**Dependency Injection:**

The way in which you decouple the conventional relationship between objects.

Suppose there are 2 objects, one is depend on the other, the idea is to decouple the dependency.

So that they are not tightly coupled to each other.

**Spring Factory Bean:**

Dependency Injection concept is possible because the spring is actually a container of Beans and spring behaves has a factory of Beans.

What do I mean by Container? Tomcat is a Servlet Container; Tomcat creates the Servlet objects, which are required in order to run an application. So, what you do, you will configure all your servlets into xml and you would supply the classes and what Tomcat does is reads the XML and then it identifies what are the servlets to be instantiated and then it creates those servlets.

Spring its container something similar but not container of Servlets but container of Beans. So pretty much any pojo that you have can be contained inside a spring Container. Inside spring container you can have as number of objects you can have and all these are managed by Spring container. Managed in the sense, it handles the instantiation of those objects, the whole life cycle of the objects and it handles destruction of those objects.

You can have objects out of the Container, but having inside has an advantage because spring handles object life cycle.

How an object is instantiated by Container? Using Factory Pattern.

We have another Object Factory, which is another Java object. The whole job of this Object Factory is to create Java Objects. This Object Factory reads from a Configuration. An object will tell to Object Factory that, I need an object with some specification, and then Object Factory finds it in the Configuration file. Now, Object Factory knows what object needs to be created, so its going to create the required new object and then its handed over to the requesting object.

So, how it works in spring is, spring has an object called Bean Factory in order to spring create objects for us.

I have an object here, instead of calling a new, I would reference Bean Factory. The Bean Factory would read from Spring XML, which contains all bean definitions. Bean Factory creates a Bean from Spring XML and it makes a new Spring Bean (nothing but object). And this Spring Bean is handed over to the Object. The advantage here is this Spring Bean is created in Bean Factory and spring knows about it and handles this bean and manages entire life cycle of the Spring Bean created.

If I am using property to Inject then it is setting through Setter type. This is called Setter Injection.

Constructor-arg is used to specify the Constructor Injection.

We have type and index attributes for representing. Index is used to tell the order of the parameters defined. Type is used to define the type of parameter defined.

<alias name =”triangle” alias =“alias-triangle”> -- This is used to have another name for triangle. We can also use alias name for bean reference or in getBean() also.

We can also give alias name in Bean definition also with “name” attribute in bean tag.

But always better to use id’s because; it has XML validation, that is only one id in the configuration. But we can same alias names.

In “ref” attribute we can define id, name or alias names. But if you want to restrict it to only to “id”, then use “idref” attribute.

<property name=”abc”>

<idref =”cde”/>

</property>

**Bean Autowiring:**

Autowiring is the feature provided by Spring framework, that helps us to skip some of the configuration that we have to do.

We have seen that every member variable in the bean (pojo) has to be configured.

**byName** – Bean name and parameter name should be same

**byType** – It will work only when we have only 1 type

**constructor** – It will work same as byType, it will work only when we have only 1 type of parameter in Spring Bean(Pojo)

**By default, if no autowire is mentioned in Bean definition, then Autowire is off**

**Bean Scopes:**

Creation of Beans are happened at the time of ApplicationContext is initialized. Not at the time of getBean() is called. This is the default behavior of the beans.

**Singleton** – Only once per Spring Container. By default all beans are singleton

**Prototype** – New bean is created with every request or reference

Singleton default is what causes Spring to initialize all the beans during load time itself. So, if a bean is defined as a Singleton then when the ApplicationContext is initialized then all the Singleton beans are initialized. However, if the bean is defined as prototype then spring waits for the getBean() to happen and only then it initializes a prototype.

**Request** – New bean per servlet request

**Session** -- New bean per session

**Global Session –** New bean per global HTTP session

If any bean implements ApplicationContextAware class, then that bean can use ApplicationContext.

To know the bean name, BeanName class.

**Bean Definition Inheritance:**

We can achieve inheritance concept i.e., we have a attribute called “parent”. Define bean with parent=”parenttriangle”, so that it inherit all the features from the parent to child.

Abstract=”true”, will help you not to create an Object for this bean definition. This bean can be used a template.

**Lifecycle callbacks:**

InitilizingBean help us to know when the bean is initialized. We override afterPropertiesSet() where we can write our code to do the things on start up of then bean.

DisposableBean help us to know when the bean is destroyed. We override destroy() where we can write our code to do the things on destroy of the bean.

registerShutdownHook is the API used to shut down the ApplicationContext. This method is available in AbstractApplicationContext.

If we use Spring specific classes, again our beans are tightly coupled with Spring. So, instead of that write your own method and define that method name in bean definition.

init-method=”myInit”

destroy-method = “cleanUp”

You can also define them at global level, if all beans has init-method and destroy-method.

default-init-method=”myInit”

default-destroy-method = “cleanUp”

If we define both the ways, the first Spring methods are called and the our defined methods are called.

**Annotations:**

In order to tell that the particular member variable is mandatory, then we need to annotate that member variable with @Required annotation. So you are telling to Spring that this is mandatory member.

**@Autowired:** It looks for the type first in the configuration, if it finds only 1, then works fine. Else if it has more than 1 type, it looks for byName.

**@Qualifier:**  It is a something that I can mention in the bean definition to say that this bean is actually circle related bean.

<context: annotation-config/> -- Define all annotations related to bean postprocessor.