**Spring Introduction:**

**Developer Name:** It is developed by Rod Jonson

**Organization Name:** Initially spring organization was Spring Org but latter it was taken over by pivoted organization. So now every new version is release by Pivoted org only.

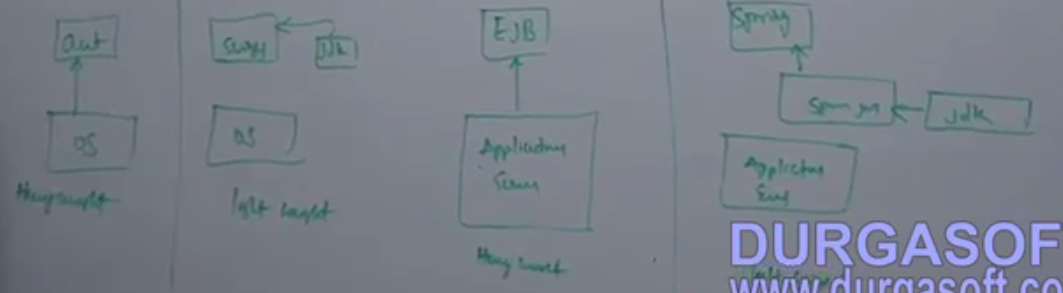
**Initial Name:** Spring initial name was Interface 21 latter they change it to spring.

**Why the spring Name is spring?**

Spring is the replacement of EJB i.e. it is an alternative of EJB (Enterprise Java Beans). In case of EJB the application developed in EJB is heavy weight and tightly coupled. But by using spring we can develop the application light weight and loosely coupled and we can reduce investment cost also. We no need to by the server and we no need to buy the services. Spring freely provides the services, so we can use these services for our enterprise applications. EJB is suitable for banking application. But of small scale e-commerce application (one lac or 2 lac) EJB is not suitable. We need to have some free sources and spring is free source. So spring is compared to EJB and EJBs are like winter season and spring comes after winter season that is why they gave this name as spring. In US winter season is not fine because temperature goes to -20 and spring is good for US people i.e. EJB is like winter season. So in sort we can say that they are saying that spring technology is easy like spring season (where temperature is 5 of 0 degree).That is why in spring logo we will get one leaf symbol.

**How spring is light weight?**

As we know that AWT is heavy weight because it uses Operating system library to run i.e. it is dependent on Operating system libraries while swings is light weight as it is using only JDK libraries and it does not dependent on any Operating system libraries. Similarly EJB is heavy weight, becauseEJB is using Particular Application server libraries i.e. EJB is dependent on particular Application server while spring does not use any Application server library, and it runs on JDK platform that is why spring is light weight. Spring can be run by using the simple spring jars along with JDK.



**How spring is loosely coupled:**

One of the best example of loosely coupled is Mobile network, we can switch to any network without changing Number, sim card and mobile i.e. now we have the compatibility to switch from one network to another ,so the same type of compatibility need to have in our application also ,our application should not have any dependency with other layers. So while building MVC based application, here presentation layer is different, it is not a java component, here we can use HTML, css, JavaScript or JSP for dynamic pages. Coming to java part we have two layers Controllers and Model. Model again is divided into several layers like Services, Business and DAO layers.

For DAO we can use JDBC, OR Hibernate or JPA (Java Persistence API). If we don’t want to use JPA we can move it with Ibatis. So we can say that many ORM tools we have for DAO. Now if Business is not compatible with JDBC then we can use Hibernate or JPA or any ORM tool. So that compatibility in our business class should have so that in future we can use any of the things according to business needs. If this type of compatibility we have then we can say that these Business and DAO layers are loosely coupled. Now the question arises, how can we make these two layers compatible, because if create an object of DAO class into business class then it will not be compatible anymore because via DAO class we have to create connectivity with ORM tools. Answer is: By using Run Time Polymorphism (RTP) we can achieve the compatibility between these two classes. To make these two classes compatible we will use Interface and we will pass that interface reference inside the service class according to our requirements i.e. if requires JDBC then will send JDBC reference Object, if requires Hibernate we will send hibernate reference object etc. So Business layer can be compatible with any DAO part (JDBC, Hibernate, Ibatis or any other ORM tool). Now we can say that these two layers are loosely coupled

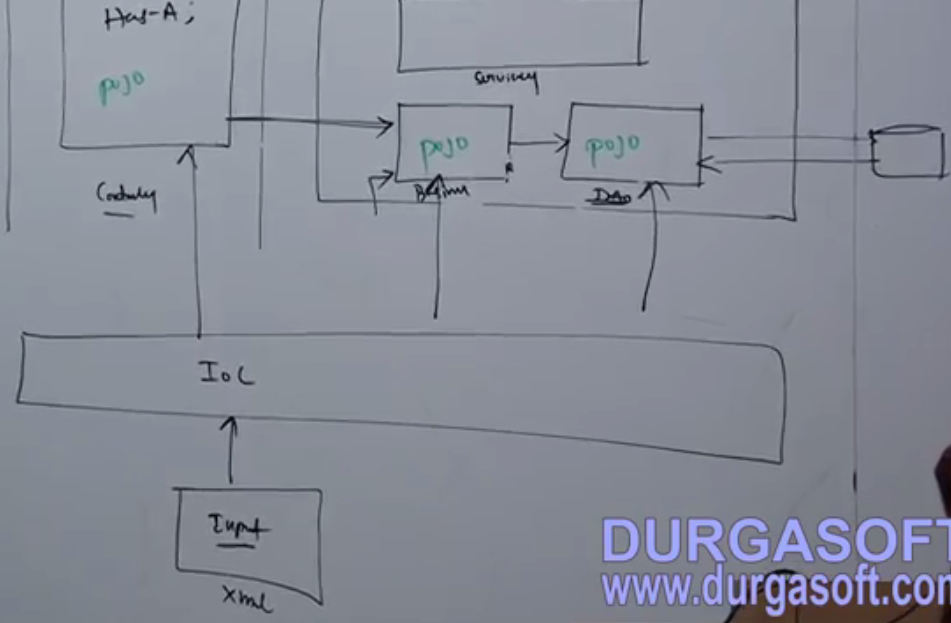
Now come to Controller and Business part: While controller is talking with business. Suppose in the Business class for some business can use Web Logic Global Transaction (TX) and Web Logic securities. Now suppose in future if we don’t want to use web logic TX and securities and we want to use JBoss TX and Securities OR Spring TX and security. Here also we can make it loosely couple with Interface reference object. We will create and interface for business and will pass that business reference object to controller according to our requirements i.e. by using RTP (Run Time Polymorphism) we can make these two layers loosely coupled. So loosely coupling is possible just because of RTP i.e. by taking Interface model we can make these layers loosely coupled.

**Two main principal used in spring implementation**

Note: So for spring implementation they are using mainly two principal

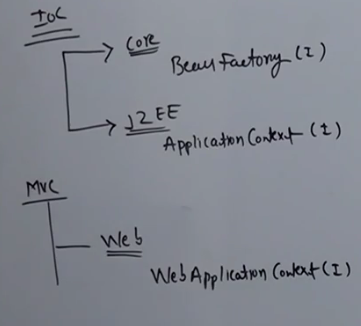
1. Run Time Polymorphism
2. Association (HAS-A) relationship

* So developer has to follow Association (HAS-A) relationship during the development and by using RTP we can pass our argument to our POJO class.
* But to manage these POJOs or to map this run time argument with the POJOs we need a container. In case of command prompt (P.S.V. main (String args []) we can easily pass the run time argument.
* In case of spring Instead of passing run time argument through command prompt by using XML file we can pass. So through XML file if you want to supply any input to this POJO class we need to have container support.
* Spring provides one container (IOC) which can read document (XML file) and by reading xml file it can pass the input to the POJOs classes.



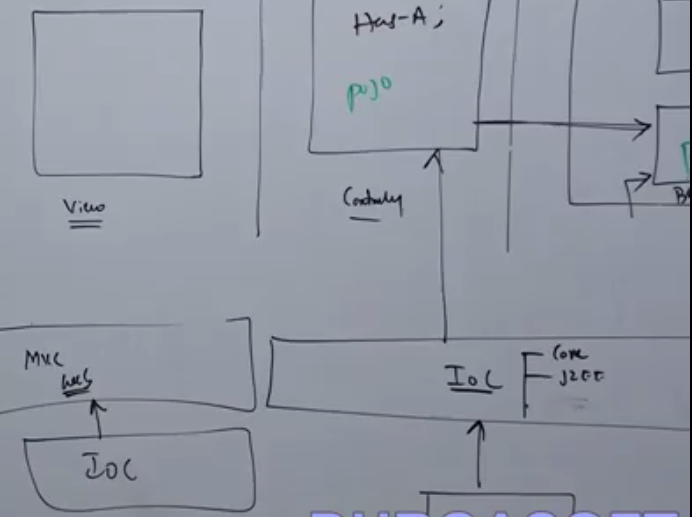
**Ques: What is IOC?**

It is an Inversion of controller it provide three containers (1- Core Container 2- J2EE Container (Advance Container) 3- Web Container



1. Core Container : It contains BeanFactory Interface
2. J2EE Container: Contains ApplicationContext Interface
3. Web Container: Contains WebApplicationContext Interface

**Note**: We don’t have a Web Container in IOC. Actually throughout spring we have three containers (Core, J2EE and Web) but Web Container is not there in IOC. Web Container is there in Spring MVC. And this web container is prepared on top of IOC only.



So in case of MVC application we don’t need to use IOC because by using IOC they have implemented web container. So by using Web Container we can manage MVC classes, we don’t need to use Core and J2EE container.

In Short the containers main jobs are:

1. **Read the data from XML** and pass the data to the POJO classes to make the classes loosely coupled. And this becomes possible because of RTP.

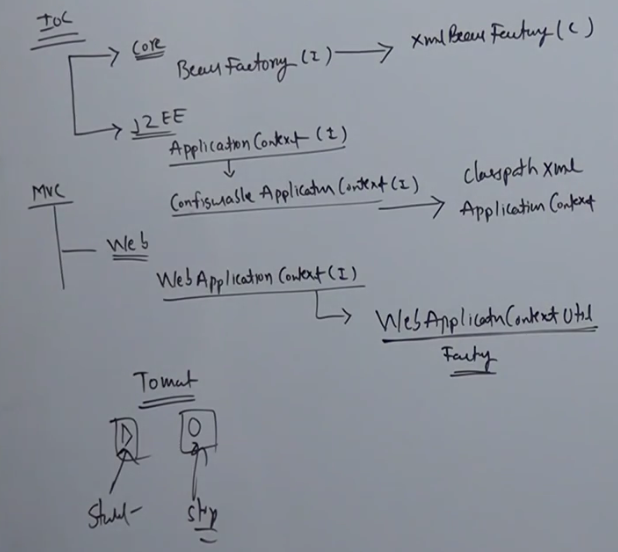
**Servlet Container/ Tomcat container**: Whenever we try to deploy our application first it will **read web.xml** file (In case Tomcat container) means the container having the capability to read XML file. After reading web.xml file it will search / find the “Load on Startup Servlet”. If it finds any “Load on Startup Servlet” then “Load on Startup Servlet” object immediately it will create. Means container read the xml file, understands the configuration .After understands the configuration it will **create the object** it will **call life cycle init () method**. So it means container **manages the life cycle** also and as well as finally it will **call destroy () method**

Before creating object the Servlet container will do something like it will create Servlet Context Object. From web.xml file it will read init parameter and context parameter. By reading init parameter and context parameter it will create Servlet Config object and Server Context Object. Means if we have any input in web.xml file then it can pass these input to the Servlet Class

**Servlet Container Vs IOC Container**

|  |  |
| --- | --- |
| In Short: **Servlet Container does:** | In Short: **IOC Container does** |
| 1. **Read the web.xml** 2. **Create the instances of XML Beans( Servlet)** 3. **It will manage the life cycle of Beans(Servlet)** 4. **Supply the Dynamic parameter the Beans ( Servlet)**   **Note**: Tomcat Container is nothing but the Servlet container that internally uses the Servlet.  **IOC Container:** It also does the same thing what the Tomcat/ Servlet container with little difference as given below | 1. Read the web.xml 2. Create the instances of XML Beans( POJO Classes) 3. It will manage the life cycle of Beans(POJO Classes) 4. Supply the Dynamic parameter the Beans (POJO Classes):- In Spring this is nothing but Dependency Injection (DI). This is the great feature that spring provides .This is the main asset of IOC container (DI means just passing XML data to the POJO classes) . Because of DI two layers becomes loosely coupled. Note is DI has been implemented by using RTP. |

**Note: How to start and stop the container:**

****

In case of Tomcat container we have start up and end button as shown in the image. By using some icons we can start /stop from system window. Now the question is how to start / Stop Spring IOC container. So as we can see in the image IOC are nothing but the simple interfaces, so how to start the interfaces. To start theses interface we need to have some implementation Class. So….

1. **For Core Container BeanFactory Interface** we have: - XmlBeanFactory (c) is the implementation class. This class will creates an object of BeanFactory **Interface**
2. **For J2EE Container Application Interface** we have: - For this interface we will have one more child interface (**ConfigurableApplicationContext Interface**) as shown in the image. This is one more J2EE container. For these two interfaces we have one implementation class (ClasspathXmlApplicationContext class)
3. **For Web Container WebApplicationContext Interface** we have: Here we have one factory class (WebApplicationContextUtil). It is a factory class like DriverManager class in JDBC. This factory class will create **WebApplicationContextUtil** object

**------------------------------------------------------------------------------------------------------------------------------------**

**HAS- A Relationship (Associations) and Runtime Polymorphism:**

Spring has been implemented by using two principal.

1. HAS – A Relationship (Association)
2. RTP – Runtime Polymorphism

One question arises here that while making complete MVC based application, so for preparing controller part or for preparing business part or DAO part which type of class need to extends or implement.

In case of servlet we need to extend abstract Generic Servlet class or abstract HTTP servlet class or implement servlet Interface or Generic Servlet class or HTTP servlet class to get features of servlet. In case of EJB we need to implement our classes from EJB API like Remote Classes, some EJB Interfaces, and EJB home. In sort we can say that while using API then Interfaces or Classes must be implement or extend.

For example: if we want to create one Hello servlet class then the fallowing code we will write.

**Class Hello** either **-implements Servlet** or **extends GenericServlet** or **extends HttpServlet**

But in case of spring we don’t need to extend any class or implement any Interface here, just simple pojo (plane old java Object)class (Bean Class) we can use to prepare Business part or DAO part i.e. simple POJO classes are enough to prepare controller part or business part or DAO part.

**Note:** All the required features we can pass through IOC container i.e. if pojo class requires any features then we pass these requirement from IOC container. As we know that to implement any Servlet features we need to extend class or implement the interface but in case of spring we don’t need to extend any class or implement any interfaces but if we requires or want some interface or class features in one particular class in spring then it can be achieve HAS-A (Association) relationship.

|  |  |
| --- | --- |
| Interface A {}  Class B {}  Class C {}  Class D {}  Class D {  B b ;  C c ;  A a ;  } | Suppose in spring, class D wants the features of Interface A, class B, and Class C then we don’t need to implement or extend these classes in D class. For getting the behavior we will associate these class references in D class i.e. instead of extending them, just associate their references in D class as shown. By associating these classes we can get all the behavior of these classes into D class. Spring recommends us to use association (HAS – A relationship) instead of using inheritance. By using association our class becomes loosely coupled, while in case of servlet class without extending we cannot take the features of servlet class or interface i.e. servlet is not loosely coupled. So in spring suppose an interface having several classes then these classes can be implemented in class D according to our requirement and in future if don’t need then we can implement another class from that interface. In this way our D class does not have any dependency on other classes. In this way in spring to make our application loosely coupled we are using association and in this way the class D which associates other class also a simple pojo class because we are not extending or implanting any class here. |

* So in spring to develop controller class or business class or DAO class we are using simple POJO class.
* As we know that spring application is developed using two principal Runtime Polymorphism (RTP) and HAS-A relationship. Developer has to use HAS- A relationship in his pojo classes and by using RTP we can pass any run time Object to the pojo classes.
* But to manage these pojo classes and to pass run time arguments we use xml files. Through the xml file we can supply any inputs to the pojo class and for this activity we need to have container (IOC) support. This IOC container reads the xml files and passes the argument to the pojo classes. Here IOC is an Inversion of controller and this inversion of controlling is possible just because of container. Here in spring IOC we have two containers.

1. **Core Container**
2. **J2EE Container**
3. **Web Container( Used in case of MVC application and build on top of IOC container)**

**Note**: Throughout spring we have three containers third is web container but this container is not present in IOC. The web container is present in Spring MVC. But the web container is prepared on top of IOC only. So in Spring MVC application we don’t need to use IOC container (Core and J2EE). By using web container we can mange Spring MVC classes.

**Types of Spring Containers:**

Throughout spring we have three containers third is web container but this container is not present in IOC. The web container is present in Spring MVC. But the web container is prepared on top of IOC only. So in Spring MVC application we don’t need to use IOC container (Core and J2EE). By using web container we can manger Spring MVC classes.

1. Core Container: Core container consist of one Interface : **BeanFactory**
2. J2EE Container: J2EE container consist of one interface: **ApplicationContext** & **ConfigurableApplicationContext**
3. Web Container: Web Container consists of one interface: **WebApplicationContext**.

**Q: What is the main job the container then?**

**A:** Containers main job is to read data from the xml file and pass the data to pojo classes into associated references to make our layer loosely coupled and this is possible just because of RTP. So by using RTP and HAS-A relationship we make our application layer light weight and loosely coupled.

Example: When we deploy and servlet application on tomcat server, here we have one tomcat container which does the following activities.

1. It will read one xml file web.xml present in the servlet application
2. After reading it will find load-on-startup servlet and if load-on-startup servlet is present then immediately one object of load-on-startup server will be created.
3. After creating object it will call servlet life cycle method (init method) .i.e. container will manage the life cycle of servlet also

**Tomcat Servlet Container Vs Spring IOC Container**

|  |  |
| --- | --- |
| Tomcat Servlet Container | Spring IOC Container |
| 1. Read xml files 2. Create instances of xml beans (Servlet) 3. It will manage the life cycle of beans (Servlet) 4. Pass dynamic parameter to the beans (Servlet) using init parameter of context parameter. | 1-Read xml files. Internally IOC uses SAX parser to check well-formness (Open and End tag) and validate (defined tag only) the xml file.  2-Create instances of xml beans (Pojo Classes)  3-It will manage the life cycle of beans (Pojo Classes)  4-Pass dynamic parameter to the beans (Pojo Classes). This is nothing but dependency injection (DI) concept. In IOC we can pass dynamic object from xml file to pojo classes. This (DI) is the most important feature or asset given by spring IOC container. Because of this DI two layers are loosely coupled and this DI is implemented using RTP. |

**How to start and stop spring container:**

As in Tomcat server we have two buttons to start and stop in the tomcat Servlet container. Using this start and stop button we can start and stop. But in spring we don’t have such type of button to start and stop. But in spring as we know that spring container contains interfaces and this interface contains some implementation classes. By creating object of these implementation classes we can start the spring container and this start up activity or code is written in init method of servlet.

1. **Core Container:** Core container consist of one Interface : **BeanFactory(I) :** contains implementation **class XmlBeanFactory( c)**
2. **J2EE Container:** J2EE container consist of one interface : **ApplicationContext(I) and ConfigurableApplicationContext(I):** contains implementation class**: ClassPathXmlApplicationContext(c)**
3. **Web Container:** Web Container consists of one interface: **WebApplicationContext** (I): This contains one factory class: WebApplicationContextUtil. This factory class will create an object of WebApplicationContext not for WebApplicationContextUtil. This factory class will internally use implementation class to create object of **WebApplicationContext class.**

**Note:** We will create one driver class which will use to drive an application in this calls we will use container class object to create main pojo class object which will use further to call another method of the pojo class.

**Example of simple HelloWorld Using core container:** Steps given below:

The main component to start container:

1. We should have a pojo class.
2. We should have one xml file.
3. We should have one driver class.

Note: XML file must have DTD (Document Type Definition) OR XSD (Xml Schema Definition). In case of Servlet application inside web.xml we don’t need to place DTD or XSD. Without DTD or XSD the web.xml will be validating as well as it will read by servlet container. In case of struts also we don’t need to have DTD or XSD, but while using hibernate in case of spring DTD or XSD must be place inside the xml file because with this it will not validate or read the xml by the spring container.

|  |  |
| --- | --- |
| **Class Test {**  **Public void hello() {**  **System.out.println(“HelloWorld”);**  **}**  **}** | **Class Test {**  **Public Test() {**  **System.out.println(“Default Construct”);**  **}**  **Public void hello() {**  **System.out.println(“HelloWorld”);**  **}**  **}** |
| **Spring.xml**  **DTD or XSD**  **<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN//EN”**  [**http://www.springframework.org/dtd/spring-beans.dtd**](http://www.springframework.org/dtd/spring-beans.dtd)**>**  **<beans>**  **<bean class = “Test” id = “t”/>**  **</beans>** | 1. **XML file contain one root tag <beans></beans>.** 2. **Inside this root tag we will have several child tag <bean></bean> to configure pojo class according to our need. This tag will have attribute class and id. This id will be used in driver class to create an object of the pojo class.** 3. **DTD or XSD lines we will get from one jar (spring-beans.jar). Go to one package (org.springframework.core.factory.xml). Here we will have one file spring-beans-2.0.dtd. From this file we need to copy paste the lines 37 and 38** |
| **Class client{**  **P s v main (String args[]){**  **// Find xml file**  **Resource res = new ClassPathResource(“Spring.xml”);**  // Load xml file into container  **BeanFactory factory = new XMlBeanFactory(res);**  // Object o = factory.getBean(“t”);  // Test test= (Test)o  **Test test= (Test)factory.getBean(“t”);**  test.hello();  **}**  **}** | **In this driver class:**   1. **Resources is used to load xml file into class path** 2. **Then object of resource res will be passed to XmlBeanFactory to start the container.** 3. **Using factory object and getBean of XmlBeanFactory class we will create the object of class test and here we will pass the id (t) defined in the xml file.** |

**How to create Project: Which jar files are required: How to add jar file into class path:**

**Spring bean scope Or Difference between Singleton Vs Prototype**

While defining the bean for pojo class, under the <bean> tag we have one attribute “scope” which is used to decide which type of bean instance should be return from spring container back to the caller.

5 types of bean scopes supported:

1. singleton – Return a single bean instance per Spring IoC container
2. prototype – Return a new bean instance each time when requested
3. request – Return a single bean instance per HTTP request. \*
4. session – Return a single bean instance per HTTP session. \*
5. globalSession – Return a single bean instance per global HTTP session. \*

In most cases, you may only deal with the spring’s core scope – singleton and prototype, and the default scope is singleton.

P.S \* means only valid in the context of a web-aware Spring ApplicationContext

**Question + Note: How many object** factory.getBean (“t”); will create?

**Answer:** It will create test one test class object. To check let us put one default constructor inside the test class for debugging purpose. As we know that usually constructor get called while creating the object of the class. So wherever we create one – one individual object then constructor should get called. So wherever I try to call (factory.getBean (“t”) ;) getBean () method by using IOC container class object (factory), it will create one test class object. In case if I create object multiple times as shown below.

|  |
| --- |
| Object o = factory.getBean (“t”);  Object o1 = factory.getBean (“t”);  Object o2 = factory.getBean (“t”);  Object o3 = factory.getBean (“t”); |

Now if we run the above program with these objects, we will see that constructor will be called only one time.

**Reason:** In case of Servlet application if we create multiple objects then in servlet class it will create multiple object but tomcat container make it singleton. In the same way when we create multiple objects inside the client class then multiple objects will be created but spring container make this object singleton so all the different object will be treated as singleton and the constructor inside the test class will be called once.

Inside the xml file we have one attribute scope which value by default is singleton. But if we change this value with Prototype then the entire above created object will treated different object for spring container and the constructor will be called as many times as many object we create.

|  |  |  |
| --- | --- | --- |
| **Class Test {**  **Public Test() {**  **System.out.println(“Default Construct”);**  **}**  **Public void hello() {**  **System.out.println(“HelloWorld”);**  **}**  **}** | **<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN//EN”**  [**http://www.springframework.org/dtd/spring-beans.dtd**](http://www.springframework.org/dtd/spring-beans.dtd)**>**  **<beans>**  **<bean class = “Test” id = “t” scope=”singleton” />**  **OR**  **<bean class = “Test” id = “t” singleton = “false”/>**  **</beans>**  **Note: For adding scope attribute we need to use:**  **<!DOCTYPE beans PUBLIC “-//SPRING//DTD** BEAN 2.0**//EN”** | **Class client{**  **P s v main (String args[]){**  **// Find xml file**  Resource res = new ClassPathResource(“Spring.xml”);  // Load xml file into container  BeanFactory factory = new XMlBeanFactory(res);  // Object o = factory.getBean(“t”);  Object o1 = factory.getBean (“t”);  Object o2 = factory.getBean (“t”);  Object o3= factory.getBean (“t”);  // Test test= (Test)o  Test test= (Test)factory.getBean(“t”);  test.hello();  **}**  **}** |

**Output of the above program:**

Default Constructor

HelloWorld

Now if run the same program with scope=”prototype”

|  |  |  |
| --- | --- | --- |
| **Class Test {**  **Public Test() {**  **System.out.println(“Default Construct”);**  **}**  **Public void hello() {**  **System.out.println(“HelloWorld”);**  **}**  **}**  **Output:**  Default Construct  Default Construct  Default Construct  Default Construct  HelloWorld | **<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN//EN”**  [**http://www.springframework.org/dtd/spring-beans.dtd**](http://www.springframework.org/dtd/spring-beans.dtd)**>**  **<beans>**  **<bean class = “Test” id = “t” scope=”Prototype” />**  **</beans>** | **Class client{**  **P s v main (String args[]){**  **// Find xml file**  Resource res = new ClassPathResource(“Spring.xml”);  // Load xml file into container  BeanFactory factory = new XMlBeanFactory(res);  // Object o = factory.getBean(“t”);  Object o1 = factory.getBean (“t”);  Object o2 = factory.getBean (“t”);  Object o3= factory.getBean (“t”);  // Test test= (Test)o  Test test= (Test)factory.getBean(“t”);  test.hello();  **}**  **}** |

**Singleton** **Prototype**

|  |  |
| --- | --- |
|  |  |

**Note**: If we compare every object then it will return true only because the same reference of the first object will be return to every newly created object because the created object scope is singleton but if we change the scope prototype then in this case for every request we will have different object with its own reference.

So internally IOC container after reading xml file, whenever we try to call getBean (“t”) method it will try to search for the references (t) in xml file and t reference class(Test) object will create based on the scope defined in the xml file. If scope is singleton then only one reference will create and the same reference will be assigned to other created object (if created). But if scope is prototype then it will create new object for every request.

**Apart from these two scopes (Singleton and Prototype) we have three more scopes (request, session and context) but these three scopes are valid for web application, for standalone application we can use singleton and prototype.**

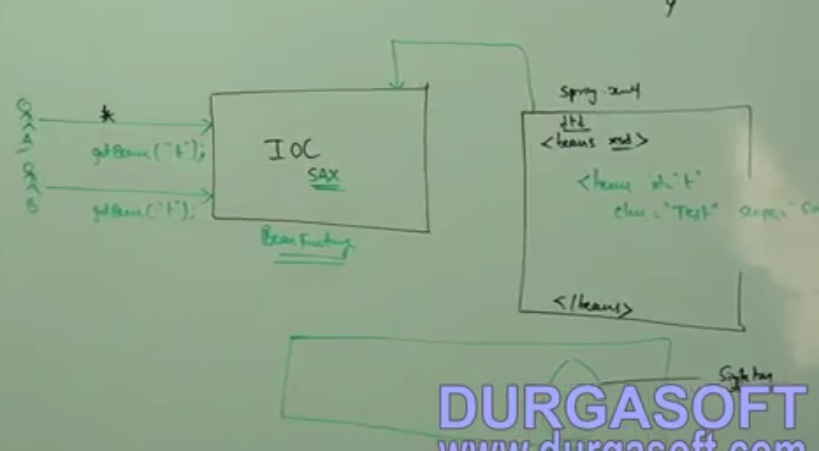
**Example of simple HelloWorld Using J2EE or Advanced container:**

|  |  |  |
| --- | --- | --- |
| **Class Test {**  **Public Test() {**  **System.out.println(“Default Construct”);**  **}**  **Public void hello() {**  **System.out.println(“HelloWorld”);**  **}**  **}** | **<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN//EN”**  [**http://www.springframework.org/dtd/spring-beans.dtd**](http://www.springframework.org/dtd/spring-beans.dtd)**>**  **<beans>**  **<bean class = “Test” id = “t” scope=”Prototype” />**  **</beans>** | **Class client{**  **P s v main (String args[]){**  **// Find xml file**  //Resource res = new //ClassPathResource(“Spring.xml”);  // Load xml file into container  ApplicationContext factory = new ClassPathXmlApplicationContext(“Spring.xml”);  // Object o = factory.getBean(“t”);  Object o1 = factory.getBean (“t”);  Object o2 = factory.getBean (“t”);  Object o3= factory.getBean (“t”);  // Test test= (Test)o  Test test= (Test)factory.getBean(“t”);  test.hello();  **}**  **}** |

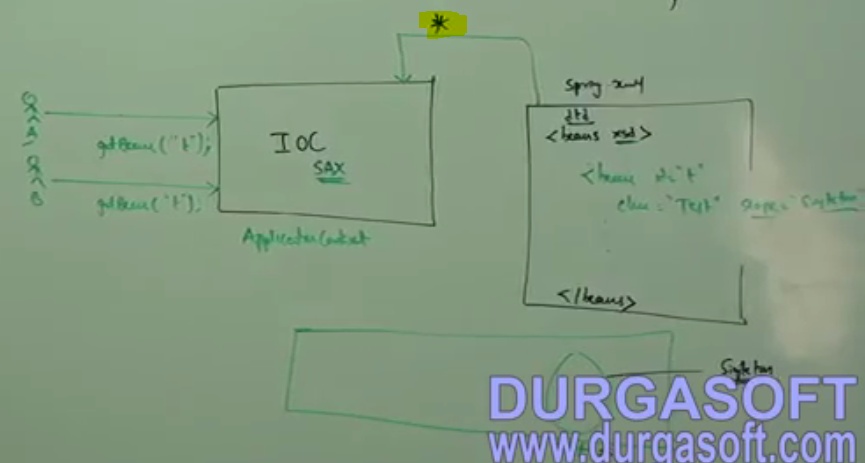
**Difference between Core Container (BeanFactory) and J2EE or Advanced Container (ApplicationContext**):

|  |  |
| --- | --- |
| **Core Container – BeanFactory** | **J2EE – Advanced Container – ApplicationContext** |
| 1. Here the object is gets created while we use factory.getBean (“t”); in other word on demand. 2. Here to load the xml file we have to use ClassPathResource(“Spring.xml) 3. This is called lazy container. 4. As here the object get created on-demand so suppose if we have 1000 object then every time it will take time to create object for every user and user has to wait. 5. It is just similar to without load-on-start up servlet | 1. Here object gets created while loading the xml file in the spring container. 2. Here we don’t need to use. 3. This is called early container or Eger container. 4. This early loading concept / approach is very useful for performance purpose because all the object get loaded while loading xml file and is available for user to use. 5. It is just similar to with load-on-startup servlet**.** |

* **Note**: AS we know that container read and validates the xml file. To validate the xml file **container internally used SAX parser.** This parser check whether the xml file is in well-formed (i.e. for every open tag there must be one end tag) as well as validate (i.e. xml file contains only defined tag, if it finds any undefined tag, it throws parser error.)
* So while loading the xml files in the **core container or J2EE container**, first of all the SAX parser which reside inside the IOC container will check the well formedness and validness of the xml file and if finds any problem, it throws parser exception.
* Suppose if the IOC container is core container then object will be created in the JVM on-demand (when the first user request comes using getBean (“t”) method) after loading the xml file. In case if another request comes using the same method getBean (“t”), then the same reference of the first object will be assigned to another request also because by default the scope is singleton . But if we change the scope from Singleton to Prototype then in this case every created object will have its own reference**.**



* Suppose if the IOC container is J2EE container then it will create object of all the bean declared in the xml file during the loading of the xml file but if the bean scope is singleton only. But if we change the scope from singleton to prototype then it will not create any object at the time of loading xml file because in prototype for each request it will create object with its own reference. So in J2EE container the object will be created in the jvm on demand after loading xml file if and only if the bean scope is prototype i.e. in this case J2EE container behaves like core container (lazy Container)



**Q: How the IOC creates object of the class defined in the <bean> tag?**

**A:** IOC container internally uses reflection concept i.e. Class.forName (“Test”) new instance (); this creates object only when the class or constructor is public. But the spring creates the object for private constructor also because spring internally converts the private constructor to public constructor then creates the object.

**Example: How to access private constructor using reflection.**

|  |  |
| --- | --- |
| **Package** **privat.constructor.beans;**  **public** **class** Test {    **private** Test(){  System.*out*.println ("Private Access Constructor .....called");  }  }  **Generally we cannot access private constructor class outside of the class but using reflection (Class.forName) we can access.** | **package** privat.constructor.client;  **import** java.lang.reflect.Constructor;  **public** **class** Client {  **public** **static** **void** main (String args[]){  **try** {  // Test t = new Test (); // Throw visibility error as the constructor is private.  Class c = Class.*forName*("privat.constructor.beans.Test");  Constructor con[] = c.getDeclaredConstructors();  // will get constructor of array. So as here only one constructor is here so will get one constructor.  con[0].setAccessible(**true**);//Accessing the constructor by making it public. Here setAccessible(true) making the constructor as public  con[0].newInstance(**null**); // Creating the instance of the class Test .    } **catch** (Exception e) {  // **TODO**: handle exception  }  }  } |
| If in the above Client class we try to call the private constructor by creating Test class object then it will throw the visibility error given in other box. |  |

**So we can conclud that private constructor can be accessed by using reflection class not by direct. So by using the same code (reflection class) our both IOC container (Core and J2EE) also creates the object of bean classes. So in spring if we have private constructor class , it will create an object for that private constructor classes.**

**Dipendency Injection:**

As we know that spring container suppoerts for POJO class instanciation , life cycle management and the most important feature supported by spring container is DIPENDENCY INJECTION. This is the most important assest given by spring. Now a question arises.

**Q : Dependency means what . A class could have what kind of dependencies ?**

Lets say if we have one DatabaseAccess(DAO) class then to access database our DatabaseAccess (DAO) class should have dependecny on connections (driver,Url, username and password). These are dependencies for DAO class to access DATABASE. These dependencies we can pass or inject to the DAO class through xml file.

1. **DriverClass**
2. **Url Name**
3. **Username 4- Password**

DB

In the same way while implementing complete MVC based applications then we have to implements three layes (Presentation Layer , Controller Layer , Model Layer(Bussiness and DAO layes)).

Model

Services (TX, JMS, Mail)

DB

Presentation Layer

Controller Pojo

DAO

Business

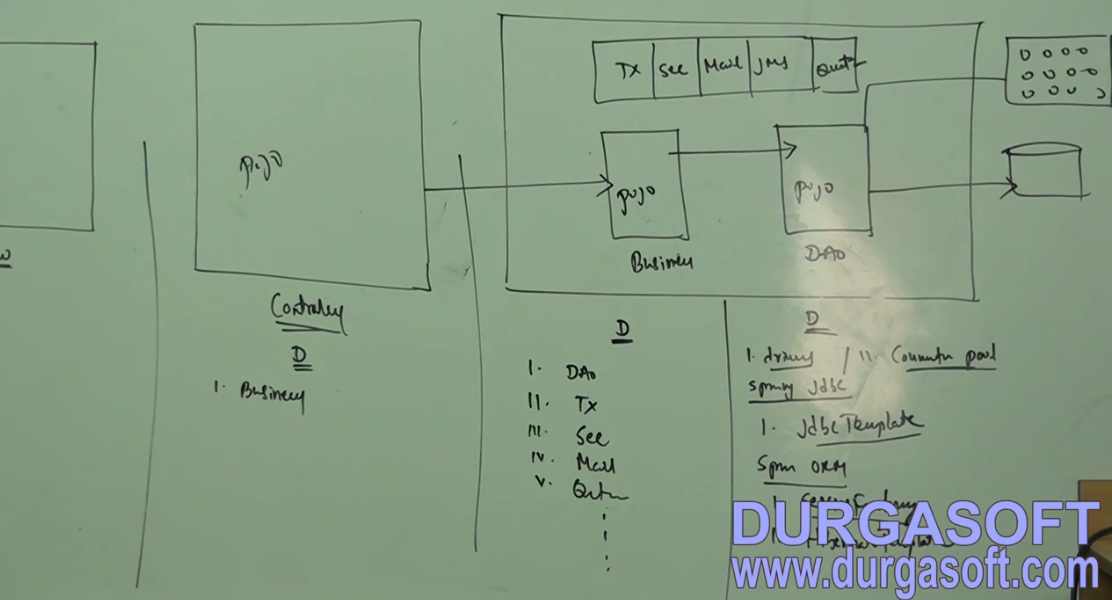
**IOC Container (Reading the data from xml file and supplying data to the pojo or bean classes)**

Spring.xml

**Spring.xml file**

|  |  |  |
| --- | --- | --- |
| **Here In case of controller has to talk with Model (Business). So by taking business dependent as a parameter controller can talk with business**  **Dependencies for DAO:**   1. **Business classes** | **Here in the Model Layer we can have Business part. So by taking DAO dependent as a parameter business can talk with DAO.**  **Here in the business class we don’t need to extend or implement any API simple pojo class is enough but to provide securities or transaction business class should have some dependencies on services and two talk with DB it will have dependency on DAO class.**  **Dependencies for DAO:**   1. **DAO** 2. **Transactions service** 3. **Securities service** 4. **Mail service and so on.** | **Here in the Model layer we can have Business and DAO**  **Dependencies for DAO:**   1. **Driver, URL, username, password if we have normal java application.** 2. **OR JDBC template if we are using spring jdbc.** 3. **OR Session factory if we are using Hibernate** 4. **Or Hibernate Template** |

**So we can say that our controller class, business class and DAO class could have these types of dependencies as defined above and these all dependencies can be passed or supply to the POJO classes through spring xml file and these inputs or dependencies can be supplied to the pojo class with the help of IOC. IOC will read the spring xml file data and the same data it will supply to all the MVC POJO classes.**

****

**Types of Dependency Injection** **So final definition of Dependency Injection:**

Passing or supply the required inputs or parameter to the pojo classes through spring xml file from spring IOC container is nothing but Dependency Injection (DI). We can pass these parameters at run time also. It makes our code loosely coupled and easier for testing.

For doing this DI i.e. for passing these data to the pojo classes, the pojo classes must have either setter method or parameterized constructor I.e. the pojo classes to take input from IOC container must have either setter method or parameterized constructer.

**So based on the above definition we have two types of dependency Injection.**

1. [**Setter-based dependency injection**](https://www.tutorialspoint.com/spring/setter_based_dependency_injection.htm)
2. [**Constructor-based dependency injection**](https://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)

**Note:** In some books they have defined one more type of DI: Interface based dependency injection which not possible. Actually internally it uses setter-based dependency injection.

|  |
| --- |
| **package** bean.dependency.injection;  **public** **class** Test {  **public** **void** hello(){  System.*out*.println("Dependency Injection is completed");  }  } |
| **package** bean.dependency.injection.client;  **import** org.springframework.beans.factory.BeanFactory;  **import** org.springframework.beans.factory.xml.~~XmlBeanFactory~~;  **import** org.springframework.core.io.ClassPathResource;  **import** org.springframework.core.io.Resource;  **import** bean.dependency.injection.Test;  **public** **class** Client {  **public** **static** **void** main(String args[]){  Resource res = **new** ClassPathResource("resources/spring.xml");//Find Xml  BeanFactory factory = **new** XmlBeanFactory(res); //Load xml into container  System.*out*.println("xml file loaded");  //Create test class object  Object obj = factory.getBean("t");  Test t = (Test)obj;  t.hello();  }  } |
| **Spring.xml**  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.dependency.injection.Test"*></bean>  </beans> |

**Example of dependency Injection**

|  |  |
| --- | --- |
| **package** bean.di.injection;  **public** **class** Test {  **private** String gender;  **public** String getGender() {  **return** gender;  }  **public** **void** setGender(String gender) {  **this**.gender = gender;  }  **public** **void** hello(String name){  System.*out*.println("Hello."+gender+""+name);  System.*out*.println("Dependency Injection is completed");  }  } | **package bean.dependency.injection.client;**  **import org.springframework.beans.factory.BeanFactory;**  **import org.springframework.beans.factory.xml.~~XmlBeanFactory~~;**  **import org.springframework.core.io.ClassPathResource;**  **import org.springframework.core.io.Resource;**  **import bean.dependency.injection.Test;**  **public class Client {**    **@SuppressWarnings("deprecation")**  **public static void main(String args[]){**  **//Find Xml**    **Resource res = new ClassPathResource("resources/spring.xml");**  **System.*out*.println("xml file found");**  **//Load xml into container**  **BeanFactory factory = new XmlBeanFactory(res);**  **System.*out*.println("xml file loaded");**  **//Create test class object**  **Object obj = factory.getBean("t");**  **Test t = (Test)obj;**  **t.hello("arun");**  **}**  **}** |

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  <property name=*"gender"* value=*"Mr."*></property>  </bean>  </beans> | OR  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  <property name=*"gender"*>  **<value>Mr.</value>**  </property>  </bean>  </beans> |

OR

Here in the xml file <property> tag is used for setter based dependency injection. Inside this tag name attribute defines the pojo class variable and value is the run time parameter that we are passing to the setter method of pojo class. We can pass run time parameter using <value> parameter also as shown in the second xml so any type of data we can inject to the bean classes through the xml file. In other word we can inject primitive data, arrays, collections etc.

**Note**: We can use name attribute only once which is representing the variable name of pojo class. If use the same name attribute with the same variable name of pojo class with different value i.e. if we want to override then container will throw an exception (parsing exception). So in setter based DI attributes cannot override. And one more thing is that only one value can be injected in setter based DI, we cannot set two pojo class values. For example suppose if we have one more variable email along with gender variable and we want to inject both of the values using setter method then container will throw an error.

|  |  |
| --- | --- |
|  | Will throw duplicate error. We cannot pass the different value with the same property attribute (like here inside property tag, name is an attribute, so we cannot create another property with same attribute with different or same value. If we try to do then we will get exception. |

**Limitation of setter based DI:**

1. We can inject only one parameter or value through setter based DI.
2. We cannot override the attribute value.

[**Constructor-based dependency injection**](https://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)

If we want to pass the parameter through constructor then we should have parameterized constructor.

This constructor could be any access type (private and public) as in spring we can access the private constructor outside the class.

In case if we have only one singe parameterized constructor in that case compiler will not declare any default constructor. So if we have only one single parameterized constructor

**Case 1- If we have any default constructor inside the class then to call the default constructor we don’t need to declare any <constructor-arg> tag between <bean></bean> tag. But if it doesn’t find any default constructor inside the class then it will throw an exception that there is no any constructor.**

|  |
| --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.dependency.injection.Test"*></bean>  </beans> |

**Case 2- :**  **If class contain any parameterized constructor then it is compulsory to pass parameter values should in spring.xml file**.

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  **<constructor-arg value= “XYZ “>**  </bean>  </beans>  Here argument is being passed using value attribute.  OR-🡪 | **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  **<constructor-arg >**  **<value> XYZ</value>**  **</constructor-arg >**  </bean>  </beans>  Here the parameter is being passed using <value> child tag. |

**Note:** Like In setter based DI we have one **attribute** “name” inside the <property> tag. But here in parameterized constructor we don’t have any such attribute; we just need to pass **value** using **value attribute**. Here we just need to fallow the same sequence of argument/parameter what the parameterized constructor have inside the class. So if we are passing only one value then it will call constructor having one single parameter only.

**Case3:-** If we have one more constructor i.e. constructor overloading, means inside the class we have constructor with different argument type (int or double or float).

By default every argument passed through **constructor-arg** will be treated as **string** by IOC, so the question arises that how IOC will call the constructor having data type like (int, double, float) other than string.

* To solve this problem spring provide one attribute “type” inside <constructor-arg> tag as shown in the fallowing syntax.

|  |  |
| --- | --- |
|  | **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  **<constructor-arg value= “123 “>**  </bean>  </beans> |

**Note:**  As here we can see that in spring.xml , if we pass “123” inside <constructor-org> tag or 123 inside <value> tag, by default it will be treated as string (as here we have not define any type attribute . So only that constructor will be called which is having one single string parameter. Other construct (which is having int parameter) will not be called even here inside <value> attribute we are sending number 123. Actually IOC first will read spring.xml file as string and then it will search the constructor having string parameter and then inject the string value to the constructor.

**So the question arises that in case of overloaded constructor how to inject the value to related constructor.**

**So the answer is** spring provide one attribute “type” which is nothing but defines the data type of the parameterized constructor. Using this type attribute we can solve this constructor overloading problem.

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.injection.Test"*>  **<constructor-arg value= “123 “ type=”int”>**  </bean>  </beans> | As here we can see that we have defined one attribute [type=”int”] inside <constructor-arg> tag. Now it will inject the argument 123 which is having int parameter inside the constructor. So from the above screen it will call Test (int age) constructor. |

***Note***: In case of Setter DI, one setter method should not contain more than one argument but constructor can have more than one argument. So now the next case 4

**Case 4:**- If we have one more overloaded constructor with two input parameter [**Test (String name, int age) { }]**

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.Test"*>  **<constructor-arg value= “123 “ type=”int”>**  **<constructor-arg value= “xyz “ type=”String”>**  </bean>  </beans> |  |

So in the above screen shot we have declared two <constructor-arg> with its type , so now it will call the constructor which is having two parameter and based on type (defined in the spring.xml) it will inject the value to the constructor **Test (String name, int age) { }**

**Note: Suppose if we have one more parameter (email)**

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"t"* class=*"bean.di.Test"*>  **<constructor-arg value= “123 “ type=”int”>**  **<constructor-arg value= “Arun “ type=”String”>**  **<constructor-arg value= “xyz @gmail.com“**  **type=”String”>**  </bean>  </beans>  Now in this case we have one int type and two string type argument, so now IOC will be confused like to whom it has to assign name and to whom it has to assign email. Here we will get ambiguity. |  |

**Then question** is how to solve this ambiguity. By using type attribute we cannot solve. Because type will work for constructor having two different type argument but it fails when the constructor having similar type argument.

**Answer**: One possible answer is, if we fallow the same order what we have inside constructor argument then we will not get an ambiguity issue as shown in the above example.

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  **<bean id=*"t"* class=*"bean.di.Test"*>**  **<constructor-arg value= “Arun “>**  **<constructor-arg value= “123 “>**  **<constructor-arg value= “xyz @gmail.com“>**  **</bean>**  </beans> | Here in the second xml file we have fallowed the same order what order of parameter we have inside the parameterized constructor. So in this case IOC will inject parameter in the same order and will not get ambiguity here. In this case we don’t need to define any type attribute also.  **Question**: So now if we don’t want to fallow order here then?  **Answer :**  If we don’t want to fallow order then to avoid ambiguity we can use attribute “index” |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  **<bean id=*"t"* class=*"bean.di.Test"*>**  **<constructor-arg value= “123 “ index=”1”>**  **<constructor-arg value= “xyz @abc.com“ index=”2”>**  **<constructor-arg value= “Arun “ index=”0”>**  **</bean>**  </beans> | So here using index attribute we can avoid ambiguity.  Since index start from zero that why name index is 0, age index is 1 and email index is 2 are given. |

Note: In case setter based DI overriding is not possible but in CBD overriding is possible so if we write one more <constructor-arg value =”Bunty” index =”0”> then it will over write Bunty with Arun.

|  |  |
| --- | --- |
| **Spring.xml file:**  <! DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  **<bean id=*"t"* class=*"bean.di.Test"*>**  **<constructor-arg value= “123 “ index=”1”>**  **<constructor-arg value= “xyz @abc.com“ index=”2”>**  **<constructor-arg value= “Arun “ index=”0”>**  **<constructor-arg value= “Bunty “ index=”0”>**  **</bean>**  </beans> | Here Bunty will override with name “Arun”. So the latest value will be injected to the constructor. |

|  |  |
| --- | --- |
|  |  |

**Example: Need to test in eclipse.**

**So far we have learnt that how to inject Primitive data type like (String, int) and how to inject data in parameterized constructor. Now we will talk about Secondary data type like (Class reference, collection objects, Arrays). In real time, maximum we will have secondary data type.**

Explanation: Let’s say we have one class Car which has dependency on class Engine. We have engine association (reference of Engine Class act as variable- means secondary data type association) in class Car.

|  |  |  |
| --- | --- | --- |
| **Car.java**  **package** secondary.type.di.bean;  **public** **class** Car {  **private** Engine engine;  // Here engine we have secondary type engine association.  **private** String carName;  // carName is primitive type.  **public** Car(){  System.*out*.println("Inside the Car class");  } //Wire the setter method of engine and carName as given below  **public** **void** setEngine(Engine engine) {  **this**.engine = engine;  }  **public** **void** setCarName(String carName) {  **this**.carName = carName;  } //Finally print the data as given below.  **public** **void** PrintCarData(){    System.*out*.println("Car Name:"+carName);  System.*out*.println("Engine Model Year:"+engine.getModelYear());  }  }  **Note**: Now to inject engine and carName we will have one spring xml file as given below | | |
| **Car.xml**  **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"**  **"http://www.springframework.org/dtd/spring-beans-2.0.dtd">**  **<beans>**  **<bean id=*"c"* class=*"secondary.type.di.bean.Car"*>**  **<property name=*"carName"* value=*"Audi"*></property>**  **<property name=*"engine"* ref=*"en"*></property>**  **</bean>**  **</beans>**  **Note: Here for secondary data type we have written ref instead of value (Which is used for primitive data type). Here the ref=”en” , en is nothing but the id defined in Engine.xml so by passing this reference “en” here we can pass or inject the Engine Class value in engine setter method. Since we are passing the “en” defined in Engine.xml in Car.xml (for injecting Engine class value in engine setter method) so we need to load both the xml file (Engine.xml and Car.xml) at a time into one single IOC container.**  **We can write one single xml file instead of two (Engine.xml and Car.xml) but we need to remember that id should be unique throughout application.** | | |
| **Engine.java**  **package** secondary.type.di.bean;  **public** **class** Engine {  **private** String modelYear;  // modelYear is primitive type.  **public** Engine(){  System.*out*.println("Inside the Engine class");  }  **public** String getModelYear() {  **return** modelYear;  }  **public** **void** setModelYear(String modelYear) {  **this**.modelYear = modelYear;  }  } | | |
| **Engine.xml**  **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"**  **"http://www.springframework.org/dtd/spring-beans-2.0.dtd">**  **<beans>**  **<bean id=*"en"* class=*"secondary.type.di.bean.Engine"*>**  **<property name=*"modelYear"* value=*"2017"*></property>**  **</bean>**  **</beans>**  **Note: While loading both the xml (Engine.xml and Car.xml) simultaneously, then IOC first creates Object for Engine class and assigns the Model Year (2107) using setter method** setModelYear**. Next IOC will create the object of Car class and assign carName value “Audi” using setter method** setCarName **and will create an engine reference which be attached from Engine.xml file and this reference will be available in Car class as we are passing the same reference to Car.xml file also.** | | |
| **Client.java**  **package** secondary.type.di.bean;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Client {  **public** **static** **void** main(String[] args) {  // To add all the xml file in one container will create one string arrays  String[] iocFiles= **new** String[]{"resources/car.xml","resources/engine.xml"};  System.*out*.println("Lording......"+iocFiles.length);  ApplicationContext ac = **new** ClassPathXmlApplicationContext(iocFiles);  Car car=(Car)ac.getBean("c");  car.PrintCarData();  /\*Load this string iocFiles(having all the configuration file into container)  As soon as we load these file in ApplicationContext IOC container, it will create object of all the beans defined in the xml file\*/  -----------------------------Using Inner Bean- One single spring xml file---------------  ApplicationContext ac = **new** ClassPathXmlApplicationContext("resources/InnerBean.xml");  Car car=(Car)ac.getBean("c");  car.PrintCarData();  }  } | | |
| **Note**: Here we have passed Engine class reference as a reference. In secondary type DI we have two approaches; we can pass or inject the variable either **Pass by Reference** or **Pass by object** | | |
| Class A {  }  Main Class  Test t = new Test();  **t.Method(new A()); OR**:  A a = new A();  t.Method(a); | Class Test{  Method(A a )  } | As in the give example we can see that we one Class A and another Class Test. Class Test has one Method having argument (A a ) . Now in the main class we can call the method using test class object as we can see in the example. Now while call Method we can pass the A class object directly object using new operator (new A () ) OR:  Another approach is we can create a reference of A class and we can pass this reference inside Method like ( t.Method(a)) |

So incase if we want to pass directly an instance to the Car Class, then we will do using inner bean approach.

|  |
| --- |
| **Inner Bean approach**  **Car.xml**  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"c"* class=*"secondary.type.di.bean.Car"*>  <property name=*"carName"* value=*"Audi"*></property>  <!-- <property name="engine" ref="en"></property> -->  <property name=*"engine"*>  ***<bean class="secondary.type.di.bean.Engine">***  ***<property name="modelYear" value="2017"></property>***  ***</bean>***  </property>  </bean>  </beans> |
| **OR**  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"en"* class=*"* ***secondary.type.di.bean.Engine*** *"*>  ***<property name="modelYear" value="2017"></property>***  ***<bean class="*** *secondary.type.di.bean.Car****"*** ref="en"***>***  <property name=*"carName"* value=*"Audi"*></property>  ***</bean>***  </bean>  </beans> |

So here we can see that now we have only one xml file Car.xml for dependency Injection. Here we have used inner bean approach. In this approach (Pass by Object) we have directly passed the instance of Engine class to Car class. But this approach is not reusable. Suppose if any other class need to have Engine class then again we need to write same code (mentioned in bold italic). But using pass by reference approach can pass the reference to the multiple classes. So inner bean can be used if we don’t want to reuse.

So far we have learnt about:

1. IOC Container
2. Types of DI
3. Now we will learn about Dependent types.

**Different type of Dependent types**

As per last discussion a class can have any type of dependencies like

1. Primitive Types
2. Simple Secondary types
3. Array Primitive / Secondary Arrays
4. Collections

Now if we want to inject array primitive or secondary array.

As we know in simple primitive type dependency injection, like in setter type DI we use one property tag and inside this tag we use one attribute “value” to supply the value of parameter. As given below

***<property name="modelYear" value="2017"></property>***

But in case of array we will have multiple values to be injected so in this case so using above syntax only one value can be injected. So in case of array the “value” attribute will be written as child tag in between <list></list> and in between <property> tag.

|  |  |
| --- | --- |
| Class Test{  Private String ***modelYear***;  // setter of ***modelYear***  } | **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"**  **"http://www.springframework.org/dtd/spring-beans-2.0.dtd">**  **<beans>**  <bean id=*"en"* class=*"****Test****"*>  ***<property name="modelYear" value="2017"/>***  </bean>  </beans> |
| **Array Primitive DI**  Class Test{  Private String ***modelYear[]***;  // setter of ***modelYear***  } | **<bean id=*"en"* class=*"Test"*>**  ***<property name="modelYear">***  ***<list>***  ***<value>2017</value>***  ***<value>2018</value>***  ***<value>2019</value>***  ***</list>***  ***</property>***  **</bean>** |

So in case of primitive array the multiple value can be passed between <list> </list> tag.

**Secondary Array DI**

Suppose we have one car class and one Engine class and we have to inject Engine class into Car class then the fallowing example will work.

|  |  |
| --- | --- |
| Class Car{  Private String ***modelYear***;  // setter of ***modelYear***  } | Class Engine{  Private String ***engine***;  // setter of ***engine***  } |
| **Here in this case we have only one engine to be injected**  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"en"* class=*"****Engine*** *"*>  ***<property name="modelYear" value="2017"></property>***  </bean>  ***<bean class="****Car****"*** ref="en"***>***  <property name=*"carName"* value=*"Audi"*></property>  ***</bean>***  </beans> | |

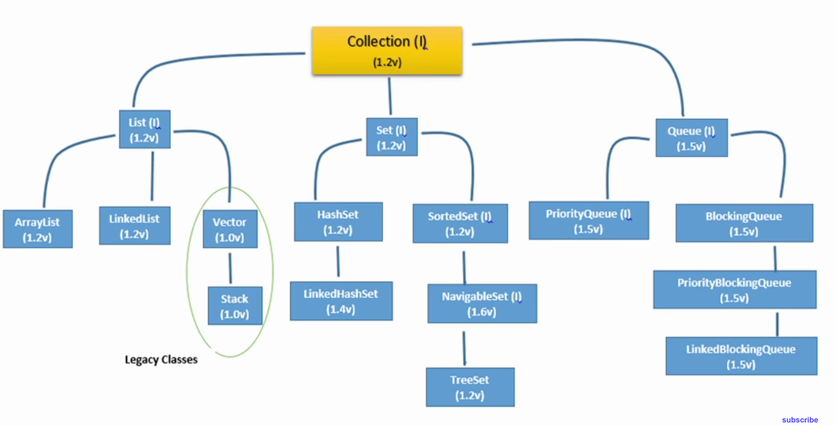
Now suppose if we have multiple engines to be injected into Car class then using ref =”en” it is not possible .The below example will be written for multiple engine to be injected.

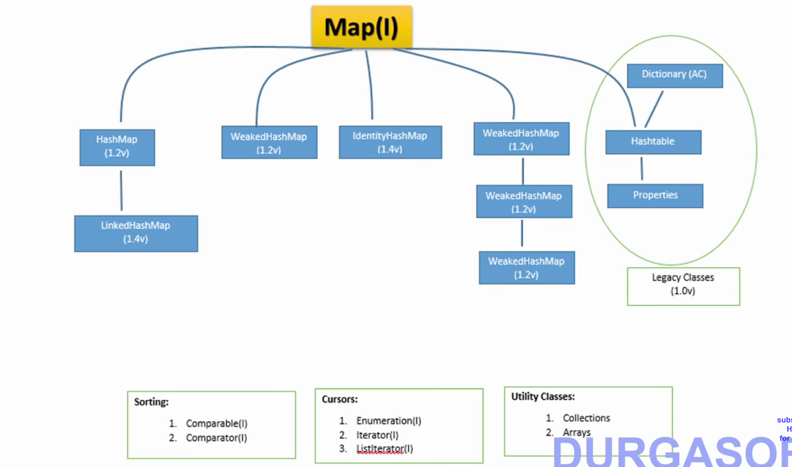
|  |  |
| --- | --- |
| Class Car{  Private String ***modelYear***;  // setter of ***modelYear***  } | Class Engine{  Private String ***engine[]***;  // setter of ***engine***  } |
| <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  <bean id=*"en"* class=*"****Engine*** *"*>  ***<property name="modelYear" value="2017"></property>***  </bean>  ***<bean class="****Car****">***  ***<list>***  ***<ref beans=”en”>***  ***<ref beans=”en”>***  ***<ref beans=”en”>***  ***</list>***  ***</bean>***  </beans> | |

* So in both the cases whether it has primitive type array or secondary type array we will inject multiple values under <list></list> tags.
* The only difference is that in case primitive type of array for multiple values we will use “value” attribute and for secondary type array we will use <ref beans>
* In short by using <list></list> we can insert arrays values.

Now if we have any collection dependencies then.

**Collection dependencies**





So from the above image we have List , Set , and Map are interface and using this interface we can create the object of any implementation classes like given below

|  |  |  |
| --- | --- | --- |
| Case 1 | Case 2 | **Case 3** |
| List obj = new ArrayList();  List obj = new Vector();  List obj = new Stack();  List obj = new LinkedList();  Vector obj = new Stack () | ArrayList obj = new ArrayList()  Vector obj = new Vector()  Stack obj = new Stack()  LinkedList obj = new LinkedList() | **But the fallowing is not possible**  ArrayList obj = new List()  Vector obj = new List()  Stack obj = new Vector()  LinkedList obj = new List() |
| Here above we are creating object of child class using parent class. So in this methodology we can create child object with the help of parent class.  **Means Parent can create child (obj)** | This is also possible of creating object.  Same class can create object of same class. | Here using child class we are trying to create parent class object. Here in this method we cannot create parent class object with child class.  **But child cannot create parent (obj).** |

Case 1- : So in the same way in spring dependency if we have any List interface dependency in any class then we can pass or inject any (implementation class like ArrayList, LinkedList, Vector, Stack etc) child class.

Case 2:- If we have ArrayList only in the class then only ArrayList can be passed or inject means same class value can be passed or inject in the bean.

Case 3:- But if we have child class Like ArrayList in the class then we cannot pass or inject parent class value in the bean.

**Question: -** Let say we have a class Test and in the class we have **– List of fruits, Set of Cricketers, and Map of (country, capital)** then how to inject the value through Collections.

1. Since Collection will accept primitive types, secondary types, primitives array and any collection type object also.
2. Suppose if we have to inject only one fruit then we will inject the one fruit using “value” attribute of property tag and if we have to inject a list of fruits using arrays then for injecting multiple values we will use <list></list> tag and between this list tag using <value> tag we will inject the values.
3. So in same way like array for Collection List also we will use <list></list> tag and between this list tag using <value> tag we will inject the values.

|  |  |
| --- | --- |
| **Collection : List(I) Primitive type**  Class **Test** {  Private List fruits;  // getter and setter of fruits.  }  So the values of fruits will be under <list> tag as shown in RHS. Here duplicate values can be injected. | <bean id=*"en"* class=*"****Test****"*>  ***<property name="*** ***fruits">***  ***<list>***  *<value>Apple</value>*  *<value>Banana</value>*  *<value>Mango</value>*  ***</list>***  ***</property>***  </bean> |

|  |  |
| --- | --- |
| **Collection : Set(I) – Primitive type**  Class **Test** {  Private map cricketers;  // getter and setter of fruits.  }  So the values of fruits will be under <**set**> tag as shown in RHS. Here set will not allow duplicate value. | **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *cricketers">***  ***<set>***  ***<value>Sachin</value>***  ***<value>Rahul</value>***  ***<value>Ganguly</value>***  ***</set>***  ***</property>***  **</bean>** |

**If it primitive type then we can use value and if it is secondary type then we can use <ref beans>**

|  |  |
| --- | --- |
| **Collection : List(I): For secondary type**  **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *fruits">***  ***<list>***  ***<ref beans>Sachin</ref beans >***  ***<ref beans>Rahul</ref beans>***  ***<ref beans>Ganguly</ref beans>***  ***</list>***  ***</property>***  **</bean>** | **Collection : Set(I): For secondary type**  **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *cricketers">***  ***<set>***  ***<ref beans>Sachin</ref beans>***  ***<ref beans>Rahul</ref beans>***  ***<ref beans>Ganguly</ref beans>***  ***</set>***  ***</property>***  **</bean>** |

**Now for Map (country, capital)**

|  |  |
| --- | --- |
| **Collection : Map(I) – Primitive type**  Class **Test** {  Private Map countryCap;  // getter and setter of fruits.  }  So the values of fruits will be under <***map*** > tag as shown in RHS. Here set will not allow duplicate value.  Here in Map we will use <entry key=” “ value =””/> instead of only value tag | **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *countryCap">***  ***<map>***  ***<entry key=”IND” value=”Delhi”/>***  ***<entry key=”PAK” value=”ISL”/>***  ***<entry key=”Chi” value=”Begi”/>***  ***</map>***  ***</property>***  **</bean>** |

**Note:** As here we are using Collection interface (List, Set and Map) for DI and these will accept its implementation class’s object /values. So spring provide the fallowing things as mentioned below

1. For List interface the default implementation class object (ArrayList class object) value will be injected i.e. in case of <list> tag internally it will create ArrayList object and this ArrayList object will be inject to List in the class.
2. For Set interface the default implementation class object (LinkedHashSet class object) value will be injected i.e. in case of <set> tag internally it will create LinkedHashSet object and this LinkedHashSet object will be inject to Set in the class.
3. For Set Map the default implementation class object (LinkedHashMap class object) value will be injected i.e. in case of <map> tag internally it will create LinkedHashMap object and this LinkedHashMap object will be inject to Map in the class.

**But: if** we have any implementation particular class instead of interfaces. Suppose if we have Vector reference in the Test class instead of List interface reference. So in this case Vector will not accept the default ArrayList object provide by spring cannot be injected because ArrayList and vector are sibling (same level) i.e. Vector and ArrayList does not have Parent child relationship.

**Question:** **So now the problem is if we have particular collection type then the above tags <list>, <set>, <map> are not suitable to inject particular type. So what is the solution?**

**Ans: 1-** if we have Vector in the Test class then only Vector object can be injected and to make this possible the use: The : <util:list list-class “java.util.Vector”> and under this tag using <value> tag we will inject the vector object.

|  |  |
| --- | --- |
| **Collection : Vector(c) Primitive type**  Class **Test** {  Private Vector fruits;  // getter and setter of fruits.  }  So the values of fruits will be  <***util:list list-class “java.util.Vector***”> tag as shown in RHS and this tags come after adding the below line in above of xml file    **<beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>**  In this case the spring container creates only Vector class object and this object then can be injected into class or bean. | **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"**  [**http://www.springframework.org/dtd/spring-beans-2.0.dtd**](http://www.springframework.org/dtd/spring-beans-2.0.dtd)**>**  **<beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>**  **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *fruits">***  **<*util:list list-class “java.util.Vector*”>**  ***<value>Apple</value>***  ***<value>Banana</value>***  ***<value>Mango</value>***  **</*util:list list-class “java.util.Vector*”>**  ***</property>***  **</bean>**  **</beans>** |

**Ans2-** if we have TreeSet in the Test class then only TreeSet object can be injected and to make this possible the use: The : <util:list set-class “java.util.TreeSet”> and under this tag using <value> tag we will inject the vector object.

|  |  |
| --- | --- |
| **Collection : TreeSet(c) Primitive type**  Class **Test** {  Private TreeSet cricketers;  // getter and setter of fruits.  }  So the values of fruits will be  <***util:Set Set-class “java.util.TreeSet***”> tag as shown in RHS and this tags come after adding the below line in above of xml file    **<beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>**  In this case the spring container creates only TreeSet class object and this object then can be injected into class or bean. | <beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>  **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *fruits">***  **<*util:Set Set-class “java.util.TreeSet*”>**  ***<value>Apple</value>***  ***<value>Banana</value>***  ***<value>Mango</value>***  **</*util:Set Set-class “java.util.TreeSet*”>**  ***</property>***  **</bean>**  **</beans>** |

**Ans3-** if we have Hashtable in the Test class then only Hashtable object can be injected and to make this possible the use: The : <util:Map Map-class “java.util.HashTable”> and under this tag using <value> tag we will inject the vector object.

|  |  |
| --- | --- |
| **Collection : Hashtable(c) – Primitive type**  Class **Test** {  Private Map countryCap;  // getter and setter of fruits.  }  So the values of fruits will be under  ***<util:Map Map-class “java.util.Hashtable”>*** tag as shown in RHS and this tags come after adding the below line in above of xml file | <bean id=*"en"* class=*"****Test****"*>  ***<property name="*** ***countryCap">***  ***<util:Map Map-class “java.util.HashTable”>***  *<entry key=”IND” value=”Delhi”/>*  *<entry key=”PAK” value=”ISL”/>*  *<entry key=”Chi” value=”Begi”/>*  ***</util:Map Map-class “java.util.HashTable”>***  ***</property>***  </bean>  **<beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>**  In this case the spring container creates only TreeSet class object and this object then can be injected into class or bean. |

So by using utill namespaces we can provide solutions for particular collection classes. We have to use schema based xml. It is not possible with DTD based xml.

**Properties (DI in property file)**

For injecting properties we have special type of configuration. Actually properties class is given for reading properties file. So here in case if we want to pass/inject anything to property file document so for that spring has given some different type of dependency injection for particular properties only. If a class contains any property type like given below.

|  |  |
| --- | --- |
| Class Test {  private property driver;  //setters and getters  } | Suppose here we have one property type parameter driver through we want pass drivers data (url, username password etc) through property files. So we can pass these information by two types (1- Using DTD based xml , 2- Xmlns based xml) |
| **<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"**  [**http://www.springframework.org/dtd/spring-beans-2.0.dtd**](http://www.springframework.org/dtd/spring-beans-2.0.dtd)**>**  <bean id=*"en"* class=*"****Test****"*>  ***<property name="*** ***driver">***  ***<props>***  *<prop key=”IND”>Delhi</prop>*  *<prop key=”PAK”>ISL</prop>*  *<prop key=”USA”>Newark</prop>*  ***</props>***  </bean> | Here all the property key and value will be between <props></props> tag.  The value against key will be between<prop></prop> tag. Here we don’t have any value attribute as in case of we have value attribute in MAP. |

Note: Here it is not possible to read external property file data using DTD based configurations. So to read external property file data we have to use namespace as we did in case of collection classes like Vector, TreeSet and Hashtable.

|  |  |
| --- | --- |
| **<beans xmlns=”beans-namespace” xmmlns:util=”util-namespace>**  **<bean id=*"en"* class=*"Test"*>**  ***<property name="* *driver">***  ***<util:properties location=”classpath:resources/drivers.properties>***  **</bean>**  **</beans>**  **Here we have given the exact path using classpath: file location.Here the in above xml config file, property file is inside resource folder.** | So here to read data from property file and to inject same data to class variable we will use beans-namespace and utill-namespace |

Example

**Q:**  In properties we can have any type values or only string type values can be passed:

**A:** Only String type values (Keys and values).

**Dependency-Check (DC)**

So the last session we discussed about the fallowing things given below.

1-      IOC Container

2-      Dependency injection type (1- Setter Type DI 2- Parameterized constructor)

Note: - In case if we have a class with only one single parameterized constructor to inject a property. So if we have only one constructor inside the class then without calling the constructor we cannot create object of class. So in case of constructor DI injecting properties values are mandatory by default. But in case of setter DI it is not mandatory to inject properties values. Actually we don’t have any limitation to supply values into setter DI but one thing is that we cannot override the values, while in case of CB DI we have the facility to override the values.

In short in case of constructor DI compulsory it is mandatory to supply the values into parameterized constructor then only it will create an object of class. But in case of setter DI it is not mandatory to supply values into setter method.

Now if we want to make setter DI also compulsory then spring has given one concept called **Dependency-Checking.** The main aim of this concept is to make setter DI is mandatory. Now the question is where we need to apply this Dependency-Check i.e.  for which bean we want to apply the DC.

Let’s say we have one bean with multiple parameters so to make the parameter mandatory we have to use one attribute inside bean tag called [dependency-check] along with bean id and class.

|  |  |
| --- | --- |
| <bean id =”c”  class= “com.test” dependency-check=”value”></bean>  Here IOC will make it mandatory to supply values into setter method. |  |

|  |  |
| --- | --- |
| Class Car{  Private String carname;  Private Engine engine;  // setter method of carname & engine  } | Class Engine{  Private String modelYear;  // setter method of modelYear  } |

There is four values can be set in dependency-check

|  |  |
| --- | --- |
| 1-   none | By default DC value will be none means it’s not mandatory to supply values into setter method.  **<bean id =”c”  class= “com.Car” dependency-check=”none”>**  **No values need to supply inside property.**  **</bean>** |
| 2-   simple | dependency-check=”simple”. In this case only primitive type (like String, int, float) values will be mandatory to supply into primitive setter method as mention below.  **<bean id =”c”  class= “com.Car” dependency-check=”simple”>**  **<property name =”carname” value=”audi”/>**  **</bean>**  In case if we don’t pass carname value then IOC will force to supply value of carname and unsatisfied DI exception will occur. |
| 3-   objects | dependency-check=”object”. In this case only secondary object type (like any class references) reference will be mandatory to supply into object or secondary type setter method.  **<bean id =”c”  class= “com.Car” dependency-check=”objects”>**  **<property name =”** engine**” ref=”en”/>**  **</bean>**    Where en is the reference of Engine class created as given below.    **<bean id =”en”  class= “com.Engine” >**  **<property name =”** modelYear**” value=”2019”/>**  **</bean>** |
| 4-   all | If we want to make primitive type and secondary type both mandatory then we will use dependency-check=”all”. Now in this case the beans will as given below.  <beans>  **<bean id =”en”  class= “com.Engine”  dependency-check=”simple”>**  **<property name =”** modelYear**” value=”2019”/>**  **</bean>**  **<bean id =”c”  class= “com.Car” dependency-check=”objects”>**  **<property name =”carname” value=”audi”/>**  **<property name =”** engine**” ref=”en”/>**  **</bean>**  </beans>  Here Engine bean declared as **simple**  DC so modelYear value is mandatory to pass into modelYear Setter method (SetModelYear()) and in case of Care bean ,it is declared as all DC so both (primitive values & secondary ref) type will be mandatory to pass into setter method (SetCarName() and SetEngine()) |

So the main aim of dependency-check is to make setter method mandatory. By default

**Dependency-check** value is none, means it will not make it mandatory because without passing values into setter method we can create object of class. But for parameterized constructor it is mandatory to pass values because until we pass the values it will not call constructor and without calling constructer we cannot create objet of class.

**Disadvantage of dependency-check:**

**While we use DC, it makes all primitive / secondary type mandatory so in this case it is compulsory to supply values from IOC to its setter method. Now let’s say in class if we have 10 primitive types and we want only one or two to mandatory then in this case it is not possible because DC will make all the primitive type mandatory and in this case compulsory we need to supply all the values from IOC to its setter methods.**

**UseCase**: If we have simple DAO class and in this DAO class we have four dependencies in String (driver, URL, username, password) and we have one getConnection () method and if we want to connect with MySql database.

So in case of MySql database it is not mandatory to enter username and password, without username and password we can connect MySql database. Only driver and URL are mandatory to connect.

Now if we declare dependency-check=”simple” then in this case IOC will make all the four dependencies in String (driver, URL, username, password) mandatory. So this is the disadvantage that username and password is not required even though unnecessarily we compulsory need to pass through xml file.

**Solution:** To solve this spring has given one annotation (@Required). By using this annotation we can make setter DI mandatory. For making any dependencies (setter method) we just need to declare @Required on top of that that setter method to which we have to make it mandatory. Like in the above use case driver and URL is mandatory so we will declare @Required on top of driver and URL setter method and in this case we don’t need to mention DC inside bean in xml file. So in this case IOC will make mandatory only two values (driver & URL).

Note: To make it use @Required on setter method lever we need to activate. To activate this @Required annotation IOC container need to create one class object before going to create our DAO class object. This class name is [**RequiredAnnotationBeanPostProcessor**]. To activate this class just declare this class under bean tag in the xml file.

|  |
| --- |
| **Question: How many types of annotations we have?**  Based on the place (level) there are three type annotations.  1-      Class level annotation  2-      Method level annotation  3-      Property (parameter) level annotation |

**Note: Dependency check is not recommended where only few properties (variable) are mandatory.**

**Example: Pending**

**Depends-On**

* As here in the above example we had two classes (Car & Engine).
* The Car class has two dependencies (carname- Car class String variable and engine- Engine class reference).
* The Engine class has one String variable dependency modelYear

In the above example First Car class object will be created after that Engine class object will be created by IOC container. Now if we want that before creating Car class object, Engine class object creation first is mandatory i.e. in other words if we want Car class **depends-on** Engine class then Depends-On concept comes into picture.

So dependency-check aim is to make setter DI mandatory while by using **depends-on** we can apply dependencies for classes. So if one class (A) is depends on other class (B) then this can be achieved by applying depends-on concept. In this case IOC container will create object of class (B) then only it will create object of class (A) because class(A) is depends-on class(B).

**UseCase: -**

In MVC based architecture fallowing layers as mentioned below.

1. **View Layer:** This is nothing but presentation layer used to make request to controller or to get response from controller.
2. **Controller Layer:** This Layer is nothing but controller who sends the user request to Model layer and based on the request gets the response from model layer.
3. **Model Layer:** The Model layer consists of **Business** and **DAO** part. Here in Model layer Business part will take request from Controller layer and sends this request to DAO. DAO based on request came from business part will send the response to business part. Business part will send this response to Controller and Controller intern send this response to view layer.

**Diagram- Pending.**

Let us consider we have one **Class (A) for controller layer** and two more classes in model layer [**Class (B) for Business part and Class(C) for DAO part**.] Now if all three classes are tightly dependent on each other [i.e. Class (A) is dependent on Class (B) and Class (B) is dependent on Class(C)].

In other world:-

Before creating object of Class (A), Class (B) object creation is mandatory **and** before creating object of Class (B), Class(C) object creation is mandatory, then in such condition we can apply Depends-On concept. So while applying Depends-On, IOC container first will create Class(C) object then Class (B) object and finally Class (A) object.

|  |  |  |
| --- | --- | --- |
| Class A {  A(){  S.o.p(“A Class Object”);  }  } | Class B {  A(){  S.o.p(“A Class Object”);  }  } | Class C {  A(){  S.o.p(“A Class Object”);  }  } |
| Spring.xml  <beans>     <bean id= “a” class =”A”  depends-on=”b”/>     <bean id= “b” class =”B”  depends-on=”c”/>     <bean id= “c” class =”C” />  </beans> | | |
| **Note: Two classes cannot be depends-on each-other .i.e. classes cannot be mutually depends-On.** <beans>     <bean id= “a” class =”A”  depends-on=”b”/>     <bean id= “b” class =”B”  depends-on=”c”/>     <bean id= “c” class =”C”  depends-on=”a”/>  </beans>    **Here Class A and Class C are mutually deepens-on each other which cannot happen**. In this case container will be confused that which class objects will create first Class A or Class B. So Mutual Dependency are not possible | | |

Example Pending.

**Setter and Constructor DI using P-NameSpace and C-NameSpace**

**P-NameSpace:**It stands for Property NameSpace and can be used in injecting values into setter method.

**C-NameSpace**: It stands for Constructor NameSpace and can be used in injecting values into

                            Parameterized constructor

Using these NameSpace we can use number of xml tags. Hence It improves performance while reading xml file

**Difference between [(**Using normal case DI**) Vs (**Using P-NameSpace and C-NameSpace DI**)]**

**Setter DI using normal case:**

|  |  |
| --- | --- |
| **Class Car{**  Private String carname;      Private Engine engine;  // Getters and Setters of carname and engine  **}** | **Class Engine{**  Private String modelYear;       Public Engine(String modelYear){       this.modelYear=modelYear;  }  // Getters and Setter of modelYear property.  **}**  **Here we have one parameterized constructor.** |
| **Spring.xml**    **<beans>**  **<bean id =”c”  class=”Car”>**                 <property **name** = “carname”  **value** =”Audi”/>                 <property **name** “engine”  **ref**=”**en**”/>  **</bean>**  **<bean id =”en”  class=” Engine”>**                 <constructor-arg  name=“carname” **value** **=”**modelYear**”/>**  **</bean>**  **</beans>** | |

**The corresponding Spring.xml of Setter DI using P-NameSpace.**

|  |
| --- |
| **Spring.xml**    **<beans>**  <bean **id** =”c”     **class**=”Car” **p:carname**=”Audi”  **p:engine-ref**=”**en**”/>     <bean **id** =”**en**”  **class**=”Car” **c: modelYear** =”2018/>  **</beans>**    **Note:**  **P: NameSpace:**So here we don’t need to use <property> tag. Inside <bean> tag only we can declare **p**:primitive\_Type = “value” and **p**:sercornd\_tyep-ref=”Reference of created class”    **C:NameSpace :**we don’t need to use <constructor-arg> tag Inside <bean> tag only we can declare **c:**primitive\_type and **c**:sercornd\_tyep-ref=”Reference of created class”    **This is recommended approach for DI instead of normal case.** |

**Autowiring (Automatic DI)**

**Aim:**The main aim of Autowiring is automatic dependency injection.

**Limitation:** We can inject Secondary types (Class reference) only

* Autowiring can be used for automatic dependency injection for secondary type only.
* Autowiring can be applied setter and constructor both.
* This autowire attribute can be used inside <bean> tag only.
* There is five way to do Autowiring

1. byType                   (Can be used only for setter DI)
2. byName                 (Can be used only for setter DI)
3. constructor           (Can be used only for constructor DI)
4. autodetect            (Can be used for both setter DI & constructor DI)
5. no                          (By default Autowiring is no.)

**UseCase of Autowiring:**

Suppose in a Bank application we have 100 dynamic pages. For each 100 pages we will have 100 controller classes or for 2 pages we will have one controller class depends on requirement.

Now for each 100 controller class we will have 100 Business classes and for each 100 business classes we will have 100 DAO classes.

|  |  |  |
| --- | --- | --- |
| Input request | Controller Class C1,C2,C3….100  Ref B2 in C2  Ref B1 in C1 | **Business** Class B1,B2,B3….100 **DAO** Class D1,D2,D3….100  D2  Ref D2 B2  Ref D1 in B1  D1  DB |

As per the diagram we can see that each 100 controller classes have reference of corresponding business class from 100 business classes. Similarly all 100 business class has references of corresponding DAO class reference from 100 DAO classes.

Now from IOC container we can create object of 100 container classes by writing spring.xml. For creating object we have write 100 of <bean> tag and inside this <bean> tag we will have to dependency injection for dependent classes (Business and DAO). This is not easy and convenient way of dependency injection.

To solve this problem Autowiring concept can be applied. We just need to declare one attribute autowire=”value” inside <bean> tag and now the IOC container will do all DI for business and DAO classes automatically.

|  |
| --- |
| <bean id=*"dc"* class=*"C1"* autowire=*"byType"/*>  <bean id=*"dc"* class=*"C2"* autowire=*"byType"/*>  <bean id=*"dc"* class=*"B1"* autowire=*"byType"/*>  ………….till 100 |

**Note**: For 100 controllers we will have to declare 100 time autowire attribute in each of 100 controller bean tag which is also not a convenient way. So to solve this problem we just need to declare [**default-autowire=”byType**” or “**byName”**] under <beans> tag

|  |
| --- |
| **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xmlns:util=*"http://www.springframework.org/schema/util"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd***  ***http://www.springframework.org/schema/util***  ***http://www.springframework.org/schema/util/spring-util.xsd"***  **default-autowire=*"byType"*>**  **<bean>**  **<property…../>**  **</bean>**  **</beans>** |

**Example: How autowire works.**

Here we have 100 controller classes, 100 business classes and 100 DAO class. All three classes are dependent on each other using HAS-A relationship, means Business has reference in Controller class and DAO class has reference in Business class. So we have to write 200 bean tag (100 for controller class and 100 for business class) and we will have to mention [autowire=”byType”] in all 200 bean tag.

Instead of writing in 200 bean tag, we can mention this [autowire=”byType”] inside parent <beans> tag. So solve this problem we just need to declare [default-autowire=”byType” or byName or constructor] under <beans> tag. Now we don’t need to configure [autowire=”byType”] in every 200 <bean> tag.

Q: **What happens if we apply [default-autowire=”byType”] inside <beans> tag?**

**Let us understand with the help of example.**

|  |  |
| --- | --- |
| **Class Car{**  Private Engine engine;  // Getters and Setters of carname and engine  setEngine(Engine engine){}  **}**  **In car class is having dependency on Engine Class.** | **Class Engine{**  Private String modelYear;  // Getters and Setter of modelYear property.  **}** |
| **Now if for this Car class if we apply Autowiring then**  **Spring.xml**    **<beans>**  **<bean id =”en”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2015**”/>**  **</bean>**  **<!—In Engine class we have one primitive type and for  this PT AW is not applicable --  >**  **<!—In Car class we have one secondary type and for  this ST AW is applicable --  >**  **<bean id =”c”  class=”Car” autowire=”byType”>**                 <property **name** “engine”  **ref**=”**en**”/>  **</bean>**  **</beans>** | |

Here in the Car class we have one setter method (setEngine (Engine engine) {}) and for doing setter automatic DI we can use byType or byName or autodetect (setter or constructor).

In the Car <bean> tag we are [applying autowire=”byType”] in this <bean> tag we don’t need to inject Engine class reference [ref=”en”] manually. The Engine class bean id (reference)”en” will inject into car class automatically. So now the Car class <bean> tag will be like as given below.

|  |
| --- |
| **<bean id =”c”  class=”Car” autowire=”byType”>**        <!—We don’t need to inject Engine Class Reference -- >  **</bean>** |

**Question:** **Now the question is when this automatic injection will happen?**

**Answer:**When we load our spring.xml document into IOC. IOC will read this xml document and will try to find out that which bean class is applied for autowire like here in the Car class bean tag autowire is applied.

Now it will scan all the declared parameter or properties (primitive type and secondary type) in the Car Class. After reading/scanning if container finds any setter method for parameter (for Secondary class reference only) then it will treat this parameter as dependent parameter for Car class.

In the Car class we have only single parameter (Secondary Type parameter) engine and for this parameter we have one setter method setEngine (Engine engine) {}, so container will treat this engine parameter as dependent of Car class.

Now the container will check the type of this dependent (engine) and this dependent (engine) is of Engine type. Now after finding this type of dependent from Car class it will search this type of dependent in xml file and if container finds any valid same type (Engine) in the xml file then based on valid type container will inject(bean id=”en”) automatically in Car class.

**Case:** Suppose In Car Class we have one more reference (private Engine1 engine1) and we don’t have setter method for this engin1 reference then in this case IOC container will not treat this reference as dependent of Car class. In short the reference with its setter method will be treated as dependent for autowire. Example is given below.

|  |  |
| --- | --- |
| **Class Car{**  Private Engine   engine;      Private Engine1 engine1;   // Don’t have setter Method  // Getters and Setters of  engine only  setEngine(Engine engine){}  **}** | **Class Engine{**  Private String **modelYear**;  // Getters / Setter of modelYear property.  **}**  **Class Engine1{**  Private String modelYear;  // Getters / Setter of modelYear property.  **}** |
| **Now if for this Car class if we apply Autowiring then**  **Spring.xml**    **<beans>**  **<bean id =”en”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2015**”/>**  **</bean>**  **<bean id =”en1”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2016**”/>**  **</bean>**    **<!—**In Engine class we have one primitive type and for  this PT AW is not applicable**--  >**  **<!—**In Car class we have one secondary type and for  this ST AW is applicable**--  >**  **<bean id =”c”  class=”Car” autowire=”byType”></bean>**  **</beans>** | |
| **Explain1-** Car class is having one dependency (**engine with setter method**) onEngine Class, so it will be injected automatically.  **Explain2- Car class**is having another dependency (**engine1 without setter method**) on **Engine1** Class, so it will not be injected automatically. It will not be treated as dependent of**Car class.** | |

**Case2**- Suppose if we have one more eligible type (Engine) with different bean id (en1) as shown in the above xml then ambiguity will occur. Because when IOC container will scan Spring.xml document by using byType it will get two eligible types with same name Engine (having different bean id). Now in this case container will get confused and we will get ambiguity exception.

So by using byType only one eligible type can be injected automatically and if we have more than one eligible type with same name (having different bean id) then container will get ambiguity exception- Bean creation exception.

**Question: Now the question arises that how can we resolve this ambiguity problem?**

**Answer**:  When we are using byType for Autowiring and we have more than one eligible type with the same name(having different bean id) then in this case we need to tell the container that this eligible type (Engine- having bean id en1) is not taking part in Autowiring or it is not eligible for Autowiring.

For this we have one attribute [autowire-candidate=”false”] that we have will have declare inside the eligible Type (Engine-having bean id=en1) like as mention below.

|  |
| --- |
| **<**bean id =”en1”  class=” Engine”**autowire-candidate=”false>**                 < property  name=“ modelYear” **value** **=”**2016**”/>**  **</bean>**  **By default its values is true.** |

Now when the container will scan the spring.xml file then it will find that another eligible type (Engine- having bean id en1) have [**autowire-candidate=”false]** and then container will understand that this eligible type is not taking part in Autowiring and this case container will not throw any ambiguity. And then only first eligible type (Engine- having bean id =en) will be injected automatically into Car class.

**Autowiring using byName:**

When we are using byName for Autowiring then in this case the IOC container will do automatic injection by its Type (Engine) as well as its name (engine). The IOC container will scan the spring.xml file and if it finds the type (Engine) and name (engine) inside the <bean> tag then only it will inject automatically. As shown in the below example.

|  |  |
| --- | --- |
| **Spring.xml**    **<beans>**  **<bean id =”engine”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2015**”/>**  **</bean>**  **<!—**Injected automatically**-- >**  **<bean id =”engine”  class=” Auto”>**                 < property  name=“ modelYear” **value** **=”**2015**”/>**  **</bean**  **<!—**Will not Injected automaticall**y  -- >**  **<bean id =”en1”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2016**”/>**  **</bean>**  **<!—**Will not Injected automaticall**y  -- >**  **<bean id =”c”  class=”Car” autowire=”byName”>**  **</bean>**  **</beans>** | **Class Car{**  Private Engine **engine**;    // engine1 Don’t have setter Method  // Getters/Setters of  engine only       setEngine(Engine **engine**){  }  **}** |

In case of byName when container will read xml file then it will try to find the bean having same eligible type (Engine) and dependent name (engine) what exactly we have in Car class. In the above example we have one bean having (class =”Engine”) and (id =”engine”) and these things are exactly matching with eligible type (Engine) and dependent name (engine) in the Car class. So now container will go for injection automatically this bean only.

In the above example we can see that there one more bean with same eligible type (Engine) having different id. Here in such situation we will not get any ambiguity what we get in case of byType because here in case of byName container do the injection with eligible type and dependent name both. So in case of byName we don’t need to mention [**autowire-candidate=”false]**for second bean having id=”en1”, only bean id should be unique for all bean.

**Example Pending**

**Autowiring using constructor:**

When we are using constructor for Autowiring then internally it uses byType Autowiring mechanism.

|  |  |
| --- | --- |
| **Class Car{**  Private Engine   engine;  **Car(Engine engine){**  **this.engine = engine;**  **}**     setEngine(Engine engine){}  **}** | **Class Engine{**  Private String **modelYear**;  // Getters-Setter of modelYear property.  **}**  **Class Engine1{**  Private String modelYear;  // Getters / Setter of modelYear property.  **}** |
| **Now if for this Car class if we apply Autowiring then**  **Spring.xml**  **<beans>**  **<bean id =”en”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2015**”/>**  **</bean>**  **<!—**en will be injecting automatically **-- >**  **<bean id =”en1”  class=” Engine”>**                 < property  name=“ modelYear” **value** **=”**2016**”/>**  **</bean>**  **<!—**en1 will not be injecting automatically will get ambiguity and avoid this autowire-candidate=”false” we will have to use**-- >**  **<bean id =”c”  class=”Car” autowire=”byType”></bean>**  **</beans>** | |
| **Explain1-** Car class is having one dependency (**engine with setter method**) on Engine Class, so it will be injected automatically.  **Explain2- Car class**is having another dependency (**engine1 without setter method**) on **Engine1**Class, so it will not be injected automatically. It will not be treated as dependent of**Car class.** | |

Here in case of constructor Autowiring, IOC container will scan constructor of the class and whatever type is declared inside the constructor, based on that type it will try to find out the eligible type in xml file and based on valid type found in xml file it will inject bean id into car class parameterized constructor. If there is more than one same eligible type (Engine) is present in the xml exactly what we have in class Car constructor then container will throw ambiguity exception.

**Example Pending**

**Autowiring using autodetect:**

This autodetect Autowiring is used when we don’t know whether class have setter DI or constructor DI then we will go for autodetect. Like we have one use case where we have 100 of controller classes, 100 of business classes and 100 of DAO classes. Now in such case we don’t know which class is setter or constructor DI then in this condition we will go with autodetect Autowiring. We have some sort of cases in autodetect.

The autodetect also uses byType search mechanism.

**Case1- When class has constructor as well as setter method and the class (Car) does not have any default constructor.**

In this case container will do constructor DI automatically using byType search mechanism. Because when we create the object of class that time only this constructor will be called first.

**Case2- When class has constructor as well as setter method and the class (Car) and along with parameterized constructor we have one default constructor.**

In this case first of all default constructor will be called during object creation of class and then instead of calling parameterized constructor it will call setter method for automatic dependency injection

**Conclusion:**

**So finally byType approach is recommended approach in case of Autowiring DI**

**Example:**

**Dependency Injection using Stereo type Annotations.**

**Autowiring Dependency Using Annotation approach:**

**@Autowired:**

* As we know that while we use xml based configuration for Autowiring then under <bean> or <beans> (parent tag) we have to use the fallowing attribute as mention below.
* **autowire=”byType” or “byName” or “constructor” or “autodetect”.**
* But if we want to use annotation approach Autowiring DI for your dependent parameter then we will use **@ Autowired** on top of dependent parameter.
* It means this @Autowired annotation is property level annotation.
* This **@Autowired** by-default uses “byType” search mechanism for automatic DI.
* As we know that in case of “byType” there is chance to get ambiguity so to resolve the ambiguity along with @Autowired annotation we can use @Qualifier(value=”en”) annotation.

**@Qualifier (value=”id”):**

* This annotation is used to resolve ambiguity and inside this we have to use (value=”id”).
* Here id (which is unique for every bean) is nothing but the bean reference (id) to which we have to go for Autowiring.
* So the combination of (@Autowired **&** @Qualifier (value=”id”)) can be used or automatic DI without any ambiguity problem and this case we don’t need to use any autowire and **autowire-candidate=”false”** (for removing ambiguity) attribute inside spring configuration file.
* To activate this ((@Autowired **&** @Qualifier) we need to instantiate one class for both.
* The class name is (**AutowireAnnotationBeanPostProcesser**). Like for **@Required** we have (**RequiredAnnotationBeanPostProcessor).**
* **We**just need to mention this class inside <bean> tag as given below
* **<**bean class=”**AutowireAnnotationBeanPostProcesser”**/**> this line will activate**((@Autowired **&** @Qualifier) annotation.
* **<**bean class=”**RequiredAnnotationBeanPostProcessor”**/**> this line will activate**((@Required))

**Example:**

**Difference between Xml Based and Annotation based Autowire:**

|  |  |
| --- | --- |
| XML Based Autowiring: | Annotation Approach Autowiring: |
| **1-**We have to apply autowire attribute under <bean> tag. | **1-**Here we have keep @Annotation on top of Secondary type property. |
| **2-**We have to give attribute option value (byType or byName or constructor or autodetect or No) | **2-**Here to resolve ambiguity @Qualifier (value =”bean id”) have to keep on top of secondary type of property. |
| **3-**To reduce ambiguity we use one more attribute “autowire-candidate=false” | **3-** |
| **4-**We no need to do any manual DI and we can reduce number of property tags. | **5-** |

**Explanation**:

**Stereo Type Annotations to create automatic objects:**

There are four stereo type annotations to create automatic object of the classes used in the spring application packages.

|  |  |
| --- | --- |
| @**Controller** | Used for controller package classes under MVC based application. |
| @**Repository** | Used for DAO package classes |
| @**Service** | Used for Business package classes |
| @**Component** | Used for Non MVC classes like utility package classes |

|  |  |  |  |
| --- | --- | --- | --- |
| com.ds.ems.controller | com.ds.ems.business | com.ds.ems.dao | com.ds.ems.utility |
| @Controller  MyControllerClass1{  @Autowired  Private B1 b; } | @Service  MyBussnessClassB1{  @Autowired  Private DAO1 d; } | @Repository  MyDAOClassDAO1{  @Autowired  Private CF cf; } | @Component  MyConnectionFactory{    } |
| @Controller  MyControllerClass2{  @Autowired  Private B2 b;} | @Service  MyBussnessClassB2{  @Autowired  Private DAO d; } | @Repository  MyDAOClassDAO2{  @Autowired  Private CF cf; } |
| @Controller  MyControllerClass3{  @Autowired  Private B3 b;} | @Service  MyBussnessClassB3{  @Autowired  Private DAO d; } | @Repository  MyDAOClassDAO3{  @Autowired  Private CF cf; } |
| So on…….  Here inside package for controller classes we have used @contorller annotation | So on ……..  Here inside package for bussines classes (B1,B2,B3..) we have used @Service annotation | So on ……  Here inside package for DAO classes we have used @Repository  annotation | So on …….  Here inside package for non MVC  classes like connection factory we have used @Componenet annotation |

Here in the above table the corresponding spring configuration file will be

|  |
| --- |
| <bean id “c1” class=”MyControllerClass1” >       <property name = “b” ref=”servise1”>  </bean>  <bean id “b1” class=” MyBussnessClassB1” >       <property name = “d” ref=”dao1”>  </bean>  <bean id “b1” class=” MyDAOClassDAO1” >       <property name = “cf” ref=”conFactory”>  </bean>  <bean id “b1” class=” MyConnectionFactory” >       <property name = “connectionFactory” value=””>  </bean>  <!—One connection object will be inject in all DAO classes -- >  **Without autowire=”byType”**. Here we have to do manual DI in each <bean> tag.    <bean id “c1” class=”MyControllerClass1”  autowire=”byType” >       <!—Don’t need to do manual DI in case autowire-- >  </bean>  <bean id “b1” class=” MyBussnessClassB1”  autowire=”byType >            <!—Don’t need to do manual DI in case autowire-- >  </bean>  <bean id “b1” class=” MyDAOClassDAO1” autowire=”byType >            <!—Don’t need to do manual DI in case autowire-- ></bean>  <bean id “b1” class=” MyConnectionFactory” >       <property name = “cf” value=””>  </bean>    **With autowire=”byType”**    This is for one (Controller Class, Business Class and DAO class). Similarly for 1000 Controller , 1000 Business and 1000 DAO we have to write 3000 <bean> tag and 3000 time autowire=”byType” for automatic DI. And one bean for connection pool (Connection Factory).  Note: We can reduce autowire=”byType” from 3000 times to only one time. We just need to mention default-autowire=”byType” in the parent <beans default-autowire=”byType”>. So  <beans default-autowire=”byType”>     <bean id “c1” class=”MyControllerClass1 >       <!—Don’t need to do manual DI in case autowire-- >  </bean>  <bean id “b1” class=” MyBussnessClassB1 >            <!—Don’t need to do manual DI in case autowire-- >  </bean>  <bean id “b1” class=” MyDAOClassDAO1” >            <!—Don’t need to do manual DI in case autowire-- >  </bean>  <bean id “b1” class=” MyConnectionFactory” >       <property name = “cf” value=””>  </bean>    </beans><!—End of parent root tag --- >             Now in this way we can reduce lots of<property> tags.           We don’t need to do manual DI.           Spring IOC will take care for auto DI. |

Note: Here in the above table we have one package for controller class , one package of business class and one package for DAO class and each package we have 1000 -1000 classes.

Now to create object of each classes we have write <bean> tag in configuration file and in this way we have to write at least 3000 <bean> tag for creating of 3000 classes. But…..

Spring can take care automatic object creation of these 3000 classes using auto scanning concept. Spring provide auto-scanning concept on package level. We just need to do auto-scanning for all the package and spring will take responsibility of object creation of all the classes inside the package.

Here in-case of above example if we apply auto-scanning for all the packages (Controller package, business package and DAO package) then spring IOC automatically scan all the packages and inside the package if IOC finds the classes by using stereo type annotations then automatically IOC will create object of all the classes.

 So finally we can say that IOC container will create object of all the classes by reading all the packages and after creation of object, by using autowire, IOC will do automatic DI. In this way spring reduces developer burden, we don’t need to create object and manual DI container will do all the things.

The corresponding tag for auto-scanning is given below.

We have to use schema based configuration.

We have to add xmlns name-space.

We have to use context name-space

|  |
| --- |
| <beans xmlns name-space context name-space>      <context: component-scan    base-package=”packageName”/>  </beans> |

 Syntax mention above for scan package: And corresponding example give below.

|  |
| --- |
| <beans xmlns name-space context name-space>      <context: component-scan    base-package=” com.ds.ems.controller”/>      <context: component-scan    base-package=” com.ds.ems.business”/>      <context: component-scan    base-package=” com.ds.ems.dao”/>      <context: component-scan    base-package=” com.ds.ems.utility”/>      <!—To activate all the annotation used in the application use line given below -- >      <context: annotation-config>  </beans> |

**Note :** <context: annotation-config>: this line will activate all type of annotation used in spring application(@Autowire and @Required, @**PostConstruct**, @**PreDestroy**) . We don’t need to write or remember particular classes for activating particular Annotation like we had written earlier to activate (@Autowire and @Required)

|  |
| --- |
| <bean class = “**AutowireAnnotationBeanPostProcesser” />** |

**This above line will activate**((@Autowired **&** @Qualifier) annotation.

|  |
| --- |
| <bean class= “RequiredAnnotationBeanPostProcessor” /> |

**this line will activate**((@Required))

So finally we can say that as soon as we load the xml spring configuration file into IOC container, it will scan the entire package and if it finds classes with stereo type annotation at class level then it will create object of all the classes after creating object if IOC finds any @Autowire annotation at property level then container will do automatic DI. In this way we don’t need to write lots of lines in xml spring configuration file and we just need to concentrate on logic and functionality.

**Note:**If a class contains primitive type then in this case @Component annotation will not work because the class having primitive type dependency will not instantiate automatically and thus we can’t do automatic DI so for the class having primitive type dependency, we will have to write <bean> tag for object creation and we will have to write <property> tag for DI manually and

**[<context: annotation-config>** and **<context: component-scan base-package=”packageName”/>**] will not work for such classes

Example:

So far we have learned about all type of dynamic DI (like PT, ST PT Array, ST Array and collection). Now will learn Static types DI.

**Static Variable Dependency Injection:**

Up-till now we have learns how to DI for dynamic variable. In case if any class contains static type variable then DI style is different.

|  |  |
| --- | --- |
| Public class Car{     Private Static String name;    // Setter of carname    Public Static void setName(String name){        Car.name= name ;    }  Public static void hello(){  S.o.p(“CarName:”+name);  } | * this.name= name will not work in case of * static variable so the below line * Class.variablename= passed variable(name) * hello method is a static method so in static method we directly can access static variable. Non static variable can’t be accessed from static method while static variable can be accessed in non-static method using classname.staticVariableName |
| <beans>     <bean id =”c” class=”Car”>        <property name=”name” value = “Audi”/>     </bean>  <beans> | The left side configuration will not work for static variable DI. This type of DI will work for dynamic variable. So DI for static variable is. |
| <beans>    <bean id =”c” class=”Car”>    <bean class=”MethodInvokingFactoryBean>     <property name=”StaticMethod”>          <value>Car.setName</value>    </property>     <property name=”arguments”>      <list>         <value>Audi</value>    </list>  </property>  </beans> | To inject static variable spring has given one support class. The class name is  **MethodInvokingFactoryBean**. This class will help in passing (injecting) static variable in static method. This a predefined class in spring. In this class they have given two setter methods  1-   setArgumenst(object[]  arguments)  This is used to pass static arguments  2-   setStaticMethod(String static methodName)  This is used configure static method name |

**Example of Static Variable DI**

**Use of Singleton Class and Factory class in spring application**

Let say if we have one singleton class and one factory class in our spring application then we must know how to create object of singleton classes and factory classes by using spring.

**What is the singleton first? -**Singleton means we need to insure that only one single instance (object) need to create throughout the entire application or main-stack (per main method). In short per main method only one object will create.

**Question:**So if we take any servlet web application then how many main method the web application will use.

**Answer:** Only one main method (stack or thread) servlet web application will use entire application.

**Question**: Do we have any singleton class in JRE or JDK

**Answer**: **ResourceBundle** class in **Java.util.ResourceBundle** package. This class is used for internationalization.

**Singleton method ResourceBundle**

So in ResourceBundle class if we try to create object using new keyword like given below.

|  |
| --- |
| **ResourceBundle rb= new ResourceBundle ();**    This ResourceBundle class constructor will not allow creating object using new keyword because this constructor is not accessible.    **Ques:**So how to create object of this Singleton class?  **Ans**: We have one static method **getBundle ()** inside ResourceBundle class. Using this static method we can create object of ResourceBundle class. So now the above code will like given below  **ResourceBundle rb = ResourceBundle.getBundle(“Messages”);**    Since static method can be accessed using **classname.staticMethod, we** don’t need to create object for calling static method. Here “Messages” is nothing but the **base name (without extentation)** of [**Messages.properties**] file.    **Note**: If we try to create another object using the above approach using static method and same property file.  **ResourceBundle rb2 = ResourceBundle.getBundle(“Messages”);**  **So**here it returns the same object and if we compare these two objects  **S.o.p(rb==rb1);**output**: true**  Then it will return true because **ResourceBundle**class is a singleton class and we will have only one object throughout the main-stack or application. So first time when it creates object, then the same object reference it assigns to other created objects. Hence we will have only one object.   So finally we can say that here **ResourceBundle.getBundle (“Messages”) internally create object of same class ResourceBundle.** |

**Question: Why ResourceBundle.getBundle (“Messages”) returns the same object every time?**

**Ans:**Whenever we try to load same property file [Messages.properties] multiple times (As we have created rb and rb2) then no use to create multiple objects for same property file having the same static content. It will consume Heap memory only.

So the **main purpose of using Singleton class** is:

1. Reduces number of instances by creating only one object under heap memory.
2. A singleton class is recommended if the class contains static content. Like Messages.properties file.
3. If singleton class contains dynamic content then singleton classes is of no use and if we try to make it singleton then one user value will be overwrite with another user value. So it is not recommended if singleton class contains dynamic content.

**Note:**Servlet classes are singleton or not?

Servlet class is not a singleton class. It does not have any private constructor internally but our Tomcat container make the servlet class as a singleton. So Tomcat container is a Factory class which makes servlet class as singleton. So factory classes can makes other classes as singleton.

**Use of factory-method attribute in spring**

**Note**: **Calendar class**: it is not possible to create Calendar class object with its constructor using new keyword because calendar class constructor is private and we cannot access it from outside of the calendar class. In java Calendar class provides one static method getInstance () which prepares calendar date and returning calendar

**But**as we know that in spring we can access the private constructor so in spring we can create Calendar class object with its constructor. But in spring also if try to access private constructor then we will get Illegal exception because private constructor does not prepares and return calendar date. . That is why we if we try to get calendar class object using constructor then we will not get any calendar class object until-unless we use getInstance() static method.

|  |  |
| --- | --- |
| Class Test{     private Test(){      }      }    This class has one user defined private constructor. | <bean id = “t”  class= “Test”>  In the above <bean> tag Spring will create user defined private constructor class object because here user is not throwing any exception. In case if user throws any exception then spring will not able create Test class object.  **But in case of Calendar class.**  < bean id =”c”  class=”Calendar”>  Here in spring if try to create object of Calendar class with its private constructor then we will get illegal exception. Because Java throws Illegals exceptions inside private Calendar constructor. |
| Class Test{     private Test(){  throw new IllegalException();      } } | If in used defined Test class we want that spring also cannot access private constructor then we have to throw exception inside private constructor of the Test class. So the above Test class will be like |

**Question: So now the question arises how to create Calendar class object in spring?**

Calendar is not a singleton, and each call to Calendar.getInstance (...) returns a different instance. The Javadoc doesn't say that each call will return the same instance, so you have no reason to assume that it will. Calendar.getInstance (...) better fits the factory design pattern.

Here getInstance () is a factory method.

So in spring also we will use factory-method getInstance () to create object of Calendar class.

So finally the xml (cal.xml) spring configuration will be as given below.

|  |
| --- |
| <beans>     <bean id =”cal”  class= “java.Util.Calendar”  factory-method=”getInstance”/>  <beans> |

Here whenever we load this xml into IOC then IOC will understand that we have to create object of Calendar class using factory method getInstance instead of using  its private constructor Calendar(). So in the main method to get the calendar classes object we just need to pass bean id “c”.

|  |
| --- |
| ApplicationContext ap = **new** ClassPathXmlApplicationContext ("resources/cal.xml");  Calendar cobj= (Calendar) ap.getBean ("cal"); |

So in case if want to create any class (Calendar) object using its factory (getInstance) in spring then we will create by using **factory-method** attribute.

**Factory Class Vs Factory Method:**

Factory classes returns other class objects instead of returning same class objects. Example

|  |  |
| --- | --- |
| Logger  log = LoggerFactory.getLogger(this.class);    Here getLogger is static method | Here LoggerFactory is a factory class and it is returning Logger class object instead of returning LoggerFactory class object |
| Validator  v =ValidatorFactory.getValidation();    Here getValidation is static method | In hibernate -ValidatorFactory is a factory class and it is returning Validator class object instead of returning ValidatorFactory class object |
| Session ses= SessionFactory.openSession();    Here openSession is instance method(Need to check this only) | In hibernate - SessionFactory is a Instance class and it is returning Session class object instead of returning SessionFactory class object |

**While Factory Method** returns the object of same class. Like in Calendar class getInstance () returns Calendar class object only . So the corresponding object of Logger class using FactoryClass in spring is

So finally the xml (factory.xml) spring configuration will be as given below.

|  |
| --- |
| <beans>     <bean id =”logger”  class= “**LoggerFactory**”  factory-method=” **getLogger**”/>     <bean id =”v”  class= “**ValidatorFactory**”  factory-method=” **getValidation**”/>  <!—For instance class **SessionFactory -- >**     <bean id =”sf”  class= “**SessionFactory**” />     <bean id =”s”  factory-bean= “sf”  factory-method=” **openSession**”/>  <beans> |
| And in main method of java. **Need to test**    ApplicationContext ap = **new** ClassPathXmlApplicationContext ("resources/factory.xml");  Logger lobj= (Logger) ap.getBean ("logger");  Validator  vobj= (Validator) ap.getBean ("v");  Session  sobj= (Session )ap.getBean ("s"); |

**Example of creating singleton class.**

**How to write Singleton Classes:**

1. Using factory method we can create singleton classes because factory method (static method) of singleton classes (e.g. **ResourceBundle**) returns the same class instances. But the static method of Factory classes (e.g.**LoggerFactory)** returns other class instances.
2. A singleton class must have a private constructor.
3. A singleton class must have one static method (like getInstance () which will create same class object and this method should be public so that it can be accessible outside of the class.
4. A singleton class must have one same class static reference variable as null and it has to instantiated in static method getInstance ().

|  |  |
| --- | --- |
| // File Name: Singleton.java  public class Singleton {     private static Singleton singleton = new Singleton( );       /\* A private Constructor prevents any other      \* class from instantiating.      \*/     private Singleton() {  System.out.println("Enters into private construtor");  }     /\* Static 'instance' method \*/     public static Singleton getSingletonInstance( ) {  System.out.println("Enters into getInstance static method");     return singleton;     }     /\* Other methods protected by singleton-ness \*/     protected static void demoMethod( ) {        System.out.println("demoMethod for singleton");     }  }  // File Name: SingletonDemo.java  public class SingletonDemo {       public static void main(String[] args) {  Singleton tmp = Singleton. getSingletonInstance ( );        tmp.demoMethod( );     }  } | Example 1  1-   Here class consists of a private constructor  2-   Same class static reference variable,  3-   And a public static accessible method with a name like getInstance ().  The private field can be assigned from within a static initializer block or, more simply, using an initializer like getInstance. The getInstance( ) method (which must be public) then simply returns this instance –  ASA we call **Singleton.getSingletonInstance** (). Immediately it will call the default private constructor and then will enter into getInstance() method finally in demoMethod()  This will produce the following result −  Output    Enters inot private construtor  Enters inot getInstance static method  demoMethod for singleton |
| public class **ClassicSingleton** {       private static ClassicSingleton instance = null;     private ClassicSingleton() {        // Exists only to defeat instantiation.     }  public static ClassicSingleton getClassInstance() {        if(instance == null) {           instance = new ClassicSingleton();        }else        return instance;     }  } | Example 2  1-      Following implementation shows a classic Singleton design pattern  2-      In class we have one static reference of the same class type which is being return from the static method getInstance()  3-      Here, ClassicSingleton class technique known is as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed. |

**Creating object of above singleton classes in spring IOC using factory-method attribute:**

|  |
| --- |
| **For Example 1- Singleton class**    **<beans>**  **<**bean**id =”**single**” class= “**Singleton” **factory-method**=”**getSingletonInstance**”/>  **</besns>**    **Now load the above xml into IOC**    ApplicationContext ap = **new** ClassPathXmlApplicationContext ("resources/singlet.xml");  Singleton lobj= (Singleton) ap.getBean ("single"); |
| **For Example 2-**ClassicSingleton**class**    **<beans>**  **<bean id =”**CS**” class= “**ClassicSingleton” **factory-method=**”getClassInstance”/>  **</besns>**    ApplicationContext ap = **new** ClassPathXmlApplicationContext ("resources/factory.xml");  ClassicSingleton lobj=(ClassicSingleton)ap.getBean("CS"); |

**Note:**As we know that spring can access private constructor, so if want that spring also could not access private constructor then we need to throw one exception from private constructor.

**throw new IllegealException ();**

**Factory Class:**

1. **Hide instantiation logic**
2. **By returning object into interface reference factory will make client independent.**

|  |  |  |  |
| --- | --- | --- | --- |
| **First create one Interface**      public interface Shape {     void draw();  } | **Second Create Implementation of Interface**  Rectangle.java  public class Rectangle implements Shape {       @Override     public void draw() {     S.o.p ("Inside Rectangle::draw() method.");     }  }  Square.java  public class Square **implements** Shape {       @Override     public void draw() {  S.o.p(“Inside Square::draw() method.");     }  }  Circle.java  public class Circle implements Shape {       @Override     public void draw() {  S.o.p("Inside Circle::draw() method.");     }  } | **Create a Factory to generate object of implementation class**  ShapeFactory.java    public class ShapeFactory {       //use getShape method to get object of type shape     public Shape getShape(String shapeType){        if(shapeType == null){           return null;        }        if(shapeType.equalsIgnoreCase("CIRCLE")){           return new Circle();          } elseif(shapeType.equalsIgnoreCase("RECTANGLE")){           return new Rectangle();          } elseif(shapeType.equalsIgnoreCase("SQUARE")){           return new Square();        }          return null;     }  } | |
| FactoryPatternDemo.java  public class FactoryPatternDemo {       public static void main(String[] args) {     ShapeFactory shapeFactory = new ShapeFactory();    //get an object of Circle and call its draw method.        Shape shape1 = shapeFactory.getShape("CIRCLE");          //call draw method of Circle        shape1.draw();    //get an object of Rectangle and call its draw method. | | Shape shape2  =shapeFactory.getShape("**RECTANGLE");**      //call draw method of Rectangle        shape2.draw();    //get an object of Square and call its draw method.  Shape shape3  = shapeFactory.getShape("SQUARE");          //call draw method of square        shape3.draw();     }  } | |
|  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Create Connection  Interface:**    **Public Interface Connection{**  **Void getDBConnection();**  **}** | **Create implementation of connection Interface**    Public MySql implements **Connection{**  **Public void getDBConnection (){**  **S.o.p(“MySql Connected”);**  **}**  Public Oracle implements **Connection{**  **Public void getDBConnection (){**  **S.o.p(“Oracle Connected”);**  **}**  Public DB2 implements **Connection{**  **Public void getDBConnection (){**  **S.o.p(“**DB2 **Connected”);**  **}**  Public MicrosoftSql implements **Connection{**  **Public void getDBConnection (){**  **S.o.p(“**MicrosoftSql **Connected”);**  **}** | **Create Factory class to generate object of implementation class**  Here inside DriverManager class we will create method of Interface Connection type which will return object of implantation class in the form of Interface Connection.  Public class **DriverManager**{    Public Connection getConnection(String **conTyp){**    **If(conTyp==null){**  **return null;**  **}**  **If(conTyp.** equalsIgnoreCase(“**MySql**){          return new MySql();        }  **If(conTyp.** equalsIgnoreCase(“**Oracle**){          return new Oracle();        }  **If(conTyp.** equalsIgnoreCase(“**DB2**){          return new DB2();        }  **If(conTyp.** equalsIgnoreCase(“**MSSql**){          return new MicrosoftSql();        }  } |
| public class FactoryPatternDemo {       public static void main(String[] args) {    **DriverManager DriverManager= new DriverManager();**     Connection MysqlCon= **DriverManager.getConnection(“MySql”);**  MysqlCon.**getDBConnection();**     Connection mysqlCon= **DriverManager.getConnection(“MySql”);**     Connection mysqlCon= **DriverManager.getConnection(“MySql”);**     Connection mysqlCon= **DriverManager.getConnection(“MySql”);**  } | | |

**Example of Factory Class creation using [Class.forName (carname).newInstance ()]**

So for creating one factory class the fallowing steps we need to fallow.

**Steps to create factory Class**

1. First of all create one Interface and inside the interface create one or several method prototype according to our requirements.
2. Create one factory class and inside this factory class create the fallowing things.

* Create one default constructor
* One Static method of Interface type which will return other class object in the form of Interface reference.

1. Now create the implementation classes by implementing created Interface according to our requirements.
2. Inside each implementation class Override the method declared in the Interface and writes the logic in the method according to our requirements.
3. Finally create one Client Class and inside the main method of the client class will call the factory class method to get the implemented class objects by passing its name (Class name).
4. Hence factory method will provide the object of other classes(implemented Class object) and finally using this object (implemented Class object) we can call the override method of (implemented Class)

|  |  |
| --- | --- |
| **Step1**- Create One Interface first having one method prototype to be implemented in its implementation classes.  **package** factory.demo.interfac;  **public** **interface** Car {  **public** **void** drive();;  } | |
| **Step2**- Create one factory class having One Static method of Interface type which is returning Interface reference.  **package** factory.main.classs;  **import** factory.demo.interfac.Car;  **public** **class** CarFactory {  **private** CarFactory() {} // Private constructor  **public** **static** Car getCar(String carname) {  System.***out***.println("Car Class Name :"+carname);  Car c= **(Car)Class.*forName*(carname).newInstance();**  System.***out***.println("Returned Class Object :"+c);  **return** c;         }  }  **Note for Step2:**  **(Car)Class.*forName*(carname).newInstance()**returns the passed class object into Interface reference  /\* **c** Will return the Car Interface implemented class object based on the user input(Class Name)  Factory class returns Car Interface implementation class object based on the user input object instead of returning CarFactory class object.\*/ | |
| **Step3- Create implementation class of Interface Car- Benz.java**  **package** factory.implementation.classes;  **import** factory.demo.interfac.Car;    **public** **class** Benz **implements** Car {    @Override  **public** **void** drive() {  System.***out***.println("Drive Benz Car");         }  } | **Honda.java**  **package** factory.implementation.classes;  **import** factory.demo.interfac.Car;    **public** **class** Honda **implements** Car{    @Override  **public** **void** drive() {  System.***out***.println("Drive Honda Car");         }  } |

**Step4**- Finally creates one Client Class and inside the main method of the client class will call the factory class method to get the implemented class objects by passing its name (Class name).

|  |
| --- |
| package factory.demo.client;    import factory.demo.interfac.Car;  import factory.main.classs.CarFactory;  import factory.implementation.classes.Benz;    public class CarClient {  public static void main(String[] args) {    // Since in Factory class it returns object of other class in the form of Interface reference so                          Car  c1 = CarFactory.getCar("factory.implementation.classes.Swift");                          System.out.println("Current class reference :"+c1.toString());                          c1.drive();                          System.out.println("\n-----------------------------------------");                          Car  c2= CarFactory.getCar("factory.implementation.classes.Benz");                          System.out.println("Current class reference :"+c2.toString());                          c2.drive();                          System.out.println("\n-----------------------------------------");                          Car  c3 = CarFactory.getCar("factory.implementation.classes.Honda");                          System.out.println("Current class reference :"+c3.toString());                          c3.drive();              }  }  Here C1 will have Swift Car class object so c1 will call Swift Car class drive() method  Here C2 will have Benz Car class object so c1 will call Benz Car class drive() method  Here C3 will have Honda Car class object so c1 will call Honda Car class drive() method |
| **Output**:  Car Class Name :factory.implementation.classes.Swift  Returned Class Object  :factory.implementation.classes.Swift@a5d4bcc5  Current class reference :factory.implementation.classes.Swift@a5d4bcc5  Have safe and happy journey with Swift    -----------------------------------------  Car Class Name :factory.implementation.classes.Benz  Returned Class Object  :factory.implementation.classes.Benz@b7e1b0c7  Current class reference :factory.implementation.classes.Benz@b7e1b0c7  Drive Benz Car    -----------------------------------------  Car Class Name :factory.implementation.classes.Honda  Returned Class Object  :factory.implementation.classes.Honda@a19d3887  Current class reference :factory.implementation.classes.Honda@a19d3887  Drive Honda Car |

So here in Factory class implementation we are hiding the actual implementation from the client. Client just need to Import Factory class and need to call the static method and need to pass his input as an argument. This Interface (Car) and Factory Class (CarFactory) collectively supposed to give from Car Vender and client just need to use.

**Creating Factory class object in spring:**

As we know that factory class design requires interface first then we will design Factory class. Implementation classes we can ask from other vendors also. A factory class can have static method as well as instance method both.

**Case1- When the factory class has static method.**

|  |  |
| --- | --- |
| **package** factory.main.classs;  **import** factory.demo.interfac.Car;    **public** **class** CarFactory {  **private** CarFactory() {} // Private constructor    **public** **static** Car getCar(String carname) {  System.***out***.println("Car Class Name :"+carname);  Car c= **(Car)Class.*forName*(carname).newInstance();**  System.***out***.println("Returned Class Object :"+c);  **return** c;         }  }  Here in simple java we are getting value (carname) through parameter. But in case of spring we need to inject this from IOC container and for injecting class must have ether setter method of parameterized constructor. So | **package** factory.main.classs;  **import** factory.demo.interfac.Car;    **public** **class** CarFactory {  **private** CarFactory() {} // Private constructor    **public** **static** Car getCar() {  private static String carname; // Static variable  setCarname(String carname){} // setter method  System.***out***.println("Car Class Name :"+carname);  Car c= **(Car)Class.*forName*(carname).newInstance();**  System.***out***.println("Returned Class Object :"+c);  **return** c;         }  }  **Benz.java implementation class**  // We should have at least one implementation class. Refer to above any of implantations class.**Benz.java** |

Now the corresponding bean.xml (spring configuration file) will be

|  |
| --- |
| <!--First we need to inject static variables. Then will create object of Factory Class-- >  <beans>       <bean MethodInvokingFactoryBean>               <property name=”Static-metbod” value =”CarFactory. SetCarname”/>                   <property name=”argument”>                        <list>                             <value>**factory.implementation.classes.Benz**</value>                        </list>                   </property>          </bean>          <! – Creating object of factory Class  having static method getCar-->          <bean id =”fc”   class =” CarFactory “ factory-method=”getCar”/>           <!—Here getCar will create one instance reference which we can get from id(fc) -- >  </beans> |
|  |
|  |

|  |
| --- |
| Load the bean.xml into IOC for getting Car Interface object.  **ApplicationContext ap = new ClassPathXmlApplicationContext("resources/bean.xml");**  **Car c =(Car)ap.getBean("cf");** |
| In the above code we are passing CarFactory reference for creating implementation class (Benz) object. In the above xml file we are injecting full qualified path of Implemented Benz class which will be injecting into CarFactory class using setter method (setCarname).  Hence using **Car c =(Car)ap.getBean("cf");**we will get Benz car class object into Interface Car c.Now using c we can call method of Benz car class  **c.drive();**  **Similarly by injecting other implemented class name under <list> tag we can get object of these classes also** |

**Case2: When the factory class does not have static method (Instance Factory Object).**

|  |  |
| --- | --- |
| **package** factory.main.classs;  **import** factory.demo.interfac.Car;  **publicclass** CarFactory {  **private** CarFactory() {} // Private constructor    **public** Car getCar() { // not static method  private String carname; // non Static variable  setCarname(String carname){} // setter method  System.***out***.println("Car Class Name :"+carname);  Car c= **(Car)Class.*forName*(carname).newInstance();**  System.***out***.println("Returned Class Object :"+c);  **return**c;  }  }  **Benz.java implementation class**  // We should have at least one implementation class. Refer to above any of implantations class. **Benz.java** | 1. Here we have one not static variable carname and non-static method getCar. 2. Here DI will do like normal setter based DI as it does not contain any static variable. 3. In the below xml file we will get CarFactory class object (cf) instead of getting implemented class (Benz). 4. Now to get Implemented Class object (Benz Class object) we need to call getCar method defined in the CarFactory class. 5. As we can see that we have another bean where we have one attribute factory-bean which takes CarFactory class object as reference and another attribute factory-method for calling not static method of CarFactory class. And finally id=”c”of this bean will be used for getting object of implemented class. As shown in the below code. |
| <beans>  <bean id =”cf” class=”CarFactory”>  <!—Here we will get CarFactory class object only because we don’t have any static method in CarFactory class as well as we don’t have static DI in xml. So to get implemented class object we will have to pass CarFactory id =”cf” as shown below -- >  <property name=”carname” value=” **factory.implementation.classes.Benz**”>  </bean>  <bean id =”c” factory-bean=”cf” factory-metbod=”getCar”/>  </beans>  Now Loading the above xml into IOC container  **ApplicationContext ap = new ClassPathXmlApplicationContext("resources/bean.xml");**  **Car cobj= (Car) ap.getBean ("c"); // Will return Benz class object.**  **cobj.drive ();// Calling Benz class method drive.** | |

Note: Let say if we have another implementation class (Honda.java) so to get object of Honda class we just need to pass Class Name(Full qualified name of class-**factory.implementation.classes.Honda**) under CarFactory bean tag and it will reflect in main client class automatically.

So finally we can say that the factory method does the fallowing things.

1. It provides the other class object. So in future if we have any other implementation class then for this class object we can create using same factory method.
2. It hides instantiation logic internally and finally it gives object of other class into Interface reference.
3. DriverManager .getConnection method will return object into connection interface. But the returned object belongs to implementation vendors (like MySQL, DB2, MSSQL, etc.)
4. We are making our client independent itdoesn’t need to dependent on particular vendor.
5. Now the same kind of factory objects we will create using spring framework.
6. It behaves like middle ware class who creates other class objects.

**Factory Class Object creation in spring using FactoryBean interface:**

* In spring they have given on interface [FactoryBean (I)]. By implementing this interface we can implement factory class.
* This FactoryBean Interface contains three abstract methods

|  |  |
| --- | --- |
| 1. **getObject() –** | This returns object. |
| 1. **getObjectType()-** | This returns class. |
| 1. **isSingleton()-** | This returns boolean |

**Note**: As we know that in simple java we had two approaches to created Factory class object. First one is when Factory class contains static method and from this static method we had to return implementation class objects. Second one when Factory class contains non static method means class contains instance method then from this instance method we need to return implementation class objects.

But here in case of spring we need to implement all three abstract method of FactoryBean interface. Using the same above Car (I) and CarFactory(c) to create object of Factory class object in spring. So

|  |  |
| --- | --- |
| Interface Car{  Public void drive();  } | Class CarFactory implements FactoryBean{   1. 1- public Object **getObject** () throws Exceptions{ } 2. // in the above method we will create object of Car(I) implementation class object [using **Class.forName(Implementation Class Name).newInstance**] and return the same object to the Car Interface reference. 3. 2- public Class **getObjectType** () throws Exceptions{ } 4. // Above method will return class Type (using Car.class)   public Boolean isSingleton(){ }   1. // If we want to make our Car implementation class object as singleton then here we just need to (return true) |

So the corresponding code for CarFactory class implementing FactoryBean interface is given below

|  |  |
| --- | --- |
| Class CarFactory implements FactoryBean{  Public string carname; // Input for DI  // setter of carname  Public String setCarname(String carname){  this.carname = carname;  }  public Object **getObject** () throws Exceptions{  Car c = (Car)Class.forName(carname).newInstance();  return c ;  }  public Class **getObjectType** () throws Exceptions{  return Car.class;  }  public Boolean **isSingleton**(){  return true;  } | **Honda.java**  **package** factory.implementation.classes;  **import** factory.demo.interfac.Car;  **publicclass** Honda **implements** Car{    @Override  **publicvoid** drive() {  System.***out***.println("Drive Honda Car");  }  }  **Note**: If we want to create object of Car (I) implementation Honda class object then we need to have capability to take input in the CarFactory class and here in case of spring we cannot take input through method getObject like we had done in case of simple java approach.  So we will create one String carname and using setter DI we will inject implementation class names in the CarFactory class. |
| Now the corresponding bean xml will be:  **<beans>**  **<bean id = “cf” class=“CarFactory”>**  **<property name=“carname” value=“Honda”>**  **</bean>**  **</beans>**  It is looking simple Setter Based DI Only | |
| Now load the above xml into IOC and passing id =cf into getBean method will give the object of Honda class object.  **Car c = (Car)getBean(“cf”);**  Here we will get Object of Car (I) implementation class object (Honda class object) | |

So to implement factory in spring we have three approaches.

1. Using Static Factory.
2. Using Instance Factory.
3. Using FactoryBean interface.

In Case of 1 and 2 here we no need to implement any interface like FactoryBean. In case 1 we just need to have one static method and in case 2 we just need to have one instance method.

**Example from videos num26-**

**Case: Where we should use Factory Class implementation in spring:**

**Diagram of Factory:**

|  |
| --- |
|  |

**As we know that** in spring we have several layers like [1- Presentation Layer 2- Controller Class layer 3- Business class layer 4- DAO layer and finally 5- Connection Pool or session factory or Ibatis factory.

So while two layers communicate with each other, if we put factory class in-between them then these two layers becomes loosely coupled and both of the layers becomes independent to each other. So here in spring MVC application we can use factory between Controller and Business, between business and DAO and between DAO and connection pool. So the main need of factory is to make two layers independent and it will hide implementation logic of RHS class from LHS class. Like here Business Layer implementation logic will be hiding for Controller developer and DAO layer implementation logic will be hiding for Business layer developer.

|  |
| --- |
| Diag from vedio 27 |

**Spring Bean class Life Cycle:**

**As we know that** in spring we have several layers like [1- Presentation Layer 2- Controller Class layer 3- Business class layer 4- DAO layer and finally 5- Connection Pool or session factory or Ibatis factory.

Now in case if controller requires any life cycle method, if business requires any lifecycle method or if DAO class requires any life cycle method then we can implement lifecycle method for these classes.

As we know in Servlet we have life cycle method:init, service and destroy and to implement any servlet class we need to implement Servlet interface or need to extends generic servlet or Http servlet but in case of Spring, here in for implementation of Controller class or Business Class or DAO class we don’t need to implement or extend any interface or class, just POJO classes are enough to handle Controller, Business and DAO classes. So in the POJO classes if want to maintain any lifecycle method then these life cycle methods can be maintain by using IOC container ,in other words by using IOC we can maintain lifecycle of controller , business and DAO classes.

But question arises that which IOC container we need to use here of maintaining lifecycle.

So ConfigurableApplicationContext IOC container (j2ee container) will be used for managing lifecycle of POJO classes.

Note: Servlet lifecycle will be managed by servlet/ tomcat container and the servlet lifecycle method will be called when we configure a servlet on “Load on startup”. So as soon as we deploy the servlet into server , the servlet object will create on load on startup of the server and after that life cycle method init() will be called and destroy method will be called during re-deployment or unemployment of the servlet from the server

In the same way in spring, POJO classes contain init and destroy method which will be maintained by ConfigurableApplicationContext.

**Question**: **When these init and destroy method of POJO classes will be called?**

**Answer:** Like servlet whose object gets created on Load-on-startup and after creation of object servlet lifecycle init method gets called. In the same way when the spring configuration file (spring.xml) having all the configuration of Controller , Business and DAO classes will be loaded into IOC container then during loading on spring.xml ,ConfigurableApplicationContext IOC container will create object of all the bean classes (Controller Classes , Business Classes and DAO Classes). After creation of all bean classes’ objects if bean classes contain init method then this init method will be executed.

Since init and destroy method gets executed only once in the whole life cycle so in init and destroy method we will write the code which has to be executed only once in whole life cycle.

**For example DAO class** responsibility is to get connection from database and using this connection DAO has to do database operations and finally when all the operation gets completed then DAO has to return back this connection to database. So in init method we will write code for openConnection.

After getting connection we will perform multiple operations like (save, update, delete, find, findAll). After performing these operation we need to close openConnection object into destroy method of POJO lifecycle. So init method is used for initialization and destroy method is used for cleanup operations.

**Coming to Business Class: In**this, let’s say we have multiple operation like (CreateStudent,UpdateStudent,DeleteStudent, FindSutdent and findAll) and while executing these operation we want to have some loggings messagethen to get loggings, the logger object we can create under init method and we can clean logger object into destroy method of business POJO class and logger object can be used within the operation. Same thing we can do for Controller class also.

**There are three approaches for bean life cycle method in spring.**

1. Programitive approach.
2. Declarative XML approach.
3. Annotation approach.

**Programitive Approach:** We will have to fallow the fallowing steps if we go with this approach.

1. POJO or Bean classes have to implement two interfaces, InitilizationBean (I) and DisposableBean (I).
2. POJO Classes has to override two abstract method [afterPropertiesSet] declared in InitilizationBean (I) and [destroy] declared in DisposableBean.

**Declarative XML approach:**

1. In this approach, inside the POJO or Bean class we can write any user-defined method name like **myInit**() for initialization purpose and **myDestroy**() for cleanup purpose.
2. This user-defined method name we just need to configure under xml file while creating our bean object under <bean> tag.

**<bean id = “obj” class = “PojoClass” init-method =”myInit” destroy-method= “myDestroy”>**

1. We don’t need to implement any interface and we don’t need to override any abstract method.

**Annotation Approach:** This is similar to xml approach with little difference.

1. In this approach, inside the POJO or Bean class we can write any user-defined method name like **myInit**() for initialization purpose and **myDestroy**() for cleanup purpose.
2. On top of this user-defined method we just need to declare annotation.
3. **@PostConstruct** will be declared for myInit () and **@PreDestroy** will be declared for myDestroy () methods.
4. **Used annotation class name: Need to write.**

Note: Here in ConfigurableApplicationContext we have one close method which will be called for calling destroy method of bean lifecycle. i.e. whenever we call the close() method it will call the destroy method.

**Spring Bean Life Cycle Call Back Method:**

Spring provides the call back methods for Spring Bean lifecycle.

//**ApplicationContext context = new ClasspathXmlApplicationContext ("BeanInheritance.xml");**

**AbstractApplicationContext context = new ClasspathXmlApplicationContext ("SpringBeanLifeCycle.xml");**

**context.registerShutdownHook ();**

**Triangle triangle= (Triangle) context.getBean ("triangle");**

As we know that ApplicationContext loads the entire bean from the xml where we configure the entire bean. As we also know that in Web Application Spring automatically takes care for shutting down the application we don’t need to specify anything but in desktop application we need to register a method for shutting down the application . This method found in AbstractApplicationContext Interface. So as we can see that now we have used AbstractApplicationContext for loading the entire bean from the xml file.

And after that we have register method using AbstractApplicationContext object as given below.

**context.registerShutdownHook ();**

Now after registering this method the destroy method would be called at the end of the bean life cycle and will close the application.

|  |
| --- |
| **package**com.cts.spring.test;  **import**org.springframework.beans.factory.InitializingBean;  **import**org.springframework.beans.factory.DisposableBean;  **import**org.springframework.context.ApplicationContext;  **publicclass** Triangle **implements**InitializingBean,DisposableBean{    **private** Point pointA;  **private** Point pointB;  **private** Point pointC;  ApplicationContext context=**null**;    **public** Point getPointA() {  **return**pointA;  }  **publicvoid**setPointA(Point pointA) {  **this**.pointA = pointA;  }  **public** Point getPointB() {  **return**pointB;  }  **publicvoid**setPointB(Point pointB) {  **this**.pointB = pointB;  }  **public** Point getPointC() {  **return**pointC;  }  **publicvoid**setPointC(Point pointC) {  **this**.pointC = pointC;  }  **publicvoid** show()  {  System.*out*.println("Coordinate A ("+getPointA().getX() +","+getPointA().getY()+")");  System.*out*.println("Coordinate B ("+getPointB().getX()+","+getPointB().getY()+")");  System.*out*.println("Coordinate C ("+getPointC().getX()+","+getPointC().getY()+")");  }  @Override  **publicvoid**afterPropertiesSet() **throws** Exception {  System.*out*.println(" The InitializingBean method of bean gets called !!! }  @Override  **publicvoid** destroy() **throws** Exception {  System.*out*.println("The destroy method gets called...");  }  } |

Note:1- Here we have implemented the Interface **InitializingBean**.So as soon as we implement this Interface InitializingBean then it override one method which gets called when <bean> gets initialized and this method would be called before all the beans getting loaded using ApplicationCotext.The Method that would be override when implementing InitializingBean is given below .

@Override

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

System.*out*.println (" The InitializingBean method of bean gets called!!!!");

}

Note:2- Here we have implemented the Interface **DisposableBean**.So as soon as we implement this Interface DisposableBean then it override one method which gets called when Application is going to shut down at the end.The Method that would be override when implementing InitializingBean is given below.

@Override

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

System.*out*.println ("The destroy method gets called...");

}

**Note3:-** This is one of the ways to implement the <bean> lifecycle method. Here we have done fallowing things as given below.

(Step1 :-) Implements (Inherit) Two Interface InitializingBean (For Initializing the bean) ,DisposableBean(Destroying the Bean).

**(Step2 :-)** As soon as we Implement these two methods then we then it ask to override **afterPropertiesSet ()** (In case of InitializingBean) and **destroy ()** (In case of DisposableBean). And for destroy () method to be called we need to register one more method after **AbstractApplicationContext**getting called. As given below.

**AbstractApplicationContext context = new ClassPathXmlApplicationContext ("SpringBeanLifeCycle.xml");**

context.registerShutdownHook ();

Triangle triangle = (Triangle) context.getBean ("triangle");

Another Way: Configuring in the XML file for lifecycle method init() and destroy(). Here we will not be implementing any interfaces.

Step1-: Here we will write simple method of init() and destroy same as concreate method .

**publicvoid**myInit(){

System.*out*.println(" The Init method of bean gets called !!!!");

}

**publicvoid**myDestroy(){

System.*out*.println(" The destroy method of bean gets called !!!!");

}

Step2-: Now configure these method name for initMethod(**init-method=*"myInit") and for destroy(***destroy-method=*"myDestroy")* in the xml configuration file as given below

<beanid=*"triangle"*class=*"com.cts.spring.test.Triangle"*autowire=*"byName"***init-method=*"myInit"*destroy-method=*"myDestroy"***>

</bean>

**Now from main () calling.**

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

**AbstractApplicationContext context = new ClassPathXmlApplicationContext("ApplicationContextAware.xml");**

**context.registerShutdownHook();**

Triangle triangle=(Triangle)context.getBean("triangle");

triangle.show();

}

**Output:**

The Init method of bean gets called !!!!

Coordinate A (0,0)

Coordinate B (-20,-40)

Coordinate C (100,200)

The destroy method of bean gets called !!!!

So Finally we have

**(1) Triangle Class:**

**package**com.cts.spring.test;

**publicclass** Triangle {

**private** Point pointA;

**private** Point pointB;

**private** Point pointC;

**public** Point getPointA() {

**return**pointA;

}

**publicvoid**setPointA(Point pointA) {

**this**.pointA = pointA;

}

**public** Point getPointB() {

**return**pointB;

}

**publicvoid**setPointB(Point pointB) {

**this**.pointB = pointB;

}

**public** Point getPointC() {

**return**pointC;

}

**publicvoid**setPointC(Point pointC) {

**this**.pointC = pointC;

}

**publicvoid** show()

{

System.*out*.println("Coordinate A ("+getPointA().getX() +","+getPointA().getY()+")");

System.*out*.println("Coordinate B ("+getPointB().getX()+","+getPointB().getY()+")");

System.*out*.println("Coordinate C ("+getPointC().getX()+","+getPointC().getY()+")");

}

**publicvoid**myInit(){

System.*out*.println(" The Init method of bean gets called !!!!");

}

**publicvoid**myDestroy(){

System.*out*.println(" The destroy method of bean gets called !!!!");

}

}

**(2) Xml Configuration File:**

|  |
| --- |
| <?xmlversion=*"1.0"*encoding=*"UTF-8"*?>  <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <beanid=*"triangle"*class=*"com.cts.spring.test.Triangle"*autowire=*"byName"*init-method=*"myInit"*destroy-method=*"myDestroy"*>  </bean>  <beanid=*"pointA"*class=*"com.cts.spring.test.Point"*>  <propertyname=*"x"*value=*"0"*></property>  <propertyname=*"y"*value=*"0"*></property>  </bean>  <beanid=*"pointB"*class=*"com.cts.spring.test.Point"*>  <propertyname=*"x"*value=*"-20"*></property>  <propertyname=*"y"*value=*"-40"*></property>  </bean>  <beanid=*"pointC"*class=*"com.cts.spring.test.Point"*>  <propertyname=*"x"*value=*"100"*></property>  <propertyname=*"y"*value=*"200"*></property>  </bean>  </beans> |

(3) Main() Method

**package**com.cts.spring.test;

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

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

**publicclass**TriangleController {

**publicstaticvoid** main(String[] args) {

AbstractApplicationContext context = **newClassPathXmlApplicationContext ("ApplicationContextAware.xml");**

**context.registerShutdownHook ();**

Triangle triangle = (Triangle) context.getBean ("triangle");

triangle.show();

}

}

Note: We can configure our own init() and destroy() method on the global bean using (default-init-method="myInit" default-destroy-method="myDestroy")

**Example of Bean life Cycle methods:**

**Lookup Method Dependency Injection:**

As till now we know only two type of DI (Setter method DI and Parameterized constructor DI). Apart from these two we have one more DI called “Lookup Method DI”. So finally we have three types of DI.

1. Setter method DI
2. Constructor DI
3. Lookup method DI

Q: What is lookup method?

A: If a method does not have any implementation or if a method requires any dependencies then we can consider those method as Lookup methods.

**Case1**- In any interface we declared the method prototype only and all the declared method inside Interface are abstract method by-default. So these methods are said to lookup methods because these methods do not have any implementation inside interface.

|  |  |
| --- | --- |
| Interface Test{  Public void A();  Public void B();  } | Here inside Test interface we have two unimplemented methods looking for implementation into its implementation classes so Methods A and B are called as Lookup Methods. So all the abstract methods inside interface are called lookup methods. |

**Case2**- In any abstract class we may have abstract method (only method declaration) as well as concrete methods (completed implemented method). So these abstract methods inside abstract classes are looking for implementation so these methods are called lookup methods

|  |  |
| --- | --- |
| abstract class Test{  abstract Public void A();  Public void B(){  S.o.p(“Hello”);  } | Here inside abstract Test class we have two methods A is an abstract method looking for implementation into its implementation classes another is complete method so Methods A is called as Lookup Methods. |

**Case3**- Example 3rd: Let’s say we have one concrete class and we have methods with body with full implementation and we want to change implementation of these methods into concrete class. In other words we want to override this method from its child class for different functionality then this method is called lookup method

|  |  |
| --- | --- |
| class Test{  Public void A(){  //have implementation  }  Public void B(){  //have implementation  } | Here inside concrete Test class we have two methods A and B and if want to change functionality with some other functionality by overriding A and B then A and B both are said to be lookup methods because these two methods looking for new implementation by overriding. |

**Note**: In case if Interface all the abstract method or lookup methods compulsory needs to be implemented into its implementation class. In case of abstract class if class contains any abstract methods then this abstract methods or look up compulsory need to be implemented into its implementation class. In case of concrete class if we want to change any implemented method then it must be override to be implemented to make previous method as lookup method.

**Question**: So the question is by using this lookup method DI concept what we can provide?

**Answer**: The answer is we can provide implementation or we can do overriding operation from its child classes.

**Question:** In spring how will do implementation for lookup methods?

**Answer**: As in normal java application lookup method implementation is done into its implementation class or into its child class (in case of overriding) but in spring, lookup method implementation will be done by using dependency injection.

So to provide lookup method DI spring will provide implementation classes. Spring will provide some implementation classes at run time to provide implementation to lookup methods.

As till now we have done two type of DI (Setter DI and constructor DI) and here we used to create object of class using bean id and we used to inject some values into setter method or in constructor from bean xml file. But here in spring lookup DI, spring injects the full method with code.

**Question:** It is not possible to inject full method with its code from the xml file then how the spring does lookup method DI. How the spring identifies the return type of lookup method which has to be implemented.

**Answer:**

Let’s understand with example. Let us consider we have one abstract class Car and inside this class we have one abstract lookup method (myEngine) whose return type is secondary type (Engine).

|  |  |
| --- | --- |
| abstract class Car{  abstract public Engine myEngine();  } | Here lookup method **myEngine**return type is Engine type so if we implement this method anywhere then compulsory its return type must be Engine type. Internal implementation is whatever it is, but finally the expected value from this method must be Engine type object. |

Before spring, in case of pure java, java developer needs to provide the implementation for abstract method, but in spring, developer no need to provide implementation for abstract method. Spring will provide the implementation for abstract methods.Spring only will generate implementation class internally and return the object of lookup method.

Spring does override operation also, developer need not to do overriding manually in child class. Spring only will generate child class internally and does overriding.

Here in the above class Car is abstract class and abstract class object cannot be created in java, it is not possible in spring also and this class contains one abstract method whose return type is Engine which is nothing but a class. So if we implement myEngine method then definitely it will return Engine class object

In spring we can implement myEngine method using <look-up> tag which will return Engine type ref.

The fallowing bean xml for Car and Engine class is given as

|  |
| --- |
| <beans>  <bean id =”c” Class = “Car”>  <**lookup-method** name=”myEngine” bean=”**e**” />  </bean>  <bean id =”**e**” Class = “Engine”/>  </beans> |

In the above xml we have one bean configuration for abstract Car class where we have implemented its abstract lookup method myEngine using lookup-method tag and we are returning object (Engine class type) which is being referred from Engine class bean configuration. In other word we have created object of Engine and same object we have mapped under lookup-method so that finally it could return the expected object (Engine Type).

Now when we load the above xml file into IOC container then internally spring IOC will generate one runtime proxy class “Carproxy” which will extends Car class and inside this proxy class it will do the actual implementation of lookup-method “myEngine” and will return the object of Engine type.

|  |  |
| --- | --- |
| Class Carproxy extends Car{  @override  public Engine myEngine(){  return **e**;  }  } | After reading xml file IOC will see only one method under lookup-method tag under Car bean class so it will generate child Carproxy class and inside this it will do implementation of abstract class and return the same object what is declared as bean-ref. |

Now if we try to get object of abstract class Car using getBean method then?

|  |
| --- |
| Car cobj= (Car) getBean (“c”); |
| Here the above code seems to provide abstract Car class object but actually it is returning Carproxy child class object, because abstract class object can’t be created. |

So finally in short, when we have any lookup method to be implemented in spring then we mention this method as lookup-method inside xml file. After loading this xml into IOC, internally spring will generate one child proxy class by extending original class and inside this child proxy class spring will do actual implementation of lookup-method and will return one object to child proxy class. Finally when we try to create object of original class then internally it will provide object of child proxy class object.

Example of Lookup method DI

**MethodReplacer (I):**

As from the Lookup method DI basically we were doing method overriding whether it is Interface method or abstract class method or concreate class method. Here the method overriding was happing in proxy classes which were creating internally by assigning it under lookup method tag.

In the same way we have one more tag “Method Replacer”. Using this tag we can override existing implementation without extending or without doing any edition in the existing class or without doing any DI (as we have done in case of Lookup Method DI). Let’s understand with the help of example:

Let us consider we have one Class “Bank” and inside this class we have several methods like

[Deposit (), withdraw (), cal\_Interenst ()]

Now in this class we want to replace implementation of one method (cal\_Interenst) without extending existing class “Bank”. Then this can be done by using method replacer method of the spring. Here to replace the particular method from the existing class we will provide the patch by using method replacer.

|  |  |
| --- | --- |
| **Existing Class:**  Class Bank {  Deposit(){  S.o.p(“Deposit in Axis”);  }  Withdraw(){  S.o.p(“withdraw in Axis”);  }  Cal\_Interenst(){  S.o.p(“7.5 in Axis”);  } | **New patch to replace existing method**  Class New\_Cal\_Interest **implementsMethodReplacer**{  Public Object reimplement(Object O, Method M , Object param[]{  // Here do your new implementation for new interest  S.o.p(“10% in Axis now”);  }  Note: Here to create new patch we must need to implement one interface “**MethodReplacer**” and we must need to override one method “reimplement” where we will do the necessary changes against existing method functionality. |
| **Now the necessary tag in Spring configuration file.**  <beans>  <bean id=”bnk” Class = “Bank”>  <replace-method name=” Cal\_Interenst” replacer= “**new\_cal**”/>  <bean>  <bean id =”**new\_cal**” Class= “New\_Cal\_Interest”/>  </beans> | |
| **Now load this spring.xml file into IOC and try to create object of Bank class.**  Bank b = (Bnak)ap.getBean(“bnk”);  **b. Cal\_Interenst();**  According to above code the output should be “7.5 in Axis” but the actual output will be “10% in Axis now”.  Explanation: While load spring.xml file into IOC then here we have two tag defined under Bank class bean tag:  **replace-method**: Here we need to mention the method name whose functionality has to be replacing without extending its existing class.  **replacer: Here** we need to mention the bean id of the reimplement class.  Now when container finds these to tag then Internally by extendingBank it create one proxy class and override the existing method Cal\_Interenst () with the same name but all the existing code of Cal\_Interenst () method will be replaced with reimplement () method of new patch class New\_Cal\_Interest.  **The equivalent proxy class code created by IOC container is**:  Class BankProxy **extends** Bank{  @override  Cal\_Interenst(){  S.o.p(“10% in Axis now”);  }  }  In this way when we do [**b.Cal\_Interenst ()**] then calls the override method of BankProxy class instead of calling existing method of Bank class and we will get new result expected. | |

**Example of MethodReplacer:**

**Autowiring by using annotation @Resource: (Paste this after Autowiring topic)**

As @Autowired annotation does the automatic DI by using byType. But @Resource does the automatic DI by using byName search mechanism. So here the name of Class reference and bean id must be same. Here we must need to fallow naming convention strictly.

Example:

**Autowiring by using annotation @Inject:**

@Inject annotation does the automatic DI by using byName first. Here in this annotation we can do Autowiring by using byType also, we just need to declare @Qualifier (“value=type”)

@Inject

@Qualifier (“value=e1”)

Note: @Autowired annotation is found in Spring3.0 onwards while @Resource and @Inject are found in java.utill.javax package. ByType search mechanism is recommended for Automatic DI. So we must go either with @Autowired or @inject. And @Resource and @Inject annotation is recommended as these annotations are provided by java. So in future if we remove spring then this Autowiring functionality will not be affected and we can run the application using EJB3.0 supporting Tomcat server to run the application. EJB is given by Sun only.

**@Named:** This annotation is similar to stereo type annotation @Component. This @Named annotation is given by J2EE. So instead of using @Component annotation we can use @Named annotation for bean configuration

**Example:**

**Injecting properties file data into Class:**

So far we came to know how to inject property file into properties references but now we will learn how to inject properties file data into class. As we know that in properties file data are stored in string datatype in the form of Key-value pair. So a property is a type of map but it stores the value (key-value) in the form of string only.

Let us consider we have one property file (DB. properties) having the fallowing data

|  |
| --- |
| Driver= oracle.jdbc.Orcledriver  Url = jdbc:orcle:thin:@localhost:1521.exe  Usr = System  pwd= Manager |
| Class ConnectionPool{  public String driver,url,username,password;  // setter method of all variables  } |

Now if the above properties file data we want to inject into class ConnectionPool to create Oracle connection. Let’s say the class have dependencies driver, URL, username and password. And we have setter method of these dependencies in the class. So the question is by using spring how to read the property file and how to pass the property data into class.

Ans: By using expression we can inject property file data into class. We have to fallow some steps.

|  |
| --- |
| <beans>  <!—Loading property file -- >  <bean class=” **PropertyPlaceholderConfigurer** “>  **<property name=”location” value=”DB.**Properties**”/>**  **</bean>**  **<!—**Reading property data from IOC context using expression **-- >**  **<bean id = “cp” class= “ConnectioPool”>**  **<property name =”driver” value=”${**Driver**}”/>**  **<property name =”url” value=”${**Url**}”/>**  **<property name =”user” value=”${**Usr**}”/>**  **<property name =”password” value=”${**pwd**}”/>**  </**bean**>  </beans> |

1. First of all we will load the property file into IOC container. we will have to use one class [**PropertyPlaceholderConfigurer**] inside spring configuration file like given below to activate property placeholder:

<**bean class= “PropertyPlaceholderConfigurer”>**

**<property name=”location” value=”DB.**Properties**”/>**

**</bean>**

1. This class has one method [setLocation (String location)] where we have to pass the property file location.
2. So by passing property file into IOC, we can load the property file data into IOC context i.e. the IOC will store the data into its context scope.
3. Finally by using expression ${IOCContextprpertykey} we will read the data from IOC context scope. This expression will be written inside spring.xml configuration file under <property> tag of <bean> tag. Like given: **<property name=”driver” value=”${ Driver}”/>**
4. **So final Spring.xml for configuring DI is given below.**

Now Load this spring configuration file into IOC, IOC will setter DI into ConnectionPool class. Hence in this way we can inject property file data into class.

**Example: PropertiesFileDataInjection**

**I18N (I Eighteen N-) and L10N (L Ten N) For Language and Business support:**

**I18N** (Internationalization) is used for language support so that product could be reachable to the local people. So if the same application is for several countries i.e. there is only change in language then in this case I18N must be used for different language support of the same application.

I18N full name is Internationalization (there is 18 letters between I& N) that is why in short we write it **I18N**

**L10N (Localization)** is used for business support or for validation support. So if the same application but having different business logic according to location then in this case we must use L10N. For example Bank Application where the interest calculation may differ from country to country so in this case we will use L10N support. **L10N**full name is **Localization** (there is 10 letters between L& N) that is why in short we write it **L10N.**

**Diagram: Pending**

**Note**: Usually for big application it is not recommended to use. In this condition we should have one-one each different application based on their business and locality.

Now if the I18N support we have to implement in spring then it can be provided by using IOC J2EE ApplicationContext container.

Now the question is how to provide I18N support in case if require multiple language support in our application?

For supporting multiple languages in our application we have to create one-one separate property file for each language of the application and in the particular property file we have to put particular language character (Unicode). For example if our application requires two languages support Hindi and telgu so for Hindi and Telgu we have to create two separate property file. One for Hindi and another for Telgu and each property file will have their own Unicode according to language and based on the browser language we need to read and populate these Unicode onto the browser.

For reading Unicode from property file we can use IOC J2EE ApplicationContext container.

Same thing we can do from core java also. By using ResourceBundle class. Steps.

1. First of all we have to create Locale object and we will pass required language.

**Locale lo= new Locale (“Hi”);**

1. Create a ResourceBundle class object and will pass property file name without extentation of Hindi language and Locale class object created above.

**ResourceBundle rb = ResourceBundle.getBundle (“HindipropertyfileBaseName”, lo);**

When the above line gets executed so based on locale object ,Hindi property gets selected.

1. Now read the Hindi Unicode from property file using getString(“key”) of ResourceBundle class. Where based on key it will return the Hindi Unicode.

**String HindiValue= rb.getString (“lable”);**

1. Now finally we will print Hinidi Value on jsp page using

**out.println (HindiValue);**  Here the Unicode will be converted in Hindi on the browsers.

**Same thing we can do by using IOC J2EE ApplicationContext: Steps**

1. First we have to create **ApplicationContext** Interface reference.

**ApplicationContext ac = new ClassPathXmlApplicationContext (“Spring.xml”);**

1. Now using ac will call one method getMessage. We will pass property key, and language locale value Hi. It will return the Unicode against passed key and language value.

String HindiValue = ac.getMessage (“lable”, null, lo);

**Q:** Here one question arises that we are not passing property file base name as we have passed in ResourceBundle, so how the container will understand that it has to read the particular Hindi language property file.

**Ans: So the answer is will pass the base name of property file in spring.xml file, like given below.**

|  |
| --- |
| **Spring.xml**  <beans>  <bean id =”messageSource” class=”**ResourceBundleMessageSource**”>  <property name=”**basename**” value=”**HindipropertyfileBaseName”/>**  </bean>  </beans> |

Here in the spring.xml file we have mentioned one class **ResourceBundleMessageSource,** this class has one setter method “**basename**” in which we are injecting as value property file base name **HindipropertyfileBaseName.** Now when we load this spring.xml file into IOC container then the J2EE ApplicationContext will understand that we have to read from this property file declared in the xml. And finally using **ac.getMessage (**(“lable”, null, lo) we will get the Unicode of the property file.

**Note**: So property file base name will be configured into spring.xml file and compulsory we have to declare id=”**messageSource**” and class=”**ResourceBundleMessageSource”.** Only this single class in spring which forces to put this id =”messageSource”

**Example of I18N:**

**Create one web application and**

**Create one index jsp file,**

**Create two property file (Data\_hi.properties and Data\_en.prpperties).**

**Dynamically we need to choose these two files.**

**For getting files dynamically first we need to know the browser language using getHeader.**

**Event Handlers in spring**