner bean. Spring’s XML-based configuration metadata provides the use of <bean/> element inside the <property/> or <constructor-arg/> elements of a bean definition, in order to define the so-called inner bean. Inner beans are always anonymous and they are always scoped as prototypes.

### 30. How can you inject a Java Collection in Spring?

Spring offers the following types of [collection configuration elements](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-collections-list-set-map-and-properties-example/):

* The <list> type is used for injecting a list of values, in the case that duplicates are allowed.
* The <set> type is used for wiring a set of values but without any duplicates.
* The <map> type is used to inject a collection of name-value pairs where name and value can be of any type.
* The <props> type can be used to inject a collection of name-value pairs where the name and value are both Strings.

### 31. What is bean wiring?

Wiring, or else bean wiring is the case when beans are combined together within the Spring container. When wiring beans, the Spring container needs to know what beans are needed and how the container should use dependency injection to tie them together.

### 32. What is bean auto wiring?

The Spring container is able to [autowire relationships](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-autowire-example/) between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for a bean by inspecting the contents of the BeanFactorywithout using <constructor-arg> and <property> elements.

### 33. Explain different modes of auto wiring?

The autowiring functionality has five modes which can be used to instruct Spring container to use autowiring for dependency injection:

* **no:** This is default setting. Explicit bean reference should be used for wiring.
* **byName:** When autowiring byName, the Spring container looks at the properties of the beans on which autowireattribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
* **byType:** When autowiring by datatype, the Spring container looks at the properties of the beans on which autowireattribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown.
* **constructor:** This mode is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
* **autodetect:**Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire bybyType.

### 34. Are there limitations with autowiring?

Limitations of autowiring are:

* **Overriding:**You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
* **Primitive data types:** You cannot autowire simple properties such as primitives, Strings, and Classes.
* **Confusing nature:** Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

### 35. Can you inject null and empty string values in Spring?

Yes, you can.

## Spring Annotations

### 36. What is Spring Java-Based Configuration? Give some annotation example.

[Java based configuration](http://www.javacodegeeks.com/2013/04/spring-java-configuration.html) option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.  
An example is the @Configuration annotation, that indicates that the class can be used by the Spring IoC container as a source of bean definitions. Another example is the@Bean annotated method that will return an object that should be registered as a bean in the Spring application context.

### 37. What is Annotation-based container configuration?

An alternative to XML setups is provided by annotation-based configuration which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

### 38. How do you turn on annotation wiring?

Annotation wiring is not turned on in the Spring container by default. In order to use annotation based wiring we must enable it in our Spring configuration file by configuring <context:annotation-config/> element.

### 39. @Required annotation

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws BeanInitializationException if the affected bean property has not been populated.

### 40. @Autowired annotation

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. It can be used to autowire bean on the setter method just like @Required annotation, on the constructor, on a property or pn methods with arbitrary names and/or multiple arguments.

### 41. @Qualifier annotation

When there are more than one beans of the same type and only one is needed to be wired with a property, the@Qualifier annotation is used along with @Autowired annotation to remove the confusion by specifying which exact bean will be wired.

## Spring Data Access

### 42. How can JDBC be used more efficiently in the Spring framework?

When using the Spring JDBC framework the burden of resource management and error handling is reduced. So developers only need to write the statements and queries to get the data to and from the database. JDBC can be used more efficiently with the help of a template class provided by Spring framework, which is the JdbcTemplate (example [here](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/)).

### 43. JdbcTemplate

JdbcTemplate class provides many convenience methods for doing things such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.

### 44. Spring DAO support

The [Data Access Object (DAO) support in Spring](http://www.javacodegeeks.com/2012/09/spring-dao-and-service-layer.html) is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows us to switch between the persistence technologies fairly easily and to code without worrying about catching exceptions that are specific to each technology.

### 45. What are the ways to access Hibernate by using Spring?

There are two ways to access Hibernate with Spring:

* Inversion of Control with a Hibernate Template and Callback.
* Extending HibernateDAOSupport and Applying an AOP Interceptor node.

### 46. ORM’s Spring support

Spring supports the following ORM’s:

* Hibernate
* iBatis
* JPA (Java Persistence API)
* TopLink
* JDO (Java Data Objects)
* OJB

### 47. How can we integrate Spring and Hibernate using HibernateDaoSupport?

Use Spring’s SessionFactory called LocalSessionFactory. The integration process is of 3 steps:

* Configure the Hibernate SessionFactory
* Extend a DAO Implementation from HibernateDaoSupport
* Wire in Transaction Support with AOP

### 48. Types of the transaction management Spring support

Spring supports two types of transaction management:

* **Programmatic transaction management:** This means that you have managed the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
* **Declarative transaction management:** This means you separate [transaction management from the business code](http://www.javacodegeeks.com/2011/09/spring-declarative-transactions-example.html). You only use annotations or XML based configuration to manage the transactions.

### 49. What are the benefits of the Spring Framework’s transaction management?

* It provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
* It provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
* It supports declarative transaction management.
* It integrates very well with Spring’s various data access abstractions.

### 50. Which Transaction management type is more preferable?

Most users of the Spring Framework choose declarative transaction management because it is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container. Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code.

## Spring Aspect Oriented Programming (AOP)

### 51. Explain AOP

[Aspect-oriented programming](http://www.javacodegeeks.com/2014/02/applying-aspect-oriented-programming.html), or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management.

### 52. Aspect

The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules. It ia a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. In Spring AOP, aspects are implemented using regular classes annotated with the @Aspect annotation (@AspectJ style).

### 53. What is the difference between concern and cross-cutting concern in Spring AOP

The Concern is behavior we want to have in a module of an application. A Concern may be defined as a functionality we want to implement.  
The cross-cutting concern is a concern which is applicable throughout the application and it affects the entire application. For example, logging, [security](http://www.javacodegeeks.com/2013/04/spring-aop-in-security-controlling-creation-of-ui-components-via-aspects.html) and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

### 54. Join point

The join point represents a point in an application where we can plug-in an AOP aspect. It is the actual place in the application where an action will be taken using Spring AOP framework.

### 55. Advice

The advice is the actual action that will be taken either before or after the method execution. This is actual piece of code that is invoked during the program execution by the Spring AOP framework.

Spring aspects can work with five kinds of advice:

* **before:** Run advice before the a method execution.
* **after:** Run advice after the a method execution regardless of its outcome.
* **after-returning:** Run advice after the a method execution only if method completes successfully.
* **after-throwing:** Run advice after the a method execution only if method exits by throwing an exception.
* **around:** Run advice before and after the advised method is invoked.

### 56. Pointcut

The pointcut is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns.

### 57. What is Introduction?

An Introduction allows us to add new methods or attributes to existing classes.

### 58. What is Target object?

The target object is an object being advised by one or more aspects. It will always be a proxy object. It is also referred to as the advised object.

### 59. What is a Proxy?

A proxy is an object that is created after applying advice to a target object. When you think of client objects the target object and the proxy object are the same.

### 60. What are the different types of AutoProxying?

* BeanNameAutoProxyCreator
* DefaultAdvisorAutoProxyCreator
* Metadata autoproxying

### 61. What is Weaving? What are the different points where weaving can be applied?

Weaving is the process of linking aspects with other application types or objects to create an advised object.  
Weaving can be done at compile time, at load time, or at runtime.

### 62. Explain XML Schema-based aspect implementation?

In this implementation case, aspects are implemented using regular classes along with XML based configuration.

### 63. Explain annotation-based (@AspectJ based) aspect implementation

This implementation case (@AspectJ based implementation) refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

## Spring Model View Controller (MVC)

### 64. What is Spring MVC framework?

Spring comes with a [full-featured MVC framework for building web applications](http://examples.javacodegeeks.com/enterprise-java/spring/mvc/spring-mvc-hello-world-example/). Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring’s MVC framework uses IoC to provide a clean separation of controller logic from business objects. It also allows to declaratively bind request parameters to business objects.

### 65. DispatcherServlet

The Spring Web MVC framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.

### 66. WebApplicationContext

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

### 67. What is Controller in Spring MVC framework?

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

### 68. @Controller annotation

The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API.

### 69. @RequestMapping annotation

@RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

Ok, so now you are ready for your interview! Don’t forget to check our dedicated page full of [Spring Tutorials](http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/spring-tutorials/)!

If you enjoyed this, then [**subscribe to our newsletter**](http://eepurl.com/xRlkj) to enjoy weekly updates and complimentary whitepapers! Also, check out [**JCG Academy**](http://academy.javacodegeeks.com/) for more advanced trainin

# Interview Questions: Transaction Management

June 12, 2013

|  |
| --- |
| Qns-1: Describe Global and Local transactions in Spring. |
| Ans: Global transactions help to work with multiple transactional resources like relational database and message queue. Global transactions are managed through JTA and JNDI.  Local transactions are resource-specific like JDBC connection. Local Transactions can work with multiple transactional resources. |
|  |
| Qns-2: What is the role of TransactionDefinition interface? |
| Ans: a. Isolation b. Propagation  c. Timeout  d. Read-only status |
|  |
| Qns-3: How can we roll back a declarative transaction? |
| Ans: We can use rollback-for and no-rollback-for attributes with transactional definition. |
|  |
| Qns-4: How many types of isolation are there? |
| Ans: a. ISOLATION\_DEFAULT: default isolation.  b. ISOLATION\_READ\_COMMITTED: dirty reads are prevented, non-repeatable and phantom reads are allowed. c. ISOLATION\_READ\_UNCOMMITTED : dirty reads are allowed, no-repeatable and phantom reads are allowed.  d. ISOLATION\_REPEATABLE\_READ: dirty reads and non-repeatable reads are prevented but phantom reads are allowed.  e. ISOLATION\_SERIALIZABLE : dirty , non- repeatable reads and phantom reads are prevented. |
|  |
| Qns-5: How many types of Propagation are there? |
| Ans: Find the Propagation type. a. PROPAGATION\_MANDATORY : supports current transaction and throws exception if no transaction available.  b. PROPAGATION\_NESTED : runs with nested transaction  c. PROPAGATION\_NEVER : does not run with current transaction and throws exception if current transaction exits. d. PROPAGATION\_NOT\_SUPPORTED : runs non -transactionaly and does not support current transaction.  e. PROPAGATION\_REQUIRED : runs with current transaction and create one if does not exist.  f. PROPAGATION\_REQUIRES\_NEW : creates new transaction and suspends if exits any.  g. PROPAGATION\_SUPPORTS: runs current transaction and runs non -transactionaly |