<https://www.youtube.com/watch?v=GB8k2-Egfv0&list=PLC97BDEFDCDD169D7>

Spring framework is lot of things put into one. Spring is really popular for dependency injection.

Two things which really means a lot

1:- Main goal of Spring framework is how to decouple relationship between dependent Object

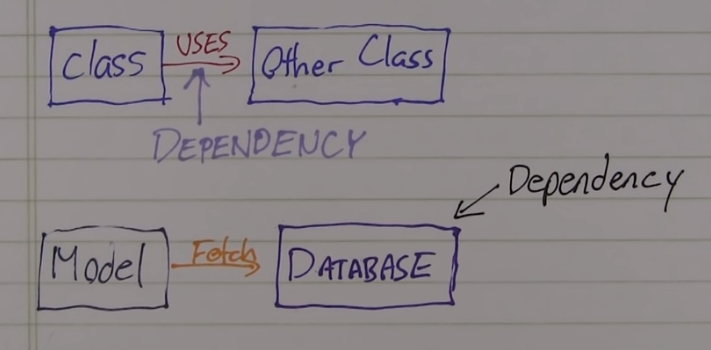
2:- Dependency Injection means separate the class which is dependent and inject them when its needed from outside.

**Dependency Injection or Dependency Inversion:**

**What is dependency injection?**

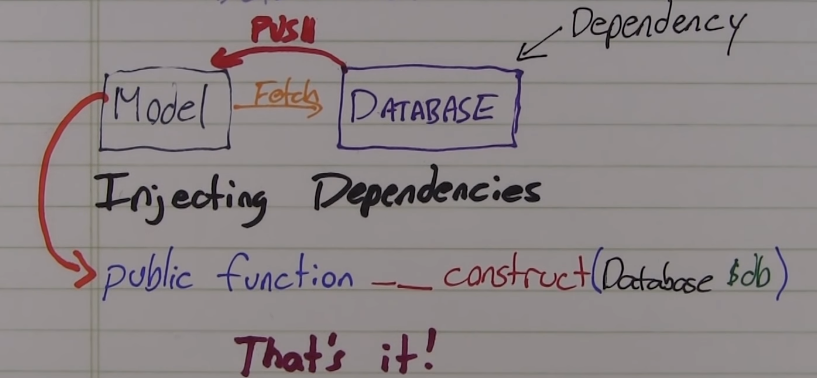
Well the literal meaning is to inject dependency.

**What is dependency?** It's just another object that your class need to function. For example, if you have a model class that fetches data from a database object we can say that your model class has a dependency of that database object.



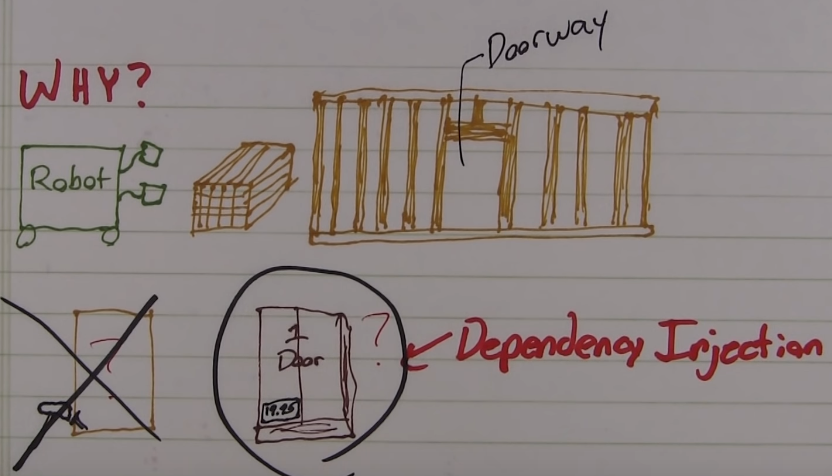
Lets look at what it means to inject dependencies?

Injecting dependencies just means that dependency is pushed into the class from the outside. All that means is that you shouldn’t instantiate dependencies using the new operator from inside of the class, instead take it as a constructor parameter or via a setter method.



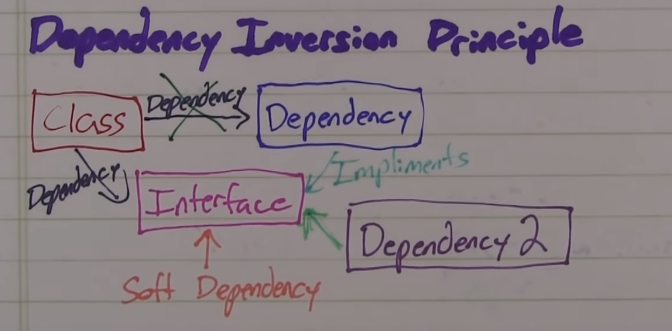
Why do we need dependency injection?

Lets imagine for minute you are programming a house building robot. You start with a pile of lumber and program it to start building walls. Then when you get to a doorway what do you do? Do you program to build a custom door out of raw material each time, or do you program it to take a ready made door from a supplier and install that. The more flexible way to do that will be to take the door from the supplier and that’s exactly what dependency injection does. It decouples your classes construction from the construction of its dependencies.



The reason that it is so important is the DEPENDENCY INVERSION PRINCIPLE.

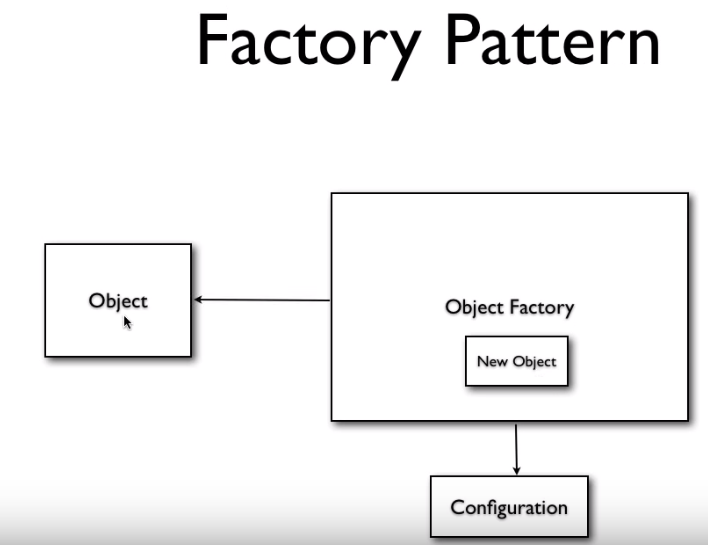
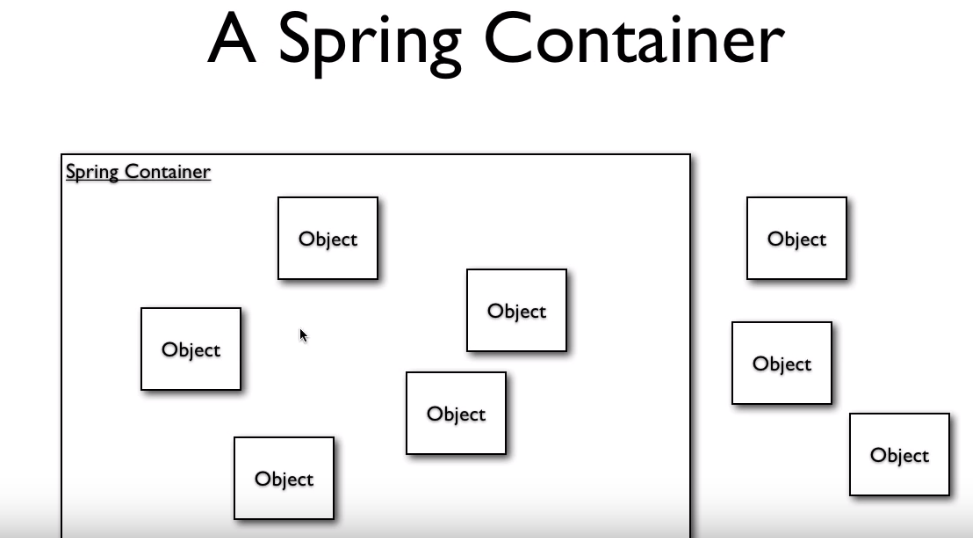
Basically dependency inversion is the principle that code should depend upon abstraction, by depending upon abstractions we are decoupling our implementations from each other. What that means is using interfaces or abstract classes which can lead to substitution of different dependencies as long as they satisfy the required interface. By using dependency injection we decouple our code from lower level implementations making our code cleaner, easier to modify and easier to reuse.



Now back to spring the whole dependency injection concept is possible in spring because the spring is actually a container of beans. It behaves as factory of beans. Now what does it mean by Spring container?

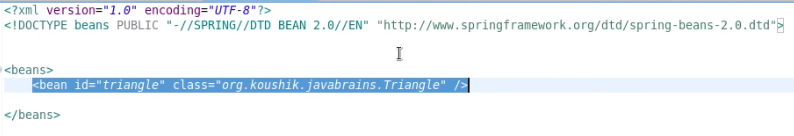
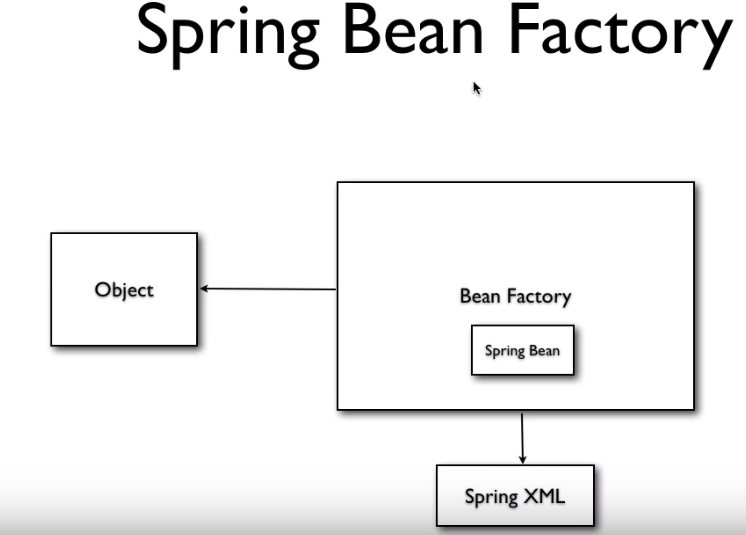
SPRING CONTAINER:

If we take an example of tomcat its an example of servlet container what it means is that tomcat creates the servlet objects which are required in order to run an application so when you are deploying an application what you will do is configure all the servlets in an xml file and you will supply the classes now what tomcat does is that it reads the xml file and it identifies what are the servlets that need to be instantiated and then it creates those servlets. So SPRING is something similar. Its a container but not container of servlets but instead a container of beans. So any POJO that you have can be contained inside a spring container. So in spring container you can have as many objects as you want and all the objects are managed by the spring container. Manage means it handles instantiation of the objects, handles who lifecycle of the object and it handles the destruction of those objects. You can have objects outside of the container as well in your application but adding objects into the spring container will give you lot of advantages.



The spring container works on the idea of FACTORY pattern.

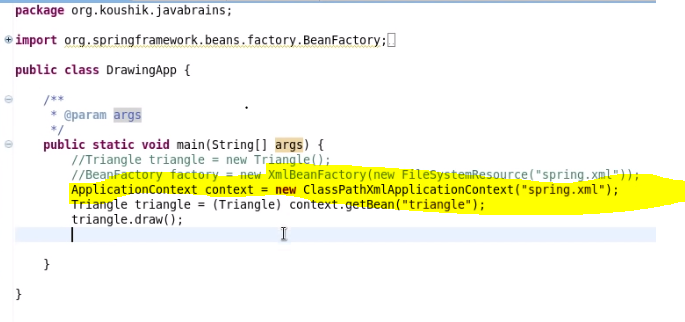
Suppose we have an object, instead of creating a new object b we can call another object which is a factory object, what this factory object will do is to create new objects and then hand it over to the object, so basically whole purpose is to product objects and pass them over. So how does the object factory produces an object what it does is it reads from a configuration file, so when the requesting object asks for new object the factory will read the config file and produce a new object and pass it over. So this is what done by spring as well. It has SPRING BEAN FACTORY, it has an object called bean factory. One can use the spring bean factory in order to let spring build new objects for us. The bean factory will get an order from some object and it will use Spring XML file to read the definitions for all the required beans. The bean factory creates a bean from the blue print of bean provided by xml file and makes a new spring bean, the spring bean is handed over back to the object. As the new bean is created by the SPRING, it knows about the bean so it manages the whole life cycle of the bean from creation to destruction so therefore its container for that bean.

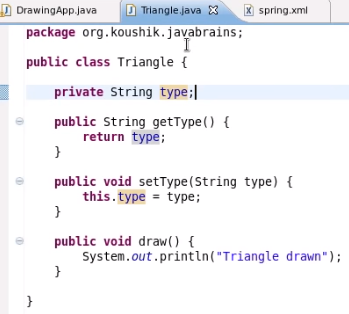


In spring.xml the highlighted part is the **BLUEPRINT** of the object we going to create.

Application Context:

It is like big brother of the bean factory. It does everything that bean factory does but it also has some additional functionality like event notification, EOP etc., the performance is same as well and you would only use bean factory when the resources are very crucial. It is an interface so you would have different implementations of it depending on your configuration.





**EXAMPLE:**

Having a member variable in a class and how we can instantiate an object with member variable preset. In order the extend the blueprint defined above, to prefill the member variable defined in the above snippet in Triangle.java class i.e. type variable. So the property tag can be used within the bean tag.

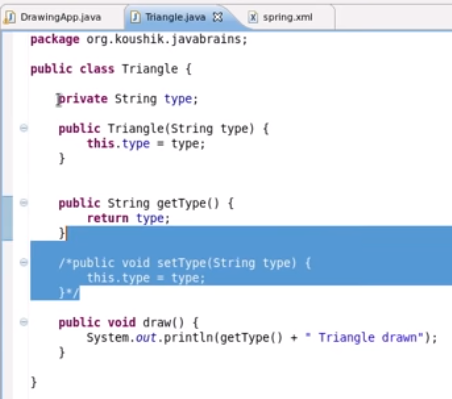


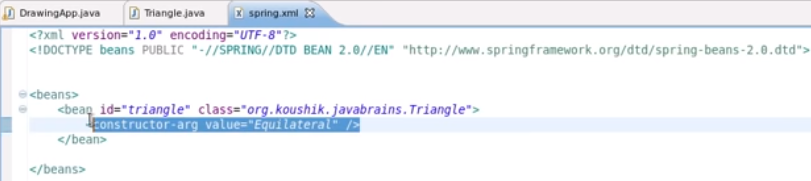
So what we are doing above is defining the blueprint of the bean and also defining preset value for member variable named “***type”*** so what’s going to happen is we ask spring to get us the object **(e.g. triangle),** spring will check the xml file for blueprint which defines the bean id with specified class and fill the value for the property assigned using xml file.

This is also an example of setter injection because setter method from Triangle.java class is used within the spring.xml file under property tag to assign ***“Equilateral”*** value to the variable **type.**

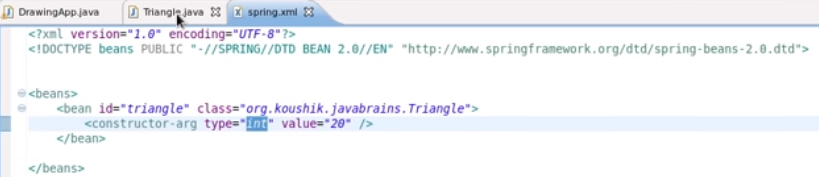
**Constructor Injection:**

To do this the process is nearly same just define the constructor to take initialize the value of a member variable instead of using setter method.

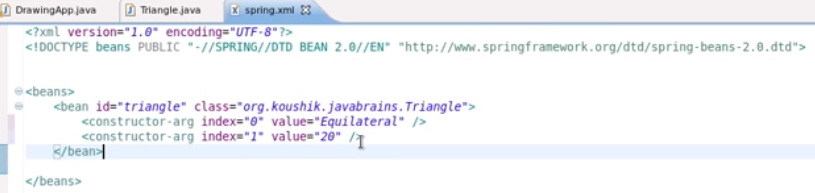




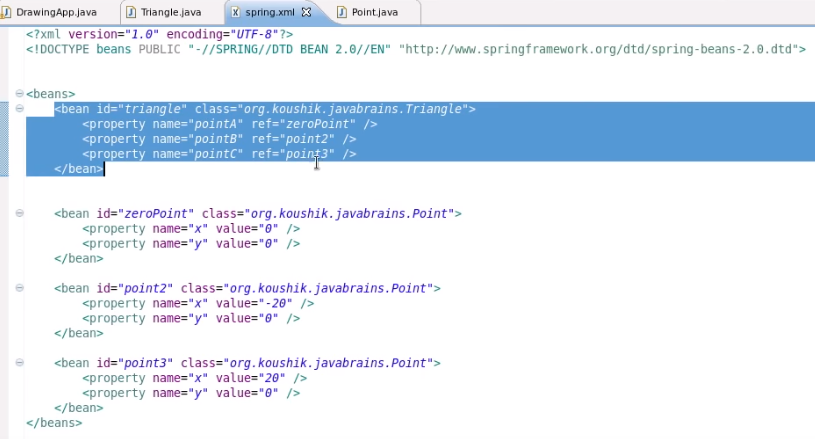
If you have multiple variables with different data types like sting, int, float etc. you can define the type in spring.xml file to tell spring which type of variable you are passing the value for.



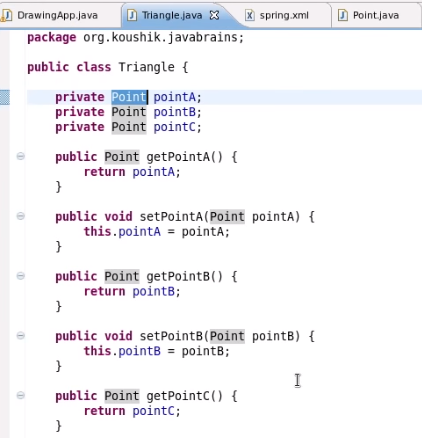
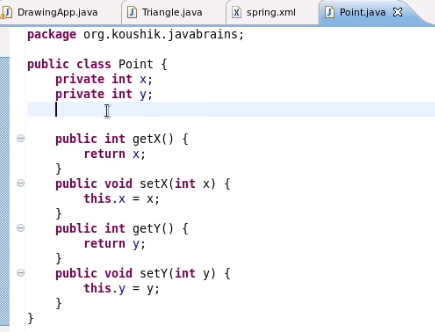
**Another way to do the same thing as above is INDEX.**



**Injecting Objects:**

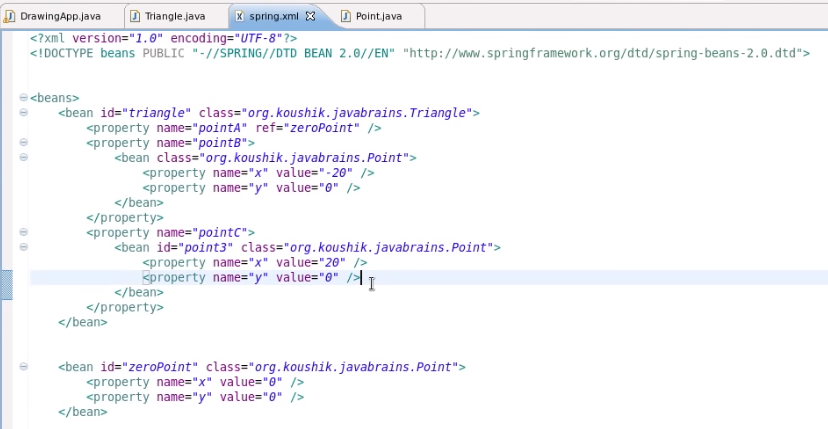


In the above snippet as it can be seen that the we have defined properties for triangle pointA, pointB, pointC which refer to other beans within the xml file. So there is another bean defined which of Point and it has two values for the properties set. When the request comes in this bean will be created first, properties will be set and instead of returning the bean, it gets assigned to the property of main bean which is, Triangle which is referring this bean. So spring do all this automatically for us. The good thing is you can go as deep in bean creating as you want, what it means that as Triangle class refers Point, its possibility that Point can refer other class and so on.



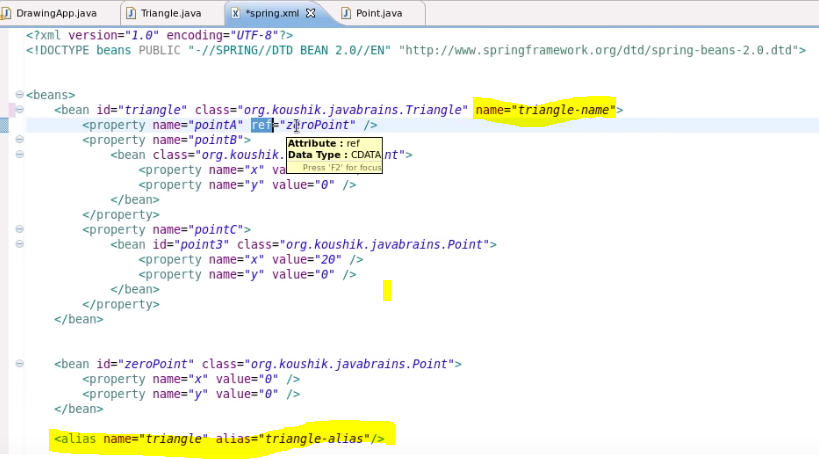
**Inner bean, Aliases, idref:**

The whole concept of the inner bean is that we don’t define the beans seperately but instead define them within the main bean. The inner beans are usually defined when we know the bean is only meant to be used by the main bean and no other beans will be using it. In the below snippet we are keeping “zeroPoint” bean outside, as its common one and can be used by other beans while the other two beans are moved inside the main bean. The benefit of inner bean is that we have isolated the bean only where it is required. Secondly you don’t need to specify Id for the inner bean because id is only required when you are referencing the bean from somewhere else.

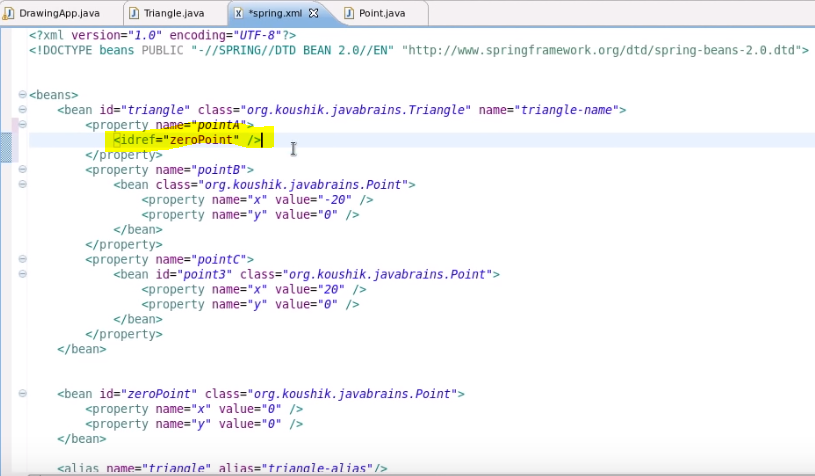


**Another naming facility available is: aliases**

The way to give aliases is to use the aliases tag give it a name and an alias-name.

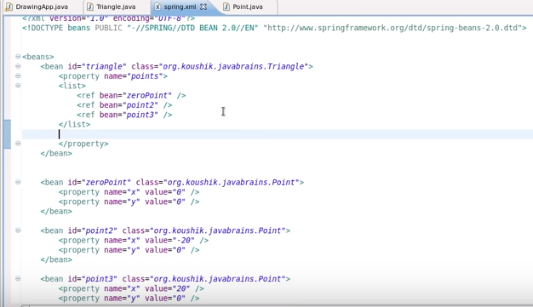


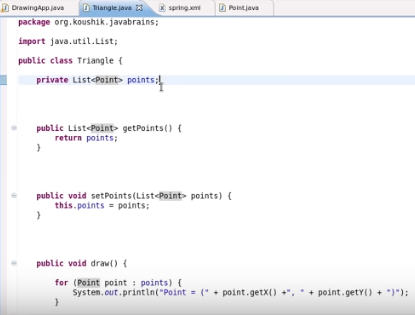
**Idref: what it does is it make sure that there is a bean with the defined id. It should be defined inside the property tag. This can be an extra validation for your spring.xml file.**



**Initializing Collections:**

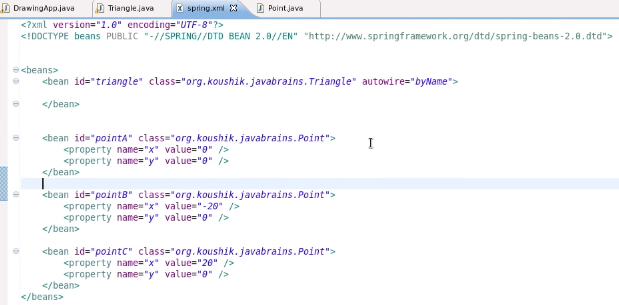
Spring provide support for three types of collections: set, list and map. This is to support if any of the member variable is defined of type collections. There are tags which can be used in the xml file, the list tag for example can be defined in the xml file to put values into the list collection. Another good thing about spring is that it automatically convert the defined values into the defined type of the variable.

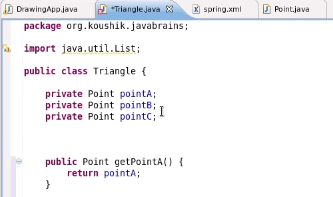




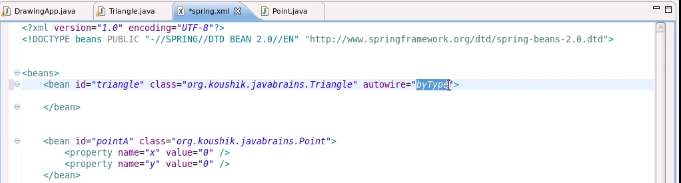
**BEAN AUTOWIRING:**

Autowiring is the feature provided by spring framework that helps us to skip some of the configuration that we have to do. One scenario for autowiring is that if the name of the member variables and those of the beans defined in the xml file are same we can use autowiring to configure it, so by defining **autowire tag** and giving it value **byName.** What byName does is that spring when finds a member variable with same name as that of bean it automatically reference the bean defined in xml file to the member variable.

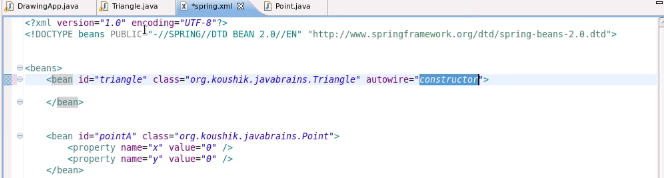




There is another way to autowire which is **byType,** this works when we only have only one member variable and only one bean of each type. Say for example there is only one member variable defined in a class and there is only one bean defined in the xml of that particular class, in that case what spring does is it looks for type even though the bean has been defined with a different name it looks for the type which is the class defined for that bean and if the defined class matches the class of member variable then it automatically does the wiring.

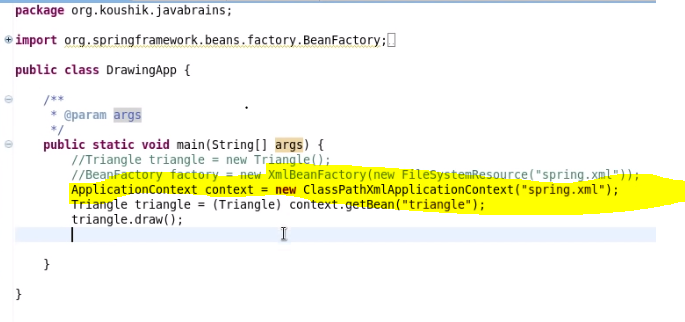


There is third way we can autowire and that is called **constructor,** it works the same as that of type but instead of using setter method to inject the value it uses the constructor to inject the value. The condition to satisfy here is that you need to have only one bean per type.



If you don’t specify any autowire then the default one is **autowire=”off”.** If you have large number of beans to define it is generally recommended to not use autowire and define them through reference instead.

**SPRING BEAN SCOPES:**



What’ happening is that: spring container is started when we define the application context by providing an xml file. What application context does is it actually refers to the spring.xml file this is before any call is made to the bean, so the application context reads the xml file and looks at all the beans that are available and creates them by itself, note that there is still no call to getBean. Now when another object that does the getBean then the applicationContext hands over the bean that is already been initialized, so the creation of the beans happens when the initialization of the application context takes place. This is the default behaviour of beans.

**There are two basic bean scopes:**

**Singleton:** only once per spring container. What this means is that when the application context is initialized and passed xml file it only initialize one bean per bean definition in xml file. No matter how many calls are made to the bean, the spring container return only that instance. It's not going to create new instances everytime there is only one instance that gets handed over for every call to the getBean. **By default all beans in spring are SINGLETON.**

**Prototype:** New bean created with every request or reference. What this means is that if another class makes call to application context for getBean then spring container would create a new bean and then pass it to that class. So its going to create new objects everytime there is a getBean.

Note: no matter if you call the getBean to get the object or you use reference within another bean to get object, new object will be created each time if the bean is prototype.

There are few other scopes which are **Web-aware Context bean scope:**

What web-aware mean here is that because spring ties in very well with web applications, say if you have jsp or servlet application you can use spring to create all your servlets to create objects related to your servlet application. Thus spring is capable of knowing when there is new request or new session so you can tie the bean scopes to the request and the session.

**Request scope:** New bean per servlet request. So spring will be aware of when a new request is happening because it ties in well with servlet api’s. Then depending upon the request it creates a new bean. So if you have getBean inside request scope then with every request its gona be new bean.

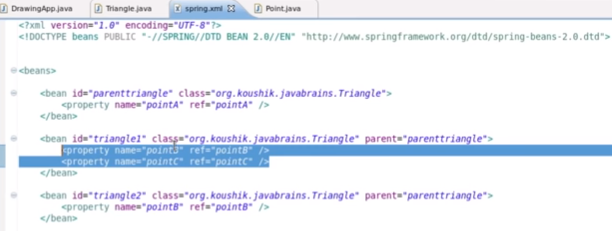
Same way you can new bean per session its called **Session:** As long as one user is accessing in a single session no matter how many calls to the getBean happens in the code only one bean is written but if its new user in a different session then a new bean is created. The third one is the **Global Session:** new bean per global HTTP session this is applicable in the **portlet context**.

**Using ApplicationContextAware**

This is an interface which is implemented by a bean, it makes the bean capable of being aware of **applicationContext**. The bean will have to implement a method called **setApplicationContext** which can be passed **context** object.

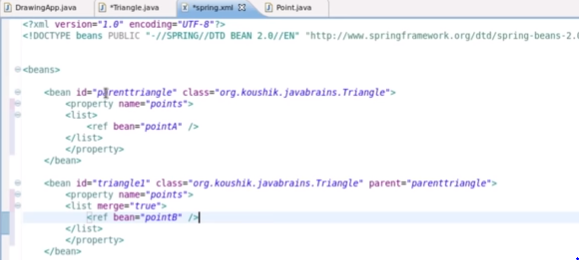
**Bean Definition Inheritance**

What this means is that if you have lots of beans defined in settings.xml file and there are some common set of values that we have to initialize for all the beans, then bean definition inheritence can be handy. So basically we can define one bean with all the values that are needed and then we can inherit that bean for all the rest of the beans. Now the parent bean can be a bean in itself or it can be an abstract bean so it can act as a template for all other beans. In the snippet below we will define a parent bean and then it will be used to define other beans. The tag used is parent. E.g., **parent = “xyz”**

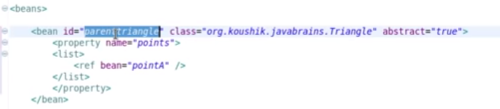




This will give the output for both coordinate A and B as triangle2 is inheriting from parenttriangle bean. So this is a handy feature if you have whole loads of common configuration to be done. One additional feature to this that we can also add and inherit a list using bean inheritance. In the below snippet say if we have a list of points and we define a parent bean with properties of that bean containing reference to some point A and we want to add to this list in the child bean we can use the following: **<list merge=”true”>** tag. So basically in beans inheritance we can add to the properties of the collection by using the merge option.



Also if there is a parent bean which you just want to define to be used as template and don’t want to initialize it you can define a property called **abstract=”true”**. What this does is spring will not create bean defined with this property.



**LIFECYCLE CALLBACKS:**

In case of desktop application you do need to close the context, so in the code we initialize application context and we get a bean from application context but we didn’t close this context, for Java EE or web application this is known but for Java SE this has to be done manually.



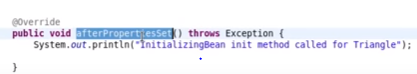
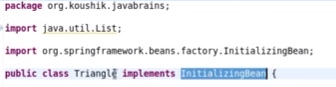
In order to close the application context we need to use another class which is called the **abstractApplicationContext** and that registers a shutdown hook for our java application so when our main method ends then the hook is called and the **context** also shuts down.

So to do this we will use **AbstractApplicationContext** and then calls the **registerShutdownHook** method so that application ends the spring context knows that it's time to call the hook and destroy all the beans.

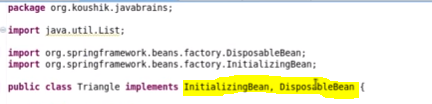


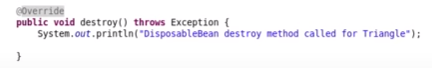
**Configuring Init method and Destroy method for beans:**

In order to have method called when a bean initialize one of the way we can get this done is by implementing an interface called **InitializingBean**, this interface tells spring that the bean **(E.g., Triangle)** needs to know when its initialized. So **afterPropertiesSet** method needs to be called.

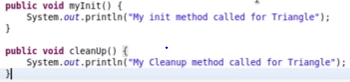


Similarly for capturing the destroy phase there is an interface called DisposableBean, this will provide a method called destroy which can be executed when the bean will be destroyed.

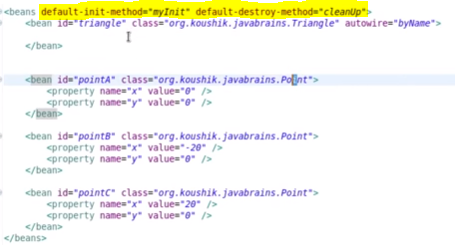




The above way is one of the way to implement initialization and destruction method for beans but disadvantage to this is that we are implementing interfaces that are specific to spring, so if you don’t want to implements some special interfaces then we can define our own methods and then call them in **spring.xml** file using keys **init-method=”name of the method”,** same way we can define the destroy method as well and the tag can be **destroy-method=”name of the method”**.



If you have a standard in your code that you want to have same name init and destroy method in all your bean class definitions then we can configure the init and destroy methods at global level like the below snippet:

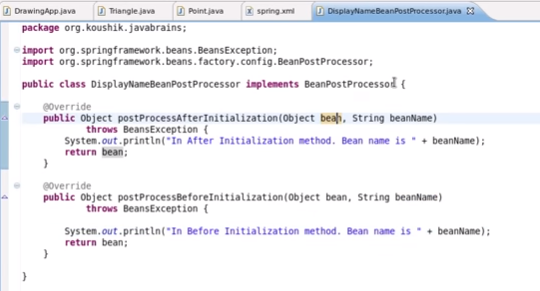


**BEAN POST PROCESSOR**

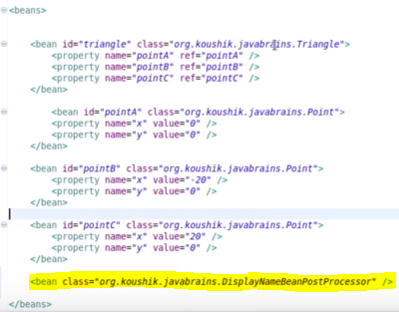
These are the classes that tells spring that there are some processing that spring needs to do after initializing the bean. When you write some code in your post bean processor then spring executes that code after initializing each bean.

The bean post processor method will be called after initialization of each and every bean defined in configuration file.

The bean post processor is a seperate class because it caters to initialization of different types of classes, so we have to define a seperate class within our code and place it within bean post processor then we define the bean post processor in our spring config so that spring knows about it, then when spring initializes each of the beans it will call the method bean post processor. The new class for post processor should implement **BeanPostProcessor**, which will indeed bring the **postProcessAfterInitialization** and **postProcessBeforeInitialization**.

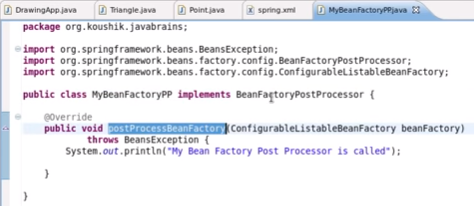


Spring expects an object will be returned. Now we need to tell spring to register the above defined class Bean post processor. So, what we do is define a bean for that class, we don’t need to define Id for this bean as it won’t be referenced by any other bean. Just by defining like below it will tell spring to automatically pick this Bean processor for other beans.



**BEANFACTORYPOSTPROCESSOR:**

This is the feature provided by spring if you want to execute code when the bean factory itself is initialized then you can use **beanFactoryPostProcessor** and you can use some of the bean processors that come out of the box. The process of creating this is same as that of the above way of declaring a class that implements the interface named **BeanFactoryPostProcessor** and declare the associated methods and then declare that class in the **spring.xml** file.



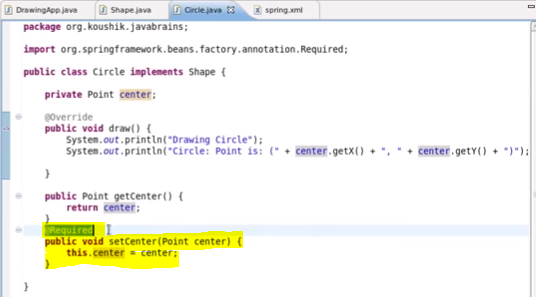


**Annotations and Required Annotation:**

The annotations are the second way in which we can configure the spring beans and its dependencies. Spring provide annotations based container configuration, there are quite a lot of annotations which we can use to configure the beans.

**Required Annotation:**

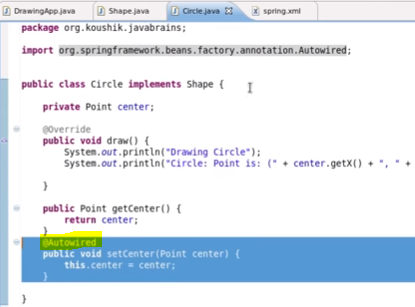
It is one of the simplest annotations to be used for bean configuration. This annotation is basically to make sure that when beans are created the variables or parameters are set as required so they do get initialized and we don’t get null pointer exception. This is to make sure the exception is thrown as soon as it happens during initialization of the bean instead of waiting until the end. To get this done we need to define the annotation in the bean class for the setter method to make sure its called to set the value and then define bean post processor which basically runs the validation at the back end. So bean post processor looks for the defined annotation and will raise exception if the condition or annotation rule is not met.



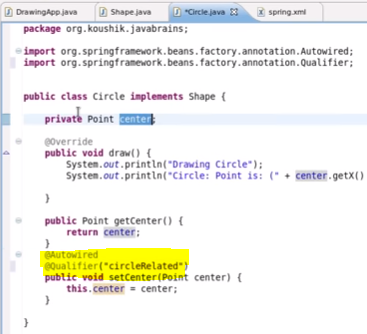


**Autowired Annotation:**

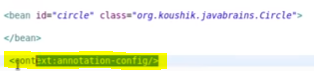
This is used to specify any wiring needed to define dependency. The @Autowired annotation will get this done for us. The autowired works by looking at the first the name of the class for which the bean is created in case of single bean defined for that class its ok but for multiple beans for same class type the spring will look at the name of the bean and if it matches with the name of the member variable for which the bean is used to populate it.



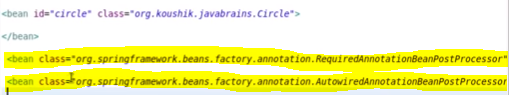
So first it looks for the type of the bean and if it find multiple beans of the same type then its going to look for clues in the bean name. If there are multiple beans with same type and name is also not matching to that of member variable or we couldn’t change the name of the bean because its already in use at other places what we can do is use the **@Qualifier** annotation. A qualifier is something we can mention in the bean definition to say that this bean is related to defined qualifier.



Instead of defining each and every post processor in the spring.xml for different annotations we can define annotation config tag which can help process all the annotations. What this will do is it will take care of initialization of all the post processors which are required for annotations.



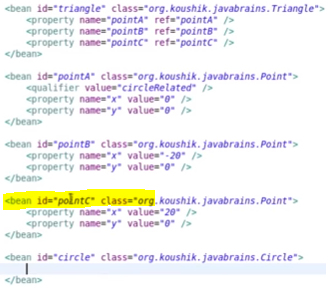
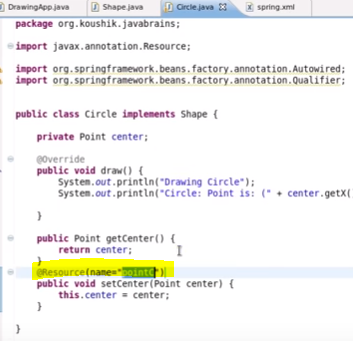
**INSTEAD OF**



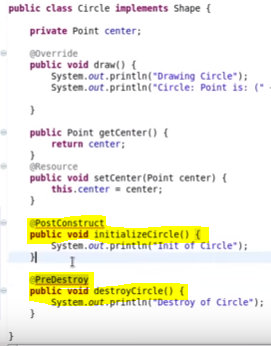
**JSR-250 ANNOTATIONS:**

This is a standard, the java specification request that defines all the standard annotations that would apply across different technology and different frameworks. Spring support some of the standard JSR-250 annotations.

@Resource: it can be used to inject dependencies into member variable of a class.it can be obtained by **importing javax.Annotation.Resource**. It can do standard dependency injection by name.we just need to define the name of the bean and it will look for it in the settings.xml and inject that. If we don’t define the name that it will look for bean with the same name as that of member variable.



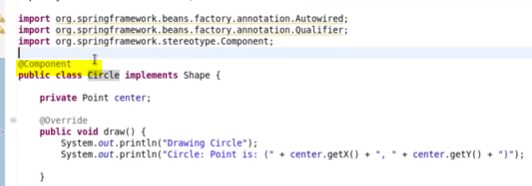
We can define annotation to call the initialize and destroy methods of the bean as well. So the annotations are: @PostConstruct and @PreDestroy.



**COMPONENT AND STEREOTYPE ANNOTATION:**

The annotation that declares a class as a bean, so we don’t have to even add xml tags and we can have all the configurations defined inside class using annotations.

So to achieve this we need to tell spring which classes we want to use as beans. So **@Component** annotation can be used to define if the class is to be used as bean. What spring does is when it sees this annotation within the class it automatically provides the configuration at the backend similar to above snippet so we don’t need to define the configuration as per above snippet.

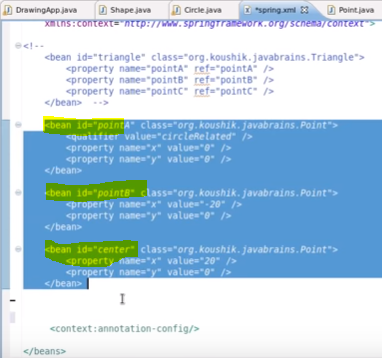


We do need to define in the xml file some tag so that the component tag is looked for within the class definition to make sure beans are defined inside the code.



The above snippet shows that tag which need to be defined in the xml file to tell spring that it needs to look for component annotation. We define the base package to make sure the search is quicker to look within certain package and look for annotation.

One disadvantage of using annotations is we can use the same class to define different bean with different metadata while if we define an annotation with a class we can’t do that. For example in the snippet below if we use the annotation for point class we won’t be able to define different beans for point class with different flavours.



StereoType Annotations:

When you write an enterprise application in spring you would have some standard spring beans that perform some standard roles. You would have a data object, a service class, a view, a controller. So all these are stereotypical roles that spring performs in any enterprise application. So what spring does is it has some annotations that let us define some particular beans to do one of those roles,

**@Service** is another annotation which can define the class as service layer class, **@Repository** is another annotation to define the class as data class.**@Controller** now this annotation, which defines the class as C of the MVC pattern. The initial working of all these annotations is same as that of component, it also defines what role the bean is playing, provides automatic documentation to define the role of the bean.

**Event Handling in Spring:**

There are 3 basic components event handling consists of:

Links:

<https://www.youtube.com/watch?v=IKD2-MAkXyQ>