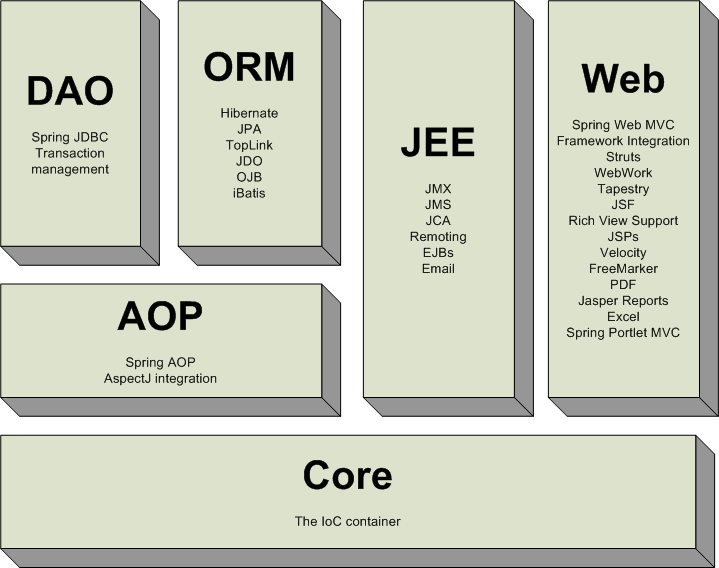
**Springs**

* The Spring Framework is an open source java framework which address the complexity involved in developing “Enterprise Web Application”
* Using Spring we can create high performing, easily testable, reusable code
* Spring is also called as **Dependency Injection OR Inversion of Control (IOC)** container
* Springs Also supports **Aspect Oriented Programming (AOP)**

**Architecture:**

The Spring Framework contains a lot of features, which are well-organized in six modules shown in the diagram below



**Spring Core:**

* The Spring Core Module is the most import component of the Spring Framework & it’s the foundation for all other modules
* This component provides the Dependency Injection features
* The ApplicationContext / BeanFactory which implements a factory pattern which separates the dependencies like initialization, creation and access of the objects from your actual program logic

**Spring ORM:**

* The Spring ORM Module helps us to interact with database via popular ORM frameworks such as Hibernate, TopLink, iBatis, etc.

**Spring Web:**

* The Spring Web module helps us to develop web application
* In Spring Web module we can also integrate other web application development frameworks such as Struts
* Spring MVC is the sub-module of Spring Web & it provides a clean separation between Controllers, models and views, and integrates with all the other features of the Spring Framework

**Steps to Setup the Spring Development Environment:**

1. Download the “spring-framework-4.1.0.RELEASE-dist.zip” file & extract it to some location

**Download Link:** <http://maven.springframework.org/release/org/springframework/spring>

1. Copy all the JAR files present under extracted folder to separate folder
2. Download the Spring support JAR files (commons-logging-1.1.1.jar) & keep it in the above folder
3. Create a Java Project in Eclipse by name “MySprings”
4. Right click on the MySprings project, go to “build path” then click on “configure build path”
5. Under “Libraries” tab click on “Add External JARs”
6. Select all the Spring & support JARs, click on Open & click on OK

**First Spring Example:**

**Capital.java**

**package** com.jspiders.springs.beans;

**public** **class** Capital

{

**private** String capitalNM;

**public** Capital() {

System.*out*.println("Initializing Capital Object");

}

/\*

\* Generate Getters & Setters

\*/

}//End of Class

**ApplicationContext.xml**

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

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

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

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

xsi:schemaLocation=

*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*

>

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

</beans>

**MyFirstSpring.java**

**import** org.springframework.context.ApplicationContext;

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

**import** com.jspiders.springs.beans.Capital;

**public** **class** MyFirstSpring

{

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

{

/\*

\* Normal Way

\*/

Capital capitalRefNormal = **new** Capital();

capitalRefNormal.setCapitalNM("Delhi");

System.*out*.println(capitalRefNormal.getCapitalNM());

/\*

\* Spring Way

\*/

ApplicationContext context

= **new** ClassPathXmlApplicationContext("applicationcontext.XML");

//Capital capitalRef = (Capital) context.getBean("capitalBean");

Capital capitalRef = (Capital) context.getBean(Capital.**class**);

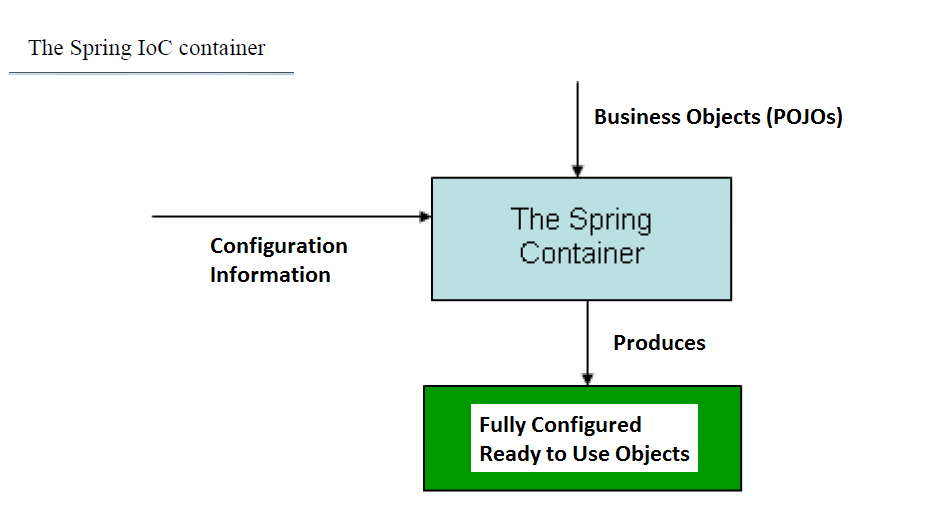
System.*out*.println(capitalRef.getCapitalNM());

}//End of Main

}//End of Class

The following diagram is a high-level view of how Spring works.

When we want create an Object with the help of Spring / Spring Container by passing the bean id present in the spring configuration file, it refer the spring configuration file, create the instance of the class defined in the config file, automatically populates the values as defined the config file & gives us the fully configured and ready to use object.



**Spring Configuration File:**

* Spring configuration file is an XML file which consists of bean declaration. It can have any name & can be present outside of the project or within the project
* In Spring we can also have more than one configuration file
* In the config file, the java bean declarations are done using “<bean>” tag which is a sub-tag of “<beans>”
* “<bean>” tag has two important attributes

1. id : It should be unique & acts like an object reference in case of java
2. class : It consists of fully qualified class name

* “<property>” tag which is a sub-tag of <bean> tag which helps us to initialize, the properties of Java Bean. However this is an optional tag & also we can have “ZERO / More” property declaration. This tag has following 3 important attributes

1. name : It consist of the exact name of the java bean property
2. value : It consist of the corresponding value for the property
3. ref : It points to other bean id if the property is of type some Object

**Note:**

* We should use either value / ref depending on the property type.
* For Literal Type we must use “value” & for Object type we must use “ref”
* We can have more than bean declaration for a particular java bean in the spring configuration file but in that case “id” attribute value must be different.

**ApplicationContext:**

* ApplicationContext is an Interface which is present in org.springframework.context package
* ApplicationContext is an Object representation of Spring Container
* Once we create ApplicationContext object it internally creates all the bean objects defined in the configuration file (only for the bean with Singleton Scope) & it initializes them
* When we invoke the getBean() method is invoked it gives these objects which are ready to use

**Add the constructor with SOP & also add the SOP in setter method. Also add certain SOP before & after the getBean Method. After running program show the console**

* Several implementations of the ApplicationContext interface are available in Spring

1. org.springframework.context.support.ClassPathXmlApplicationContext

We have to create the instance of this class if the configuration file present under “Project Classpath” (either directly under src or sub-folders of src)

1. org.springframework.context.support.FileSystemXmlApplicationContext

We have to create the instance of this class if the configuration file present in “File System” (i.e. outside of the project)

1. org.springframework.web.context.support.XmlWebApplicationContext

We have to create the instance of this class if we are developing a web application

* In case of standalone java applications we have to create the instance of either ClassPathXmlApplicationContext or FileSystemXmlApplicationContext & in case of web applications we have to create the instance of XmlWebApplicationContext
* Depending on the number of Configuration Files we have to use appropriate Constructor in the above classes

ApplicationContext appContext

= **new** ClassPathXmlApplicationContext("applicationcontext.xml");

String[] contextXMLs = **new** String[] {"abc/applicationcontext.xml"};

ApplicationContext appContext

= **new** ClassPathXmlApplicationContext(contextXMLs);

ApplicationContext appContext

= **new** FileSystemXmlApplicationContext("D:\\Praveen\\applicationcontext.XML");

**Methods Present in ApplicationContext Object**

ApplicationContext interface extends below set of Interfaces. Hence ApplicationContext is going to inherit all the methods present in the below interfaces along with having its own methods

1. org.springframework.core.env.EnvironmentCapable
2. org.springframework.beans.factory.ListableBeanFactory
3. org.springframework.beans.factory.HierarchicalBeanFactory
4. org.springframework.context.MessageSource
5. org.springframework.context.ApplicationEventPublisher
6. org.springframework.core.io.support.ResourcePatternResolver

Note: these interfaces internally extend some other interfaces

1. Object BeanFactory.getBean(String beanID) throws BeansException

* This method takes bean id information as input argument, creates an instance of the bean for the corresponding bean id by populating values as defined in the configuration file, up cast it to java.lang.Object & returns it
* We can use this method if we have more than one bean declaration for a bean in spring config file
* Return type of this method is java.lang.Object & hence if we use this method we must down cast it to desired bean class

1. <Corresponding Class Object> BeanFactory.getBean(Class classNM) throws BeansException

* This method takes class name as input argument, creates an instance of that particular class by populating values as defined in the configuration file & returns it
* We can use this method if we have Only One bean declaration for a bean in spring config file
* If we have more than one bean declaration & if we use this method then spring throws run time exception

1. <Corresponding Class Object> BeanFactory.getBean(String beanID, Class classNM) throws BeansException

* This method takes bean id & class name as input argument, creates an instance of the bean for the corresponding bean id by populating values as defined in the configuration file & returns object of the input class name

1. String[] ListableBeanFactory.getBeanDefinitionNames()

* This method returns all the bean id’s present in the Spring config file (one / more)

1. String[] ListableBeanFactory.getBeanNamesForType(Class classNM)

* This method returns all the bean id’s present in the spring config file (one / more) for the given class name

1. boolean BeanFactory.containsBean(String beanID)

* This method checks whether the bean definition is present in the spring config file for the given bean id
* This method return true if it is present otherwise return false

1. boolean BeanFactory.isPrototype(String beanID) throws NoSuchBeanDefinitionException

* This method return true, if the given bean id has the scope defined as “prototype” otherwise returns false

1. boolean BeanFactory.isSingleton(String beanID) throws NoSuchBeanDefinitionException

* This method return true, if the given bean id has the scope defined as “singleton” otherwise returns false

**org.springframework.beans.factory.BeanFactory:**

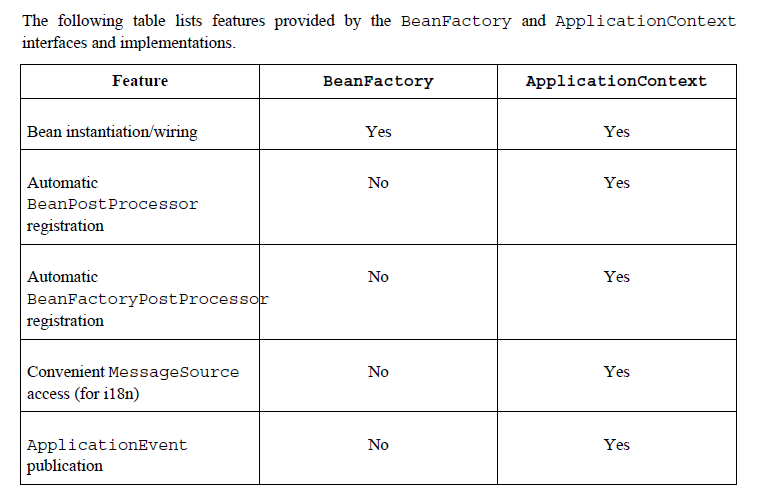
* BeanFactory is an Interface which is present in “org.springframework.beans.factory” package
* Using BeanFactory object we can also create instance of the beans which are defined in the spring config file
* Even in case of BeanFactory spring config file may be present in outside of the project or inside of the project (directly under src or subfolders of src)
* ApplicationContext is a sub-interface of BeanFactory i.e.

ApplicationContext extends ListableBeanFactory

ListableBeanFactory extends BeanFactory

**Hence ApplicationContext is a complete superset if BeanFactory**

* In short, the BeanFactory provides the basic functionality, and the ApplicationContext adds more enterprise-specific functionality
* Use ApplicationContext object unless you have a good reason for not doing so. Because the ApplicationContext includes all functionality of the BeanFactory along with other enterprise specific functionality
* Creating ApplicationContext object is a costly operation w.r.t time & resources are concerned as compared to BeanFactory. Hence if we are developing an application where memory consumption is critical & few extra KB make a difference then we can go for BeanFactory object



**import** org.springframework.core.io.Resource;

**import** org.springframework.core.io.FileSystemResource;

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

**import** org.springframework.core.io.ClassPathResource;

Resource res = **new** FileSystemResource("D:\\Praveen\\applicationcontext.XML");

XmlBeanFactory factory = **new** XmlBeanFactory(res);

ClassPathResource res = **new** ClassPathResource("applicationcontext.xml");

XmlBeanFactory factory = **new** XmlBeanFactory(res);

Interview Question :

What is the difference between Bean Factory and ApplicationContext

**Java Bean vs Spring Bean:**

**<TBD>**

**Handling Different Java Data Types in Spring:**

**public** **class** MyClass

{

**private** String name;

**private** **char** gender;

**private** **byte** houseNo;

**private** **short** pincode;

**private** **int** age;

**private** **long** mobile;

**private** **float** salary;

**private** **double** expenses;

**private** **boolean** isAlive; //not sure how to populate boolean

**private** String[] knownTechs;

//Generate Getters & Setters

}

<bean id=*"myClass"* class=*"com.vishnu.common.MyClass"*>

<property name=*"name"* value=*"Praveen"* />

<property name=*"gender"* value=*"M"* />

<property name=*"houseNo"* value=*"10"* />

<property name=*"age"* value=*"100"* />

<property name=*"pincode"* value=*"560070"* />

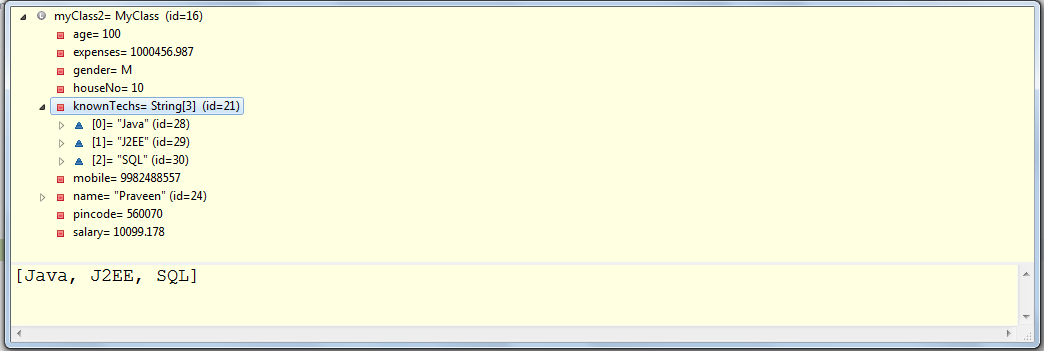
<property name=*"mobile"* value=*"9982488557"* />

<property name=*"salary"* value=*"10099.178"* />

<property name=*"expenses"* value=*"1000456.987"* />

<property name=*"knownTechs"* value=*"Java, J2EE, SQL"* />

</bean>

****

**Bean Scopes:**

* Bean Scope helps us to set the “Life Span” of the bean while creating an instance of the Bean with the help of Spring
* This can be achieved with the help of “scope” attribute present in <bean> tag. While creating the bean instance, Spring refer the scope & creates the instance accordingly.
* This approach is powerful and flexible so that we can define the scope of the objects in configuration file instead of taking care of the scope at the Java level.
* By default, Spring Framework supports five scopes. Three of which are available only for web application. We can also create a custom scope.

|  |  |
| --- | --- |
| Scope | Description |
| singleton (It’s default) | * If the scope of the bean is “singleton” then spring container creates ONLY ONE instance of that bean * It’s the default behavior of the spring container   <bean id=*"beanID"* class=*"pkg.ClassNM"* scope=*"singleton"*>  **OR**  <bean id=*"beanID"* class=*"pkg.ClassNM"*> |
| prototype | * If the scope of the bean is “prototype” then container creates new instance of the bean every time a request for that specific bean is made   <bean id=*"beanID"* class=*"pkg.ClassNM"* scope=*"prototype"*> |
| request | * If the scope of the bean is “request” then container creates new instance of the bean for every HTTP Request * Once the HTTP Response is given back, container garbage collects the instance * Functionality is similar to Request Attributes in case of Servlets   <bean id=*"beanID"* class=*"pkg.ClassNM"* scope=*"request"*> |
| session | * If the scope of the bean is “session” then container creates new instance of the bean for every new HTTP Session * Once Session expires, this object will be garbage collected * Functionality is similar to Session Attributes in case of Servlets   <bean id=*"beanID"* class=*"pkg.ClassNM"* scope=*"session"*> |
| globalSession | * If the scope of the bean is “globalSession” then container creates new instance of the bean at the time of application startup & is available till the application goes down * Functionality is similar to Context Attributes in case of Servlets   <bean id=*"beanID"* class=*"pkg.ClassNM"* scope=*"globalSession"*> |

**Thread-scoped beans:** As of Spring 3.0, a thread scope is available, but is not registered by default. For more information, see the documentation for SimpleThreadScope. For instructions on how to register this or any other custom scope, see the section called “Using a custom scope”.

**Show examples for the below Methods:**

boolean BeanFactory.**isSingleton**(String beanID) throws NoSuchBeanDefinitionException

boolean BeanFactory.**isPrototype**(String beanID) throws NoSuchBeanDefinitionException

**Dependency Injection (Inversion of Control):**

* The basic concept of the dependency injection is that we do not create objects but we describe how they should be created in a Spring config file. Also we don't directly connect dependency objects and main objects together in code but we describe which dependencies are needed by which objects in a configuration file.
* The spring container takes care of creating dependent objects while trying to create main object & inject them into main object.
* This process is fundamentally inverse in nature hence this behavior of Spring is also called as “Inversion of Control (IOC)” & this behavior is the main functionality of the Spring Framework
* DI / IOC helps us to achieve loosely coupled architecture because dependent object & main object information is provided in the configuration file & this can be changed without changing anything in the source code.
* Also this feature helps us to easily identify which are objects are dependent on what & how they are going to be get created at runtime without referring the code. Also we can easily change the dependent object information without changing anything in the code
* Dependency Injection exists in two different types

1. Setter based Dependency Injection
2. Constructor based Dependency Injection

**Setter based Dependency Injection**

* In case of Setter based Dependency Injection, dependent objects are injected into main object using setter methods of the main object

**Capital.java**

**public** **class** Capital {

**private** String capitalNM;

//Generate Getter & Setter

}//End of Class

**Country.java**

**public** **class** Country {

**private** String countryNM;

**private** Capital capitalObj;

//Generate Getter & Setter

**public** String getInfo() {

// Some Business Logic

String capitalNm = getCapitalObj().getCapitalNM();

String msg = capitalNm + " is the Capital City of " + countryNM;

**return** msg;

}

}//End of Class

**Spring Config file:-**

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<bean id=*"CountryBean"* class=*"com.jspiders.springs.beans.Country"*>

<property name=*"countryNM"* value=*"India"*/>

<property name=*"capitalObj"* ref=*"capitalBean"*/>

</bean>

**Program:**

//Normal Way :

Capital cap = **new** Capital();

cap.setCapitalNM("Delhi");

Country country = **new** Country();

country.setCountryNM("India");

// Manually Inject the Dependecny

country.setCapitalObj(cap);

System.*out*.println(country.getInfo());

//Spring Way : Dependency Injection / Inversion of Control (IOC)

ApplicationContext context

= **new** ClassPathXmlApplicationContext("applicationcontext.XML");

Country countryBeanRef = (Country) context.getBean("CountryBean");

System.*out*.println(countryBeanRef.getInfo());

**Constructor based Dependency Injection**

* In case of Constructor based Dependency Injection, dependent objects are injected into main object using parameterized constructor of the main object
* There are 4 different ways exists in Spring to pass values to constructor arguments

1. **Constructor Argument Name**
2. **Constructor Argument Index**
3. Constructor Argument Type Matching
4. Constructor Argument Resolution

**Country.java**

**public** **class** Country {

**private** String countryNM;

**private** Capital capitalObj;

**public** Country(String countryNM, Capital capitalObj) {

System.*out*.println("Initializing Country Object");

**this**.countryNM = countryNM;

**this**.capitalObj = capitalObj;

}

//Generate Getter & Setter

}//End of Class

**Spring Config file:-**

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<!-- Constructor Argument Index :

Constructor arguments can have their index specified explicitly by use

of the index attribute. Specifying an index also solves the problem of

ambiguity where a constructor may have two arguments of the same type.

-->

<bean id=*"CountryBean1"* class=*"com.jspiders.springs.beans.Country"*>

<constructor-arg index=*"0"* value=*"India"* />

<constructor-arg index=*"1"* ref=*"capitalBean"* />

</bean>

<!-- Constructor Argument Name : Introduced in Spring 3.0 -->

<bean id=*"CountryBean2"* class=*"com.jspiders.springs.beans.Country"*>

<constructor-arg name=*"countryNM"* value=*"India"* />

<constructor-arg name=*"capitalObj"* ref=*"capitalBean"* />

</bean>

<!-- Constructor Argument Type Matching :

We can use type matching with simple types by explicitly specifying the

type of the constructor argument using the 'type' attribute. -->

<bean id=*"CountryBean3"* class=*"com.jspiders.springs.beans.Country"*>

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

<constructor-arg type=*"Capital"* ref=*"capitalBean"* />

</bean>

<!-- Constructor Argument Resolution : DI occurs using the argument's type.

If there is no potential for ambiguity in the constructor arguments of a

bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments will be supplied to the appropriate constructor when it is being instantiated.

-->

<bean id=*"CountryBean4 "* class=*"com.jspiders.springs.beans.Country"*>

<constructor-arg>

<value>India</value>

</constructor-arg>

<constructor-arg>

<ref bean=*"capitalBean"* />

</constructor-arg>

</bean>

**When to use Setter OR Constructor based DI?**

* The Spring team generally advocates the usage of setter injection. Because, if a constructor has large number of arguments then it becomes cumbersome & also there might be a chance that some arguments might be optional. Also setter based DI can be easily re-configured at later time.
* But the drawback of setter injection is that it does not ensure dependency Injection. You cannot guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.
* Since you can mix both, Constructor and Setter-based DI, it is a good rule to use constructor arguments for mandatory dependencies and setters for optional dependencies.

**Autowiring in Spings:**

* Autowiring is a functionality of Spring where it automatically resolve relationships/dependencies of a Bean by inspecting the contents of the Spring config file.
* In Spring config file we can specify the autowire mode for a bean with the help of “autowire” attribute in the <bean> tag.
* The autowiring functionality has 4 modes.

1. autowire=*"none"* [it’s Default]

It means no auto-magical wiring. Bean references must be defined in the configuration file via "ref" attribute. Spring recommends this option as it makes documentation more explicit.

1. autowire=*"* *byName"*

* Autowiring by property name. This option will instruct the container to look for a bean named exactly the same as the property which needs to be autowired.
* If there is no matching bean by name found then nothing special happens. We can use dependency-check=*"objects"* to raise an error in that case.
* **In this case we can have any number of child bean definitions with different bean id in the configuration file.**

**Example:**

**public** **class** Country

{

**private** String countryNM;

**private** Capital capitalObj;

/\*

\* Generate Getter & Setter

\*/

}//End of Class

<!-- AutoWire : byName -->

<bean id=*"capitalObj"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<bean id=*"CountryBean"* class=*"com.jspiders.springs.beans.Country"*

autowire=*"byName"*>

<property name=*"countryNM"* value=*"India"*/>

</bean>

1. autowire=*"* *byType"*

* **Allows a property to be autowired if there is exactly one bean of the property type in the configuration file.**
* If there is more than one, a fatal exception is thrown, and this indicates that we can not use byType autowiring for that bean.
* If there is no matching bean by type found then nothing special happens. We can use dependency-check=*"objects"* to raise an error in that case.

<!-- AutoWire : byType -->

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<bean id=*"CountryBean"* class=*"com.jspiders.springs.beans.Country"*

autowire=*"byType"* >

<property name=*"countryNM"* value=*"India"*/>

</bean>

1. autowire=*"constructor"*

* This is analogous to byType, but applies to constructor arguments.
* If there is not exactly one bean of the constructor argument type in the bean factory, an exception is thrown.

**Example:**

**public** **class** Country

{

**private** String countryNM;

**private** Capital capitalObj;

**public** Country(String countryNM, Capital capitalObj)

{

System.*out*.println("Initializing Country Object");

**this**.countryNM = countryNM;

**this**.capitalObj = capitalObj;

}

/\*

\* Generate Getter & Setter

\*/

}//End of Class

<!-- AutoWire : constructor -->

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<bean id=*"CountryBean"* class=*"com.jspiders.springs.beans.Country"*

autowire=*"constructor"*>

<constructor-arg index=*"0"* value=*"India"* />

</bean>

**NOTE:**

In below Spring 3.0 version’s it had one more mode by name autowire=*"* *autodetect".*This mode chooses "constructor" or "byType" through introspection of the bean class. If a default constructor is found, "byType" gets applied. This mode has been disabled from Spring version 3.0 onwards

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"* >

<!-- AutoWire : autodetect -->

<bean id=*"capitalBean"* class=*"com.jspiders.springs.beans.Capital"*>

<property name=*"capitalNM"* value=*"Delhi"*/>

</bean>

<bean id=*"CountryBean"* class=*"com.jspiders.springs.beans.Country"*

autowire=*"autodetect"*>

<constructor-arg index=*"0"* value=*"India"* />

</bean>

</beans>

**About “default-autowire” attribute:**

* If we have too many beans declarations with **same autowire attribute value**, then we don't need to declare autowire attribute on each individual bean declaration. Instead spring framework provides the flexibility to apply the same autowiring style to all beans it creates by using “default-autowire” attribute in the <beans> tag as follows

<?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-autowire=*"byType"*>

<!-- Bean Declarations Goes Here -->

</beans>

* Default value for “default-autowire” is set to none i.e. we need to explicitly define the autowiring mode for each of the beans.
* “default-autowire” would be applied to all beans in a given Spring configuration file but it won’t get applied to all beans in a Spring application context. If we have multiple configuration files & each can have their own default autowiring setting. In that case, bean declarations of a particular Spring config file will get applied with corresponding autowiring mode.
* Also, even if we have “default-autowire”, it is still possible to override this value on a bean-by-bean basis using the autowire attribute.

**Spring Aware Interfaces**

Spring framework comes with lot of Aware Interfaces and all of these interfaces implement the Marker Interface by name “”

## org.s Marker superinterface indicating that a bean is eligible to be notified by the Spring container of a particular framework object through a callback-style method.

## During bootstrapping, Spring will examine each bean to determine if it implements any of the xxxAware interfaces. When one is found, it invokes the interface method, providing the piece of information that is being asked for.

If You are developing application using Spring then sometime it's important that your beans aware about Spring IOC container resources. Here, Spring Aware comes into picture.  
  
To use Aware Interfaces, You bean class should implement aware interfaces. and That's way they will be aware about container resources. I like to use these interfaces :)  
  
**Common Aware Interfaces used in Spring 3.2:**

1. BeanNameAware
2. ApplicationContextAware
3. BeanFactoryAware
4. MessageSourceAware
5. ApplicationEventPublisherAware
6. ResourceLoaderAware

## 1. BeanNameAware :  By Implementing this, Bean will be aware about it's bean name in spring ioc container.  2. ApplicationContextAware : By Implementing this, One can get current application Context while doing programatically code handling and can be used to invoke container's services 3. BeanFactoryAware : By Implementing this, One can get current bean factory which can be used to invoke container's services

## pringframework.beans.factory  Interface Aware

**All Known Subinterfaces:**

[ApplicationContextAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/ApplicationContextAware.html), [ApplicationEventPublisherAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/ApplicationEventPublisherAware.html), [BeanClassLoaderAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/BeanClassLoaderAware.html), [BeanFactoryAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/BeanFactoryAware.html), [BeanNameAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/BeanNameAware.html), [BootstrapContextAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/jca/context/BootstrapContextAware.html), [EmbeddedValueResolverAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/EmbeddedValueResolverAware.html), [EnvironmentAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/EnvironmentAware.html), [ImportAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/annotation/ImportAware.html),[LoadTimeWeaverAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/weaving/LoadTimeWeaverAware.html), [MessageSourceAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/MessageSourceAware.html), [NotificationPublisherAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/jmx/export/notification/NotificationPublisherAware.html), [PortletConfigAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/portlet/context/PortletConfigAware.html), [PortletContextAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/portlet/context/PortletContextAware.html), [ResourceLoaderAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/context/ResourceLoaderAware.html), [SchedulerContextAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/scheduling/quartz/SchedulerContextAware.html), [ServletConfigAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/context/ServletConfigAware.html), [ServletContextAware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/context/ServletContextAware.html)

public interface **Aware**

Marker superinterface indicating that a bean is eligible to be notified by the Spring container of a particular framework object through a callback-style method. Actual method signature is determined by individual subinterfaces, but should typically consist of just one void-returning method that accepts a single argument.

Note that merely implementing [Aware](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/Aware.html) provides no default functionality. Rather, processing must be done explicitly, for example in a [BeanPostProcessor](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html). Refer to ApplicationContextAwareProcessor and[AbstractAutowireCapableBeanFactory](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.html) for examples of processing \*Aware interface callbacks.

<http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/beans/factory/Aware.html>

## Spring Aware Interfaces

Sometimes we need Spring Framework objects in our beans to perform some operations, for example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That’s why spring framework provides a bunch of \*Aware interfaces that we can implement in our bean classes.

org.springframework.beans.factory.Aware is the root marker interface for all these Aware interfaces. All of the \*Aware interfaces are sub-interfaces of Aware and declare a single setter method to be implemented by the bean. Then spring context uses setter-based dependency injection to inject the corresponding objects in the bean and make it available for our use.

Spring Aware interfaces are similar to [servlet listeners](http://www.journaldev.com/1945/servlet-listener-example-servletcontextlistener-httpsessionlistener-and-servletrequestlistener) with callback methods and implementing [observer design pattern](http://www.journaldev.com/1739/observer-design-pattern-in-java-example-tutorial).

Some of the important Aware interfaces are:

* **ApplicationContextAware** – to inject ApplicationContext object, example usage is to get the array of bean definition names.
* **BeanFactoryAware** – to inject BeanFactory object, example usage is to check scope of a bean.
* **BeanNameAware** – to know the bean name defined in the configuration file.
* **ResourceLoaderAware** – to inject ResourceLoader object, example usage is to get the input stream for a file in the classpath.
* **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
* **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.
* When an ApplicationContext creates a bean / object that implements the ApplicationContextAware interface, the class is provided with a reference of ApplicationContext.
* One use would be the programmatic retrieval of other beans. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style. They may be required by some beans that require programmatic access to the container.
* Thus beans can manipulate programmatically the ApplicationContext that created them, through the ApplicationContext interface, or by casting the reference to a known subclass of this interface, such as ConfigurableApplicationContext, which exposes additional functionality
* As of Spring 2.5, autowiring is another alternative to obtain the reference of ApplicationContext.
* Spring provides lot of “Aware Interfaces” (BeanFactoryAware, BeanNameAware, etc) & usage of these interfaces ties your code to the Spring API and does not follow the Inversion of Control style.

**AnotherCountry.java**

**public** **class** AnotherCountry **implements** ApplicationContextAware

{

**private** String countryNM;

**private** ApplicationContext context;

**public** AnotherCountry()

{

System.*out*.println("Initializing Another Country Object");

}

**public** String getCountryNM() {

**return** countryNM;

}

**public** **void** setCountryNM(String countryNM) {

**this**.countryNM = countryNM;

}

**public** String getInfo()

{

/\*

\* Dependency Injection : Using ApplicationContextAware Interface

\*/

Capital capitalObj = (Capital) context.getBean("capitalBean");

String capitalNm = capitalObj.getCapitalNM();

String msg = capitalNm + " is the Capital City of " + countryNM;

**return** msg;

}

@Override

**public** **void** setApplicationContext(ApplicationContext context)

**throws** BeansException

{

**this**.context = context;

}

}//End of Class

**Capital.java :**  Same as Above

**Application-Context.xml**

<bean id=*"anotherCountry"* class=*"com.jspiders.springs.beans.AnotherCountry"*>

<property name=*"countryNM"* value=*"India"* />

</bean>

**ApplicationContextAwareEx.java**

ApplicationContext context

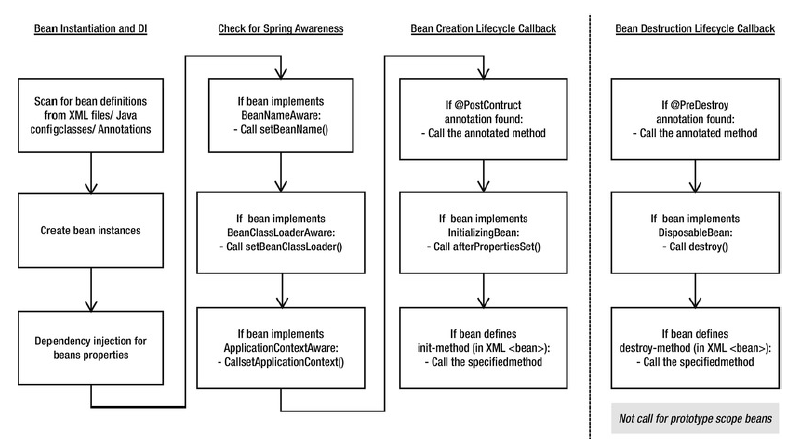
= **new** ClassPathXmlApplicationContext("applicationcontext.XML");

AnotherCountry anotherCountryRef

= (AnotherCountry) context.getBean("anotherCountry");

System.*out*.println(countryBeanRef.getInfo());

**Spring Bean Lifecycle OR Spring Lifecycle Callbacks Methods:**

****

In contrast, the lifecycle of a bean within a Spring container is more elaborate. It’s important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created. Figure 1.5 shows the startup lifecycle of a typical bean as it’s loaded into a Spring application context. As you can see, a bean factory performs several setup steps before a bean is ready to use. Breaking down figure 1.5 in more detail:

**1** Spring instantiates the bean.

**2** Spring injects values and bean references into the bean’s properties.

**3** If the bean implements BeanNameAware, Spring passes the bean’s ID to the set- BeanName() method.

**4** If the bean implements BeanFactoryAware, Spring calls the setBeanFactory() method, passing in the bean factory itself. DisposableBean’s destroy() Call custom destroy-method **Bean is ready to use Container is shut down Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.** Licensed to Christian Cederquist <chrisman@kaus.dk> **20 CHAPTER 1 *Springing into action***

**5** If the bean implements ApplicationContextAware, Spring will call the set- ApplicationContext() method, passing in a reference to the enclosing appli- cation context.

**6** If any of the beans implement the BeanPostProcessor interface, Spring calls their postProcessBeforeInitialization() method.

**7** If any beans implement the InitializingBean interface, Spring calls their afterPropertiesSet() method. Similarly, if the bean was declared with an init-method, then the specified initialization method will be called.

**8** If there are any beans that implement BeanPostProcessor, Spring will call their postProcessAfterInitialization() method.

**9** At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed. **10** If any beans implement the DisposableBean interface, then Spring will call their destroy() methods. Likewise, if any bean was declared with a destroy- method, then the specified method will be called.

* Sometimes we may be required to perform some initialization when a bean is instantiated like opening a DB Connection or reading the data from property file. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required like closing the DB Connection or closing the text file etc.,
* In Spring, the Lifecycle of a Bean is controlled by Spring Container & it provides following ways for controlling life cycle events of bean:

1. By implementing InitializingBean and DisposableBean callback interfaces
2. By using “init-method” & “destroy-method” attributes in <bean> tag
3. By using @PostConstruct and @PreDestroy annotations in a Bean

* In Spring’s we can provide lifecycle methods to the beans. This allows the bean to perform certain actions upon initialization and destruction of that particular bean.

1. **By implementing InitializingBean and DisposableBean callback interfaces :**

* The org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The InitializingBean interface has only one method within it. Hence if a Bean implements this interface it has to provide implementation for this method

**void** afterPropertiesSet() **throws** Exception;

* Similarly, implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to perform cleanup once the object being garbage collected. The DisposableBean interface has only one method within it. Hence if a Bean implements this interface it has to provide implementation for this method

**void** destroy() **throws** Exception;

**public** **class** LifeCycleBean1 **implements** InitializingBean, DisposableBean

{

@Override

**public** **void** afterPropertiesSet() **throws** Exception {

System.*out*.println("Inside Init Method");

}

**public** **void** myMethod() {

System.*out*.println("Inside my Method");

}

@Override

**public** **void** destroy() **throws** Exception {

System.*out*.println("Inside Destroy Method");

}

}//End of Class

<bean id=*"lifeCycleBean1"* class=*"com.jspiders.springs.beans.LifeCycleBean1"* />

ClassPathXmlApplicationContext appContext

= **new** ClassPathXmlApplicationContext("applicationcontext.xml");

LifeCycleBean1 lifeCycleBean1

= (LifeCycleBean1) appContext.getBean("lifeCycleBean1");

lifeCycleBean1.myMthod();

appContext.close();

1. **By using “init-method” & “destroy-method” attributes in <bean> tag**

* We can define our own methods as life cycle methods in <bean> tag with the help of “init-method” and “destroy-method” attribute values for the bean in the spring config file.

**public** **class** LifeCycleBean2

{

**public** **void** myInitMethod() {

System.*out*.println("LifeCycleBean2 : Inside Init Method");

}

**public** **void** myOtherMethod() {

System.*out*.println("LifeCycleBean2 : Inside my Method");

}

**public** **void** myDestroyMethod() {

System.*out*.println("LifeCycleBean2 : Inside Destroy Method");

}

}//End of Class

<bean id=*"lifeCycleBean2"* class=*"com.jspiders.springs.beans.LifeCycleBean2"*

init-method=*"myInitMethod"*

destroy-method=*"myDestroyMethod"*/>

**Default Initialization and destroy method:**

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

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*

default-init-method=*"myInit"*

default-destroy-method=*"myDestroy"*>

<!-- Bean Declaration Goes Here -->

</beans>

1. **By using @PostConstruct and @PreDestroy annotations**

* Spring framework also support @PostConstruct and @PreDestroy annotations for defining life cycle methods. These annotations are part of javax.annotation package.
* @PostConstruct annotated method will be invoked after the bean has been instantiated, initialized and just before its instance is returned to requesting object.
* @PreDestroy annotated method is called just before the bean is about be destroyed by spring container.
* However for these annotations to work, we need to configure our spring application to look for annotations. We can do this either by defining bean of type org.springframework.context.annotation.CommonAnnotationBeanPostProcessor or by context:annotation-config element in spring bean configuration file.

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**public** **class** LifeCycleBean3

{

@PostConstruct

**public** **void** myInitMethod() {

System.*out*.println("LifeCycleBean3 : Inside Init Method");

}

**public** **void** myNormalMethod() {

System.*out*.println("LifeCycleBean3 : Inside my Method");

}

@PreDestroy

**public** **void** myDestroyMethod() {

System.*out*.println("LifeCycleBean3 : Inside Destroy Method");

}

}//End of Class

<bean class=*"org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"* />

<bean id=*"lifeCycleBean3"* class=*"com.jspiders.springs.beans.LifeCycleBean3"* />

**OR**

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

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

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

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"* >

<context:annotation-config />

<bean id=*"lifeCycleBean3"* class=*"com.jspiders.springs.beans.LifeCycleBean3"* />

<!-- Other Bean’s Declaration Goes Here -->

</beans>

**Initializing collections in spring**

Spring supports initializing the following java collections via Sping config file.

1. List
2. Set
3. Map
4. Properties

**HandlingCollectionsInSpring.java**

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Properties;

**import** java.util.Set;

**public** **class** HandlingCollectionsInSpring

{

**private** List<Object> myList;

**private** Set<Object> mySet;

**private** Map<Object, Object> myMap;

**private** Object[] myArray;

**private** Properties myProps;

/\*

\* Generate Getters & Setters

\*/

}

**Application-context.xml**

<beans>

<bean id=*"PersonBean"* class=*"com.mkyong.common.Person"*>

<property name=*"name"* value=*"mkyong1"* />

<property name=*"age"* value=*"28"* />

</bean>

<bean id=*"CustomerBean"* class=*"com.mkyong.common.Customer"*>

<!-- java.util.List -->

<property name=*"myList"*>

<list>

<value>1</value>

<ref bean=*"PersonBean"* />

<bean class=*"com.mkyong.common.Person"*>

<property name=*"name"* value=*"mkyongList"* />

<property name=*"age"* value=*"28"* />

</bean>

</list>

</property>

<!-- java.util.Set -->

<property name=*"mySet"*>

<set>

<value>1</value>

<ref bean=*"PersonBean"* />

<bean class=*"com.mkyong.common.Person"*>

<property name=*"name"* value=*"mkyongSet"* />

<property name=*"age"* value=*"28"* />

</bean>

</set>

</property>

<!-- java.util.Map -->

<property name=*"myMap"*>

<map>

<entry key=*"Key 1"* value=*"1"* />

<entry key=*"Key 2"* value-ref=*"PersonBean"* />

<entry key=*"Key 3"*>

<bean class=*"com.mkyong.common.Person"*>

<property name=*"name"* value=*"mkyongMap"* />

<property name=*"age"* value=*"28"* />

</bean>

</entry>

</map>

</property>

<!-- java.util.Properties -->

<property name=*"myProps"*>

<props>

<prop key=*"admin"*>admin@nospam.com</prop>

<prop key=*"support"*>support@nospam.com</prop>

</props>

</property>

<property name=*"PersonBean"*>

<array>

<value>Java</value>

<ref bean=*"capitalRef"*/>

<bean class=*"com.mkyong.common.Person"*>

<property name=*"name"* value=*"mkyongSet"* />

<property name=*"age"* value=*"28"* />

</bean>

</array>

</property>

</bean>

</beans>

**SpringTest.java**

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

{

ApplicationContext context

= **new** ClassPathXmlApplicationContext("SpringBeans.xml");

Customer cust = (Customer)context.getBean("CustomerBean");

System.*out*.println(cust); //Show the Object in a Debugger Mode

}



### 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.

**Spring Expression Language**

* Spring 3 introduced the Spring Expression Language (Spring EL), a powerful & neat way of wiring values into a bean’s properties or constructor arguments using expressions that are evaluated at runtime i.e. bean creation time.
* Using Spring EL, **we can accomplish different ways of bean wiring that would be much more difficult (or even impossible) using spring’s traditional wiring style**. Couple of them are,

1. The ability to reference beans by their ID
2. Invoking methods and accessing properties on objects
3. Mathematical, relational, and logical operations on values
4. Regular expression matching
5. Collection manipulation

value=*"#{ null }"*

**Spring Hibernate Integration**

<bean id=*"hibernateDAObean"* class=*"com.jspiders.dao.StudentDAOHibernateImpl"*>

<property name=*"factory"* ref=*"factoryBean"* />

</bean>

<bean id=*"factoryBean"*

class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"configLocation"* value=*"hibernate.cfg.xml"* />

</bean>

**public** **interface** StudentDAOInterface

{

StudentsInfoBean getStudentData(**int** regno);

**boolean** createStudentProfile(StudentsInfoBean data);

}

**public** **class** StudentDAOHibernateImpl

**implements** StudentDAOInterface

{

**private** SessionFactory factory = **null**;

**public** SessionFactory getFactory() {

**return** factory;

}

**public** **void** setFactory(SessionFactory factory) {

**this**.factory = factory;

}

**public** StudentsInfoBean getStudentData(**int** regno)

{

Session session = factory.openSession();

//4. Operate with DB

Transaction txn = session.beginTransaction();

StudentsInfoBean bean

= (StudentsInfoBean)session.get(StudentsInfoBean.**class**, regno);

txn.commit();

**return** bean;

}//End of getStudentData

**public** **boolean** createStudentProfile(StudentsInfoBean data)

{

Transaction txn = **null**;

**try**

{

Session session = factory.openSession();

txn = session.beginTransaction();

session.saveOrUpdate(data);

txn.commit();

**return** **true**;

} **catch** (HibernateException e) {

txn.rollback();

**return** **false**;

}

}//End of createStudentProfile

}//End of Class

**public** **class** SpringHibernateIntegration\_MainMethodClass {

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

{

ApplicationContext context = **new** ClassPathXmlApplicationContext("app-context.xml");

StudentDAOInterface dao = context.getBean(StudentDAOHibernateImpl.**class**);

/\*

\* Select

\*/

StudentsInfoBean bean = dao.getStudentData(1);

**if**(bean==**null**)

{

System.*err*.println("Record Does Not Exists !!!");

}**else**{

System.*out*.println("Record Exists ...");

System.*out*.println("Reg. No. : "+bean.getRegno());

System.*out*.println("First Name : "+bean.getFirstNM());

System.*out*.println("Middle Name : "+bean.getMiddleNM());

System.*out*.println("Last Name : "+bean.getLastNM());

}

/\*

\* Insert

\*/

StudentsInfoBean bean2 = **new** StudentsInfoBean();

bean2.setFirstNM("Praveen 123");

bean2.setMiddleNM("NA 123");

bean2.setLastNM("Dyamappa 123");

bean2.setRegno(1);

dao.createStudentProfile(bean2);

StudentsInfoBean bean3 = dao.getStudentData(1);

**if**(bean3==**null**)

{

System.*err*.println("Record Does Not Exists !!!");

}**else**{

System.*out*.println("Record Exists ...");

System.*out*.println("Reg. No. : "+bean3.getRegno());

System.*out*.println("First Name : "+bean3.getFirstNM());

System.*out*.println("Middle Name : "+bean3.getMiddleNM());

System.*out*.println("Last Name : "+bean3.getLastNM());

}

}//End of Main

}//End of Class

**About LocalSessionFactoryBean & its declaration in Spring config file**

**OR**

**SessionFactory setup in a Spring container**

**public class LocalSessionFactoryBean**

Revisit the HibernateUtil.java which we created in the StudentsAppHibernate Web application & then tell how exactly it creates SessionFactory object. Similarly LocalSessionFactoryBean functionlaity wise its exactly as same as this class.

* as we know the objects which needs to interact with DB using hibernate should get org.hibernate.Session object with the help of SessionFactory object
* In Spring, **org.springframework.orm.hibernate3.LocalSessionFactoryBean** produces a SessionFactory object, the SessionFactory object can then be passed to Hibernate-based DAOs classes via dependency injection.
* In short, the one & only job of LocalSessionFactoryBean is to produce a SessionFactory object.
* As we know in case of hibernate, to create the SessionFactory object we need to load the config file(s) & Hibernate Mapping file(s). Hence LocalSessionFactoryBean object has a property by name “*configLocation*” which helps us to provide the “Hibernate Config file location”

Quite simple here. the LocalSessionFactoryBean implements FactoryBean, which is a special interface for Spring. Spring sees that the type of that class is a FactoryBean, so instead of creating the instance of LocalSessionFactoryBean, it calls the special public Object getObject() method, and the object that that returns is what is given out. It is the Factory [pattern](http://www.javaranch.com/patterns/) at work. So LocalSessionFactoryBean's getObject() method returns a SessionFactory object.

To avoid tying application objects to hard-coded resource lookups, you can define resources such as

a JDBC DataSource or a Hibernate SessionFactory as beans in the Spring container. Application

objects that need to access resources receive references to such predefined instances through bean

references

The following excerpt from an XML application context definition shows how to set up a JDBC

DataSource and a Hibernate SessionFactory on top of it:

In this case,

you need to define a Hibernate LocalSessionFactoryBean, which your application code will use to

obtain Hibernate Session instances.

**<beans>**

**<bean id**=**"myDataSource" class**=**"org.apache.commons.dbcp.BasicDataSource" destroymethod**=**"**

**close">**

**<property name**=**"driverClassName" value**=**"org.hsqldb.jdbcDriver"/>**

**<property name**=**"url" value**=**"jdbc:hsqldb:hsql://localhost:9001"/>**

**<property name**=**"username" value**=**"sa"/>**

**<property name**=**"password" value**=**""/>**

**</bean>**

**<bean id**=**"mySessionFactory" class**=**"org.springframework.orm.hibernate3.LocalSessionFactoryBean">**

**<property name**=**"dataSource" ref**=**"myDataSource"/>**

**<property name**=**"mappingResources">**

**<list>**

**<value>**product.hbm.xml**</value>**

**</list>**

**</property>**

**<property name**=**"hibernateProperties">**

**<value>**

hibernate.dialect=org.hibernate.dialect.HSQLDialect

**</value>**

**</property>**

**</bean>**

**</beans>**

Spring's LocalSessionFactoryBean supports Hibernate's

SessionFactory.getCurrentSession() method for any Spring transaction strategy, returning

the current Spring-managed transactional Session

**Implementing DAOs based on plain Hibernate 3 API**

A corresponding DAO implementation resembles the following example,

based on the plain Hibernate API

**public class** ProductDaoImpl **implements** ProductDao

{

**private** SessionFactory sessionFactory;

**public void** setSessionFactory(SessionFactory sessionFactory) {

**this**.sessionFactory = sessionFactory;

}

**public** Collection loadProductsByCategory(String category)

{

**return this**.sessionFactory.getCurrentSession()

.createQuery(***"from test.Product product where product.category=?"***)

.setParameter(0, category)

.list();

}

}

This style is similar to that of the Hibernate reference documentation and examples, except for holding

the SessionFactory in an instance variable. We strongly recommend such an instance-based setup

over the old-school static HibernateUtil class from Hibernate's CaveatEmptor sample application.

(In general, do not keep any resources in static variables unless *absolutely* necessary.)

The above DAO follows the dependency injection pattern: it fits nicely into a Spring IoC container, just

as it would if coded against Spring's HibernateTemplate. Of course, such a DAO can also be set up

in plain Java (for example, in unit tests). Simply instantiate it and call setSessionFactory(..) with

the desired factory reference. As a Spring bean definition, the DAO would resemble the following:

**<beans>**

**<bean id**=**"myProductDao" class**=**"product.ProductDaoImpl">**

**<property name**=**"sessionFactory" ref**=**"mySessionFactory"/>**

**</bean>**

**</beans>**

**Spring MVC (Model View Controller)**

1. **Web.xml changes:**

<servlet>

<servlet-name>sprighibernateApp</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

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

<param-value>

/WEB-INF/config/sprighibernateApp-Application-Context.xml

OR

/WEB-INF/sprighibernateApp-Application-Context.xml

</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>sprighibernateApp</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

1. **sprighibernateApp-Application-Context.xml**

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

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

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

<context:component-scan

base-package=*"com.jspiders.springhibernateapp.controllers"* />

<bean

id=*"jspViewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

<!-- This looks for the JSP in directly under Webcontent -->

<property name=*"prefix"* value=*"/"* />

OR

<!-- This looks for the JSP in subdirectory "jsp" under Webcontent -->

<property name=*"prefix"* value=*"/jsp/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

<bean id=*"hibernateDAObean"* class=*"com.jspiders.springhibernateapp.dao.StudentDAOHibernateImpl"*>

<property name=*"factory"* ref=*"factoryBean"* />

</bean>

<bean id=*"factoryBean"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"configLocation"* value=*"hibernate.cfg.xml"* />

</bean>

</beans>

1. **StudentProfileController:**

**package** com.jspiders.springhibernateapp.controllers;

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.servlet.ModelAndView;

**import** com.jspiders.springhibernateapp.dao.StudentDAOHibernateImpl;

@Controller

**public** **class** StudentProfileController {

@Autowired

**private** StudentDAOHibernateImpl dao;

@RequestMapping(value="/listStudents", method=RequestMethod.*GET*)

**public** ModelAndView listStudents()

{

Map<String, Object> model = **new** HashMap<String, Object>();

model.put("students", dao.listStudents());

//model.put("obj", new Object());

**return** **new** ModelAndView("studentsList", model);

}

@RequestMapping(value="/search", method=RequestMethod.*POST*)

**public** ModelAndView getStudent(@RequestParam **int** regno)

{

Map<String, Object> model = **new** HashMap<String, Object>();

model.put("student", dao.searchStudent(regno) );

**return** **new** ModelAndView("searchResult", model);

}

@RequestMapping(value="/studentRegistration", method=RequestMethod.*POST*)

**public** ModelAndView createProfile(@ModelAttribute StudentsInfoBean formData)

{

Map<String, Boolean> model = **new** HashMap<String, Boolean>();

model.put("result", daoService.studentRegistration(formData) );

**return** **new** ModelAndView("ProfileCreationResult", model);

}

}//End of Class

1. **studentsList.jsp:-**

<%@page import=*"com.jspiders.springhibernateapp.dao.StudentsInfoBean"*%>

<%@page import=*"java.util.List"*%>

<%@page import=*"java.util.Map"*%>

<html>

<body>

<%

List<StudentsInfoBean> dataList

= (List<StudentsInfoBean>)request.getAttribute("students");

Object obj = request.getAttribute("obj");

**for**(StudentsInfoBean data : dataList)

{

%>

Some Val : <%= obj.toString() %>

Reg. No. : <%= data.getRegno()%>

<%} %>

</body>

</html>

1. **searchResult.jsp:-**

<%@page import=*"com.jspiders.springhibernateapp.dao.StudentsInfoBean"*%>

<html>

<body>

<%

StudentsInfoBean data

= (StudentsInfoBean)request.getAttribute("student");

**if**(data!=**null**)

{

%>

<table>

<tr bgcolor=*"green"*>

<td>Reg. No.</td>

<td>First Name</td>

<td>Middle Name</td>

<td>Last Name</td>

</tr>

<tr>

<td><%= data.getRegno()%></td>

<td><%= data.getFirstNM()%></td>

<td><%= data.getMiddleNM()%></td>

<td><%= data.getLastNM()%></td>

</tr>

</table>

<%

}**else**{

%>

<font color=*"red"*>No Records Found !!!</font>

<%

}

%>

<BR><BR><BR>

<%@include file=*"./Search.html"* %>

</body>

</html>

**About DispatcherServlet & its declaration in Spring config file**

* This class is a sub-class of HttpServlet acts like a central dispatcher for all the HTTP Requests for the entire web application
* It accepts the request & dispatches to the appropriate controller

**Abut InternalResourceViewResolver & its declaration in Spring config file**

**About @Controller**

**About @RequestMapping**

**About @RequestParam**

**About @ModelAttribute**

**About ModelAndView**

**Redirect in Spring MVC**

**Forward in Spring MVC**

**Sending & Receiving files using Spring MVC**

**Sending mail using Spring**

* Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @RequestMapping, @RequestParam, @ModelAttribute, and so on.

@Controller

**public class** HelloWorldController {

@RequestMapping("/helloWorld")

**public** String helloWorld(Model model) {

model.addAttribute(***"message"***, ***"Hello World!"***);

**return *"helloWorld"***;

}

}

* As you can see, the @Controller and @RequestMapping annotations allow flexible method names and signatures.
* In this particular example the method accepts a Model and returns a view name as a String
* @Controller and @RequestMapping and a number of other annotations form the basis for the Spring MVC implementation.
* 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.
* The @Controller annotation acts as a stereotype for the annotated class, indicating its role.
* The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations
* You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.
* To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the spring-context schema as shown in the following XML snippet:

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

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

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

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

**xsi:schemaLocation**=**"**

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

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

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

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

**<context:component-scan base-package**=**"org.springframework.samples.petclinic.web"/>**

*<!-- ... -->*

**</beans>**

* You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

@Controller

**@RequestMapping("/appointments")**

**public class** AppointmentsController

{

**private final** AppointmentBook appointmentBook;

@Autowired

**public** AppointmentsController(AppointmentBook appointmentBook) {

**this**.appointmentBook = appointmentBook;

}

**@RequestMapping(method = RequestMethod.GET)**

**public** Map<String, Appointment> get() {

**return** appointmentBook.getAppointmentsForToday();

}

**@RequestMapping(value="/{day}", method = RequestMethod.GET)**

**public** Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE)

Date day, Model model) {

**return** appointmentBook.getAppointmentsForDay(day);

}

**@RequestMapping(value="/new", method = RequestMethod.GET)**

**public** AppointmentForm getNewForm() {

**return new** AppointmentForm();

}

**@RequestMapping(method = RequestMethod.POST)**

**public** String add(@Valid AppointmentForm appointment, BindingResult result) {

**if** (result.hasErrors()) {

**return *"appointments/new"***;

}

appointmentBook.addAppointment(appointment);

**return *"redirect:/appointments"***;

}

}

* @RequestMapping on the type (class) level indicates that all handling methods on this controller are relative to the /

appointments path.

* The get() method has a further @RequestMapping refinement: it only accepts GET requests, meaning that an HTTP GET for /appointments invokes this method. The post() has a similar refinement, and the getNewForm() combines the definition of HTTP method and path into one, so that GET requests for appointments/new are handled by that method.
* The getForDay() method shows another usage of @RequestMapping: URI templates.
* A @RequestMapping on the class level is not required. Without it, all paths are simply absolute, and not relative.

**URI Template Patterns**

*URI templates* can be used for convenient access to selected parts of a URL in a @RequestMapping method.

A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

For example, the URI Template http://www.example.com/users/{userId} contains the variable *userId*. Assigning the value *fred* to the variable yields <http://www.example.com/users/fred>.

In Spring MVC you can use the @PathVariable annotation on a method argument to bind it to the value of a URI template variable:

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)

**public** String findOwner(**@PathVariable** String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute(***"owner"***, owner);

**return *"displayOwner"***;

}

The URI Template "/owners/{ownerId}" specifies the variable name ownerId. When the controller handles this request, the value of ownerId is set to the value found in the appropriate part of the URI. For example, when a request comes in for /owners/fred, the value of ownerId is fred.

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)

**public** String findOwner(**@PathVariable**(***"ownerId"***) String theOwner, Model model) {

*// implementation omitted*

}

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)

**public** String findOwner(**@PathVariable** String ownerId, Model model) {

*// implementation omitted*

}

A method can have any number of @PathVariable annotations

@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)

**public** String findPet(**@PathVariable** String ownerId, **@PathVariable** String petId, Model

model)

{

Owner owner = ownerService.findOwner(ownerId);

Pet pet = owner.getPet(petId);

model.addAttribute(***"pet"***, pet);

**return *"displayPet"***;

}

When a @PathVariable annotation is used on a Map<String, String> argument, the map is populated with all URI template variables.

Add Example

A URI template can be assembled from type and path level *@RequestMapping* annotations. As a result the findPet() method can be invoked with a URL such as /owners/42/pets/21.

@Controller

@RequestMapping(**"/owners/{ownerId}"**)

**public class** RelativePathUriTemplateController {

@RequestMapping(**"/pets/{petId}"**)

**public void** findPet(@PathVariable String ownerId, @PathVariable String petId, Model

model)

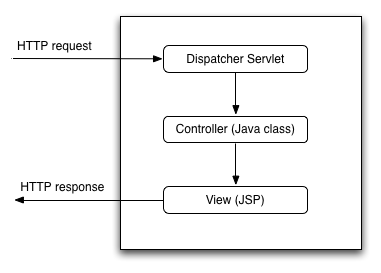
{

*// implementation omitted*

}

}

A @PathVariable argument can be of **any simple type** such as int, long, Date, etc. Spring automatically converts to the appropriate type or throws a TypeMismatchException if it fails to do so.



The job of the *DispatcherServlet* is to take an incoming URI and find the right combination of handlers (generally methods on *Controller* classes) and views (generally JSPs) that combine to form the page or resource that's supposed to be found at that location.

I might have

* a file /WEB-INF/jsp/pages/Home.jsp
* and a *method* on a class
* @RequestMapping(value="/pages/Home.html")
* private ModelMap buildHome() {
* return somestuff;

}

The *Dispatcher servlet* is the bit that "knows" to call that method when a browser requests the page, and to combine its results with the matching JSP file to make an html document.

How it accomplishes this varies widely with configuration and Spring version.

There's also no reason the end result has to be web pages. It can do the same thing to locate *RMI*end points, handle *SOAP* requests, anything that can come into a servlet.

DispatcherServlet is Spring MVC's implementation of the [front controller pattern](http://www.oracle.com/technetwork/java/frontcontroller-135648.html).

See description in the Spring docs [here](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-servlet).

Essentially, it's a servlet that takes the incoming request, and delegates processing of that request to one of a number of handlers, the mapping of which is specific in the DispatcherServletconfiguration.

We can say like DispatcherServlet taking care of every thing in Spring MVC.

At web container start up

1)DispatcherServel will be loaded and initialized by calling init() method.

2) init() of DispatcherServlet will try to identify the Spring Configuration Document with naming conventions like "servlet\_name-servlet.xml" then all beans can be indentify.

Ex:

public class DispatcherServlet extends HttpServlet{

ApplicationContext ctx=null;

public void init(ServletConfig cfg){

//1.try to get the spring configuration document with default naming conventions

String xml="servlet\_name"+"-servlet.xml";

//if it was found then creates the ApplicationContext object

ctx=new XmlWebApplicationContext(xml);

}

...

}

so in generally DispatcherServlet capture request URI and hand over to HandlerMapping,HandlerMapping search mapping bean with method of controller, where controller returning logical name(view) then this logical name is send to DispatcherServlet by HandlerMapping then DispatcherServlet tell viewResolver to give full location of view by appending prefix and suffix,then DispatcherServlet give view to the client.