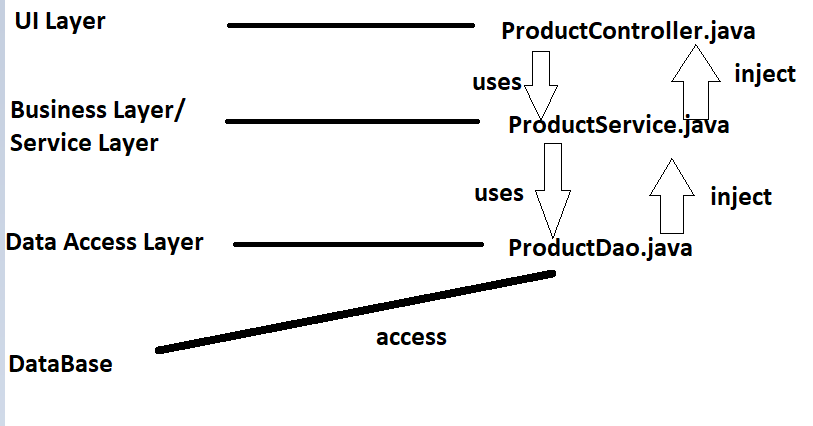
**Basic Terminology**

1. DTO : Data Transfer Object
2. DAO : Data Access Object. Used to fetch and add data to database
3. ORM : Object Relational Modal

**Inversion Of Control**

Giving control of object creation to Spring so that the object created can be injected to another class.



ProductService.java contains method of ProductDao.java. Similarly ProductController.java contains method of ProductService.java.

These methods can be created manually(not preferable) or we can ask Spring to use dependency Injection for creating method.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Example of UI Layer 🡪 Spring MVC,JSF

Example of Business Layer/ Service Layer 🡪Spring Security Module,Transaction Mgmt Module

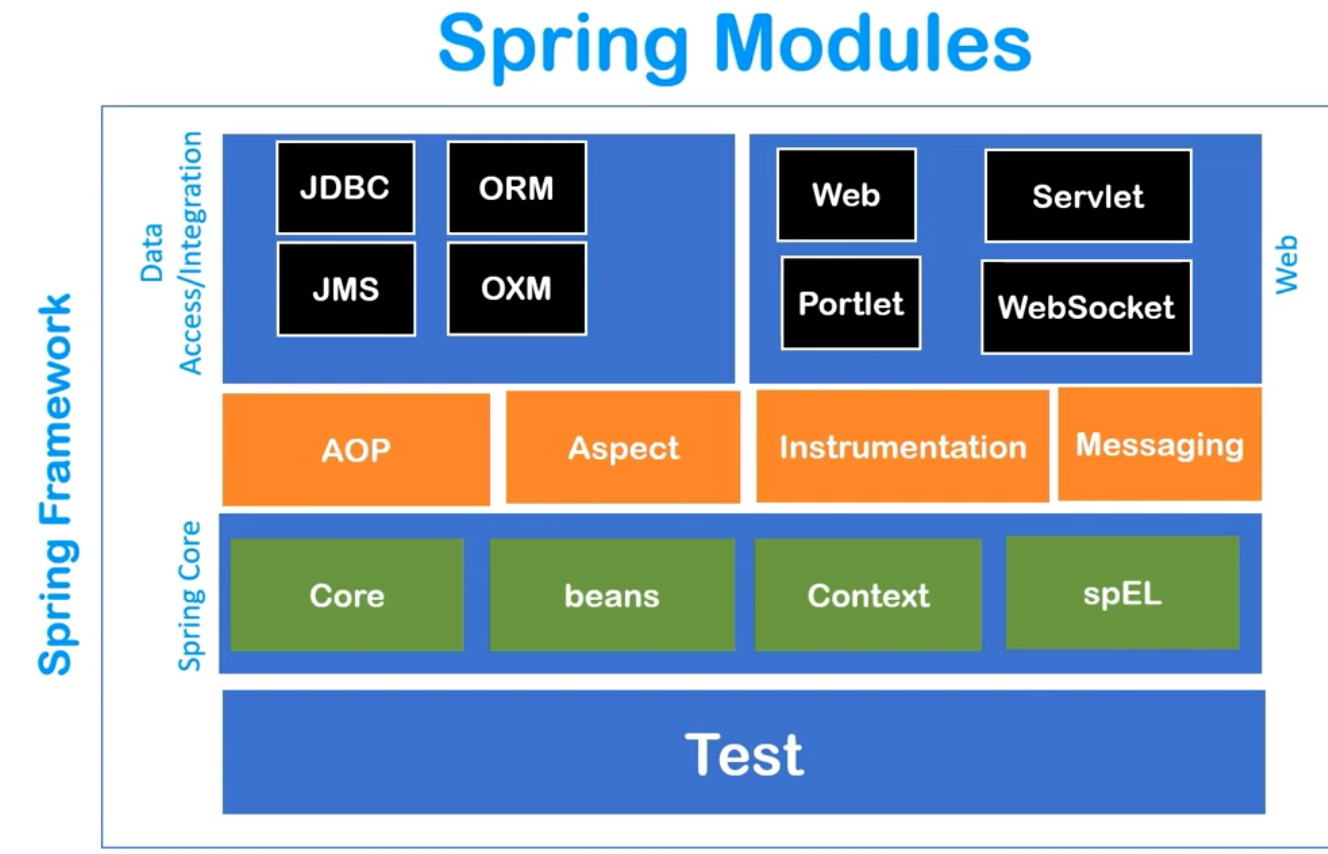
Example of Data access layer🡪 Spring JDBC Module, Spring ORM Module

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Spring internally uses meta Data or Annotations or XML for injection.

Spring ORM Module allow use of another ORM tools like Hibernate

**SPRING MODULES**

****

1. Core
2. Beans
3. Context
4. spEL (spring expression language)

Note :- Context module inherit many feature from Beans Module. Many other features are also added like internationalization, event propagation, resource loading & transparent creation of context. It also provide J2EE features like EJB, JMS & Basic Remoting.

1. Aspect Oriented Programming(AoP) 🡪 It is another module which allows us to define method intercept & point cuts so that we can decouple the code very easily.
2. Data Integration Module has 4 Module
3. JDBC :- can connect with database
4. ORM :- Provide integration with other ORM tools like hibernate.
5. JMS :- Java Messages Service
6. OXM :- Support Object XML Mapping Tools (such as JAXB,XStream)
7. Web Module has 4 Module
8. WEB
9. Servlet
10. Portlet
11. Web Socket
12. Test Module is used for testing. It provide Junit & Test-ng for unit testing and integration testing.

**SPRING IoC Container**

Spring IoC container performs:-

1. It creates the object.
2. Hold the Object in memory.
3. And Inject them in another object (Dependency Injection)

It must know following things before performing its task.

1. Beans or POJO classes it has to manage.
2. Configuration Files. Generally XML Configurations. In configuration files we tell that which bean is dependent on other things.

**Application Context**

It is an interface which represents Spring IoC Container. It also implements BeanFactory. Since it is an interface, we need to create sub class of an object. Some of the important subclasses are :-

1. ClasspathXMLApplicationContext
2. AnnotationConfigApplicationContext
3. FileSystemXMLApplicationContext

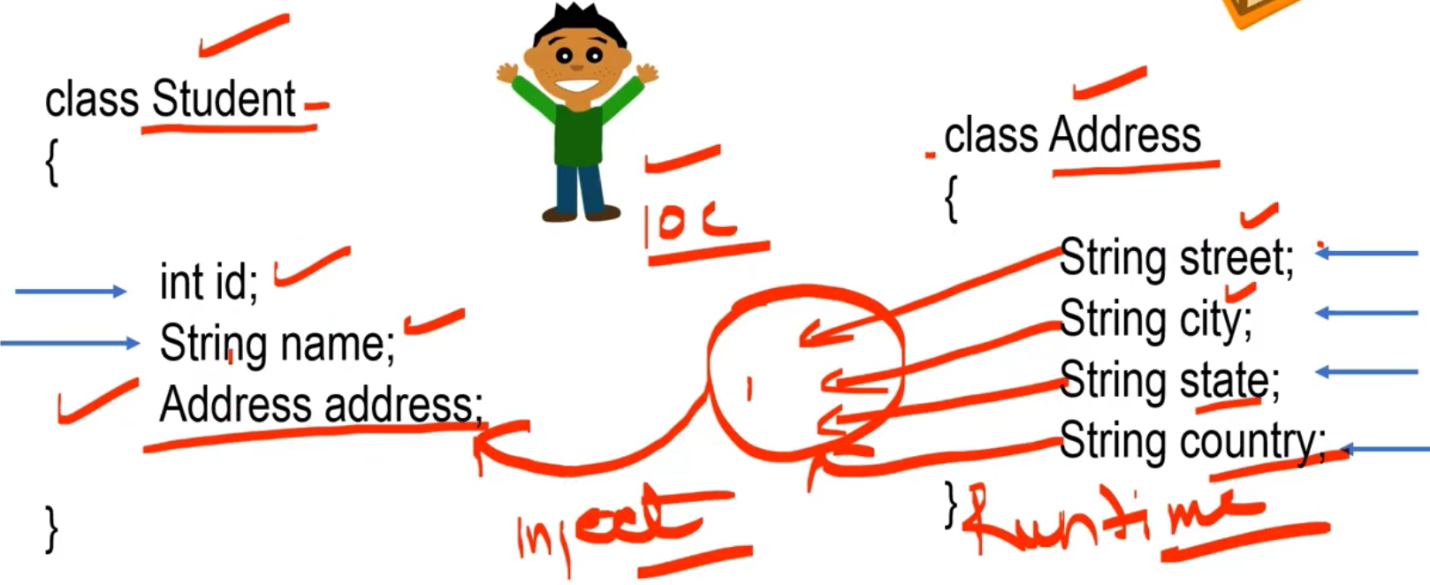
classpathXMLApplicationContext searches for XML configuration using JAVA classpath.

AnnotationConfigApplicationContext is used in annotations. Searches for desired Bean for the used annotation.

FileSystemXMLApplication searches for config file from file system.

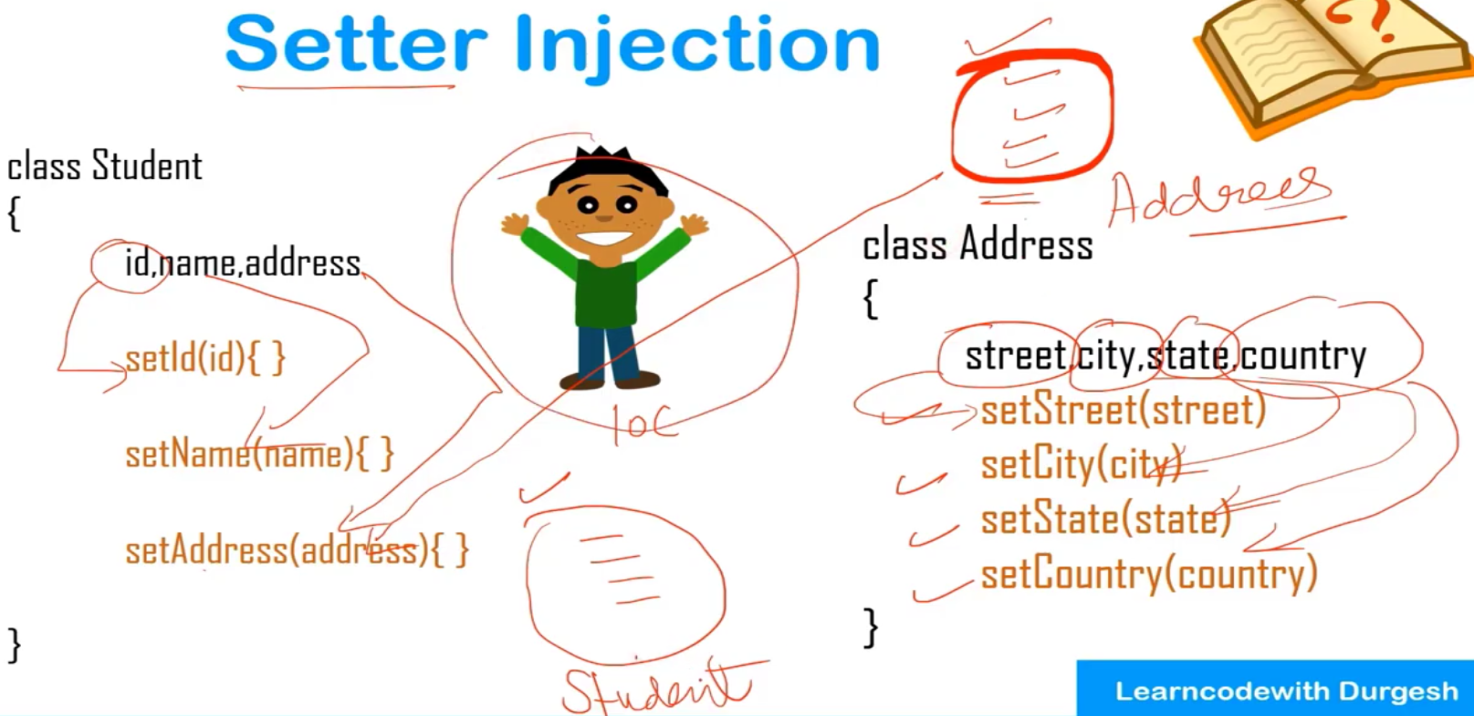
**How many Ways a Dependency can be Injected?**

Dependency Injection = Technique in which an object receives other object that it depends on. These other objects are called dependencies.

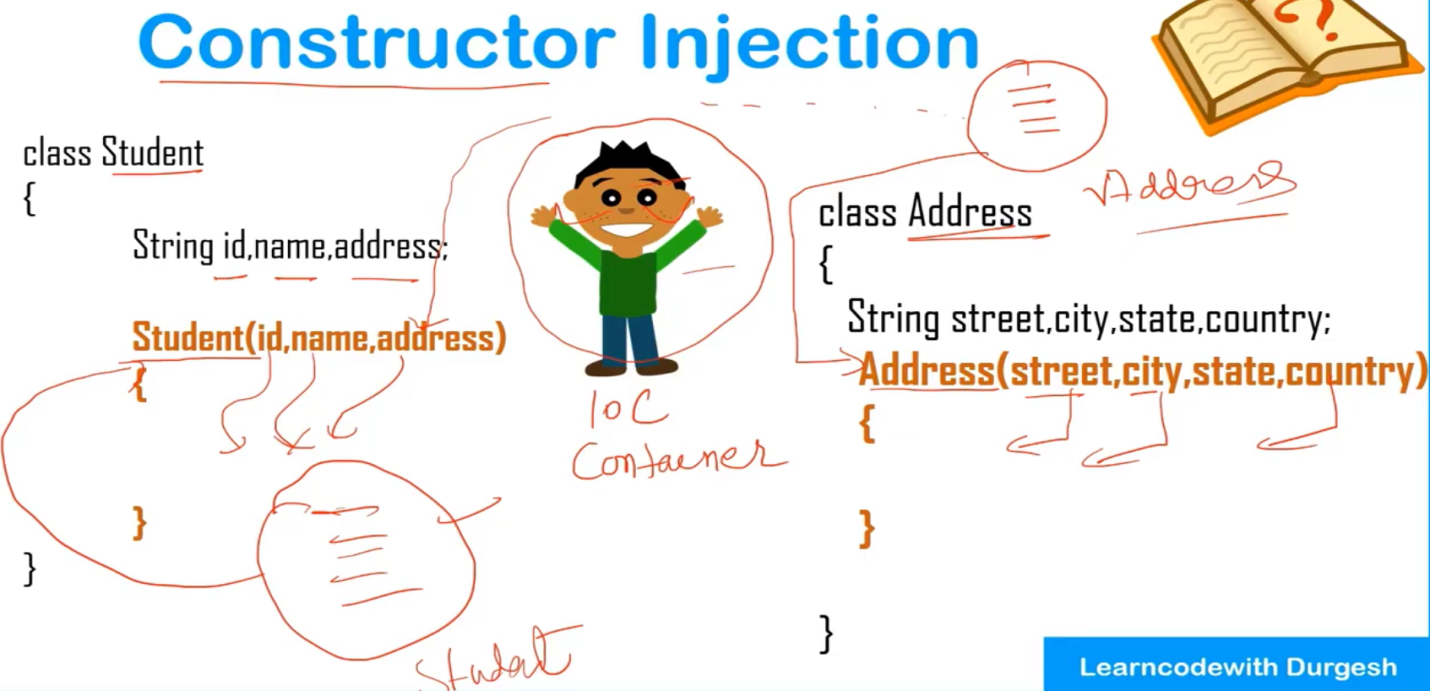


Since student depends on address, address is dependency and student is dependent class.

1. Using setter Injection. (Property Injection)

****

1. Using Constructor Injection.

****

Whichever method want to use, we will specify in config file.

Config file is an XML file where we declare the Beans and dependencies.

Beans are the java class which are injected in XML file(config file).

Config file looks like

*<beans>*

*<bean>--------</bean>*

*<bean>--------</bean>*

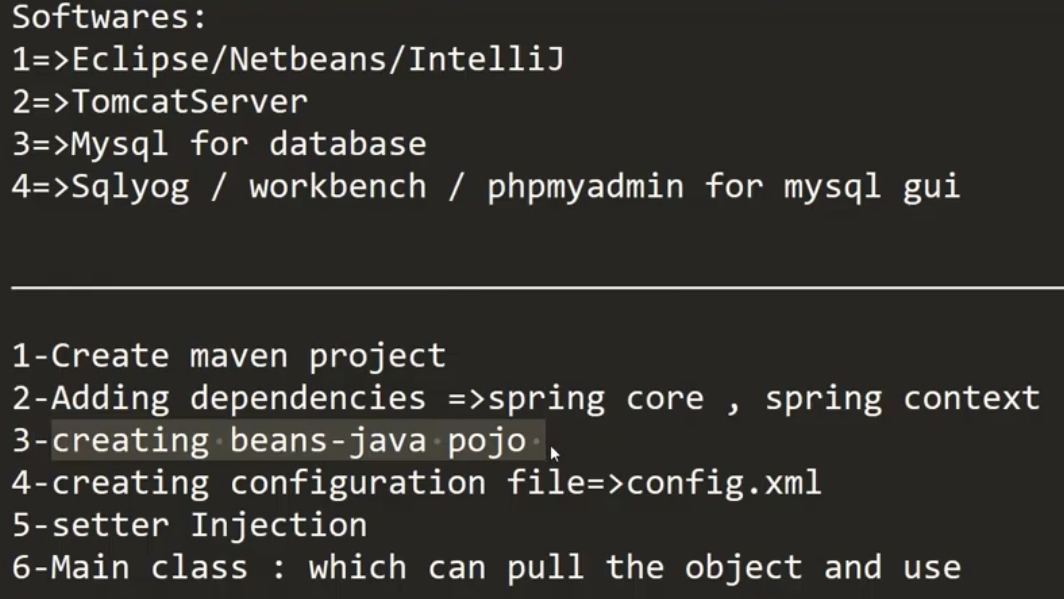
*<bean>--------</bean>*

*</beans>*

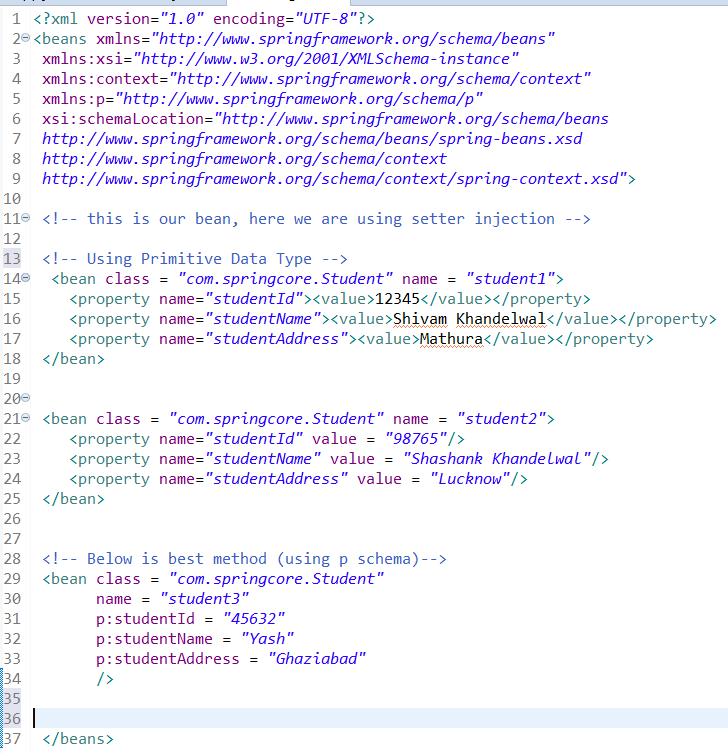
For every Bean, spring provide 2 methods. They are also called life methods.

1. public void **init()** : used for initializing code, Loading config, connecting db etc.
2. public void **destroy() :** used for writing code clean up.

We can change the name of methods but the signature should remain same.

****

**Injecting primitive data type in xml configuration file using setter injection.**

Below is theview of config.xml ****

P schema = Property schema

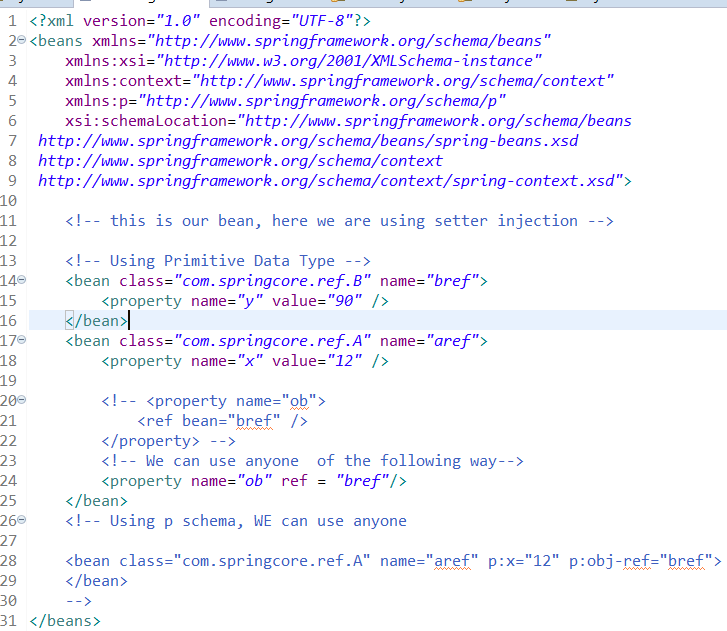
We can also use id = “student1” instead of name = “student1”

**Injecting Collection data type in xml configuration file using setter injection**

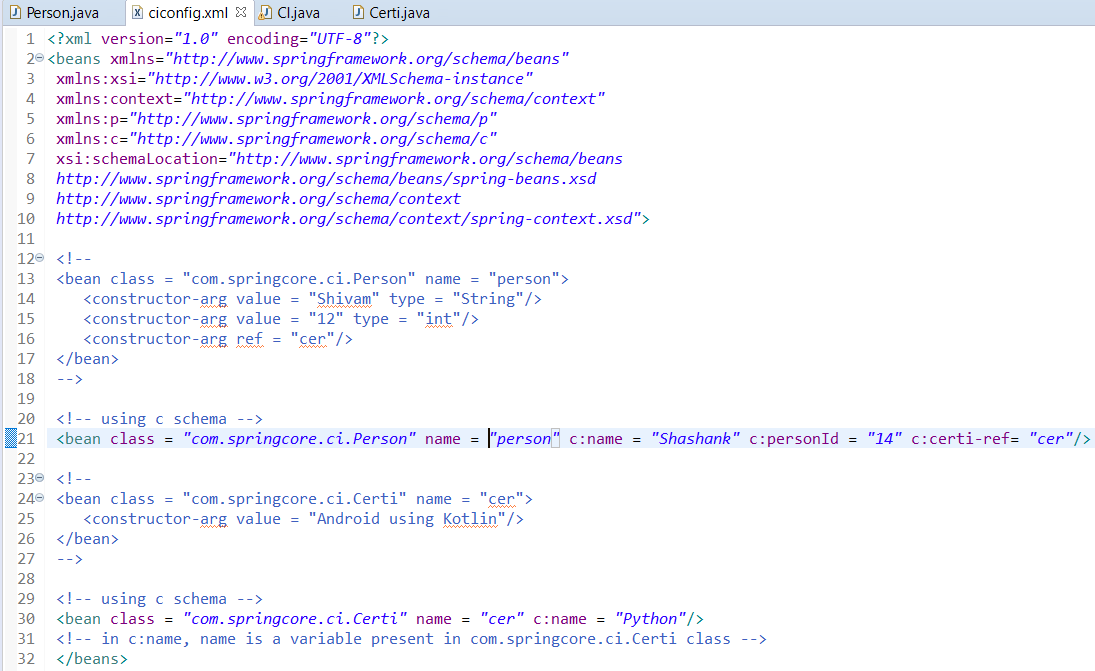
****

**Injecting Reference data type in xml configuration file using setter injection**

A.java depends on B.java. A.java has a variable named ob which is of type B.

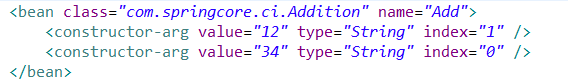
****

**Injecting Reference & primitive data type in xml configuration file using constructor injection**



**Ambiguity Problem in Constructor Injection**

To remove ambiguity we use **type** and **index** property



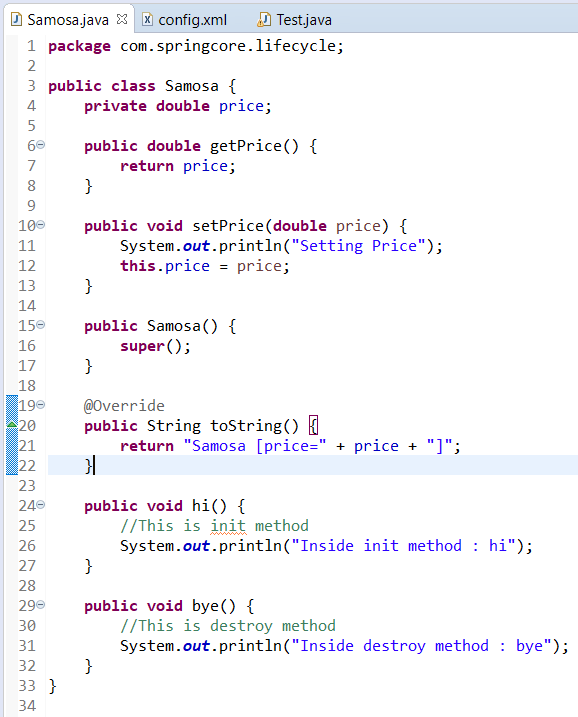
**LifeCycle Methods**

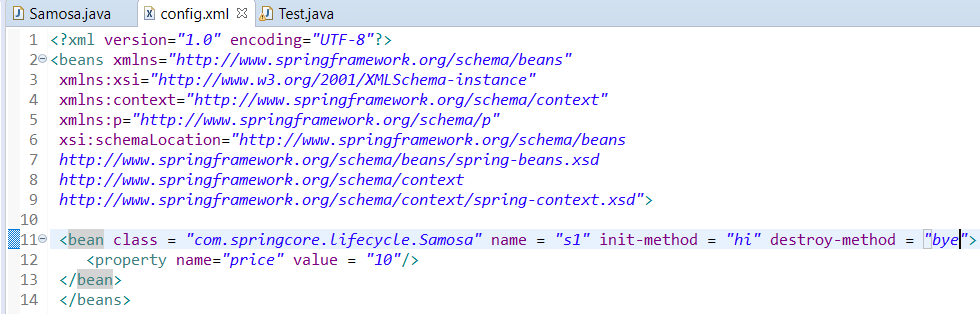
1. public void init()
2. public void destroy()

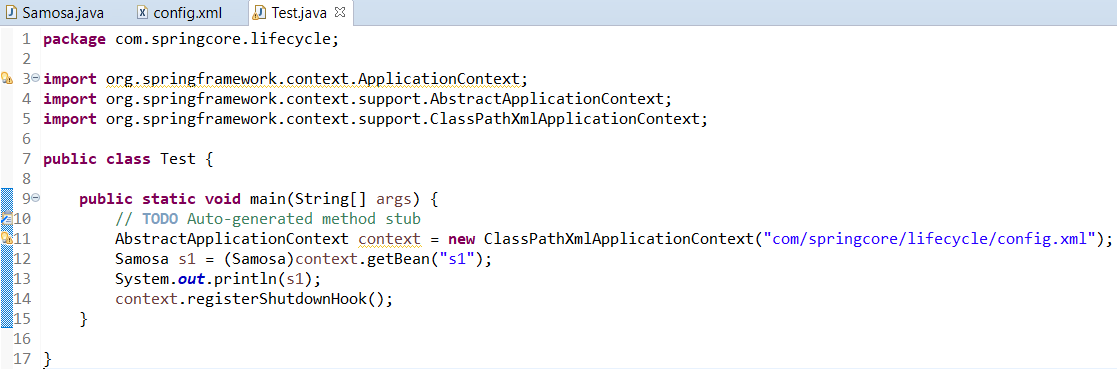
**Configuration Technique**

1. XML
2. Spring Interface.
3. Annotation.

**Configuration using XML**







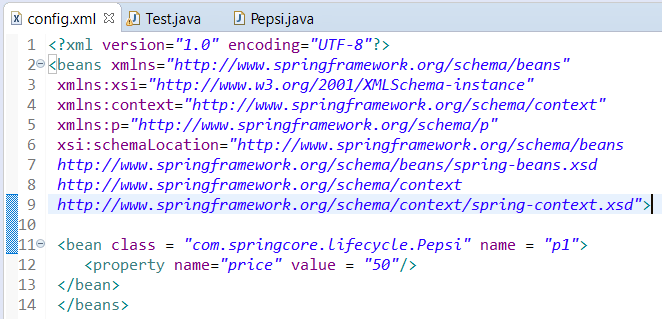
For calling destroy method we need to call another method context.registerShutdownHook()

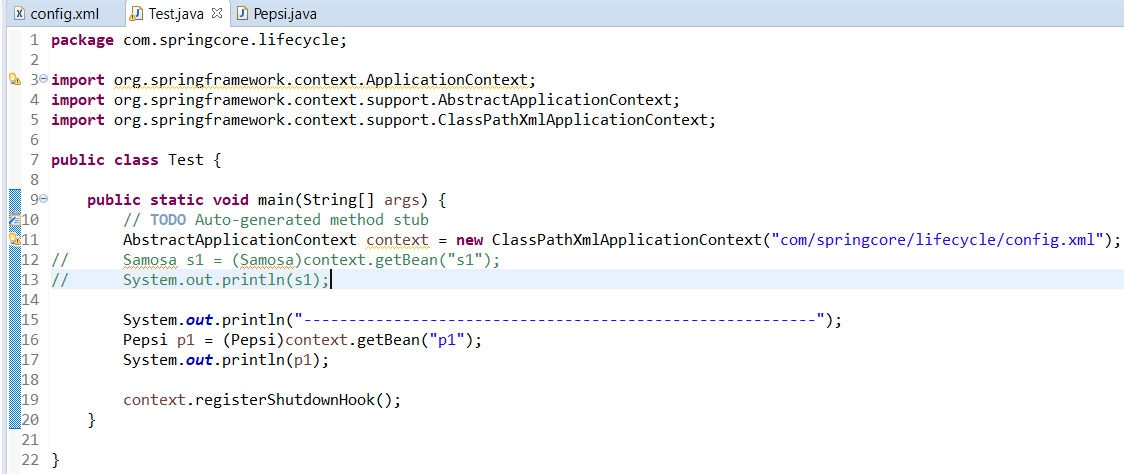
registerShutdownHook() is present with AbstractApplicationContext

**Configuration using Spring Interfaces**

We will use **InitilizatingBean** interface & **DisposableBean** Interface.

Note: registerShutdownHook() will be used here also.







**Configuration using Annotations**

We will use annotations like @PreConstruct, @PreDestroy.  
@PreConstruct will provide init functionality & @PreDestroy will provide destroy functionality.

registerShutdownHook() will be used here also.

Both these annotations are removed in java 9 and beyond. To use these annotations we have to add a dependency in POM.xml

*<dependency>*

*<groupId>javax.annotation</groupId>*

*<artifactId>javax.annotation-api</artifactId>*

*<version>1.3.2</version>*

*</dependency>*

By default these 2 annotations are disabled. We can enable these annotations using **<context:annotation-config/>** tag in our config file.

**<context:annotation-config/>** enables all annotations. If we want only the above mentioned 2 annotations then we will add a bean in config.xml

<bean class = “org.springframework.context.annotation.CommonAnnotationBeanPostProcessor”/>

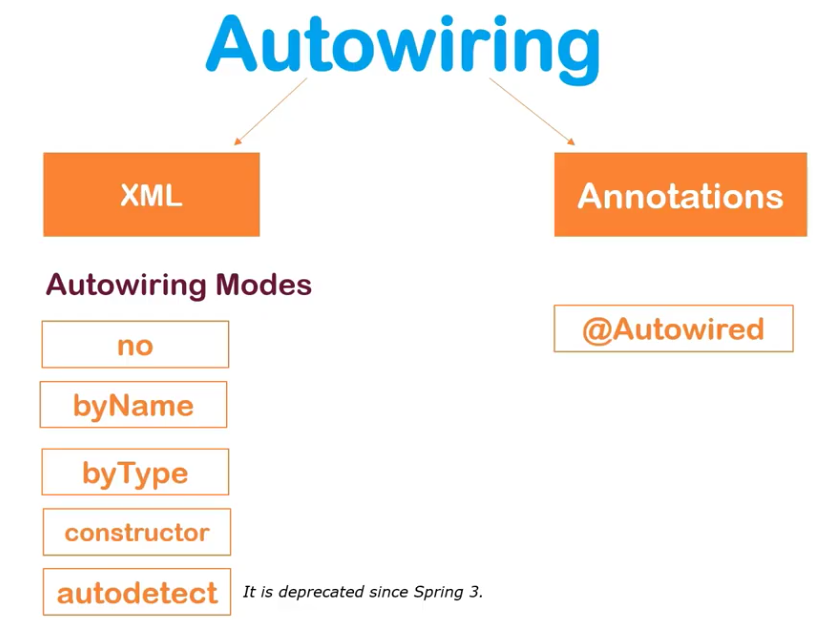
**AutoWiring in SPRING**

It is feature of SPRING in which spring container inject the dependency automatically.

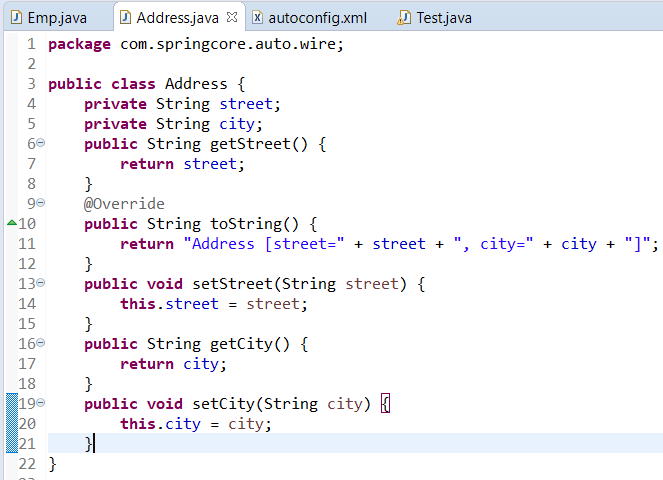
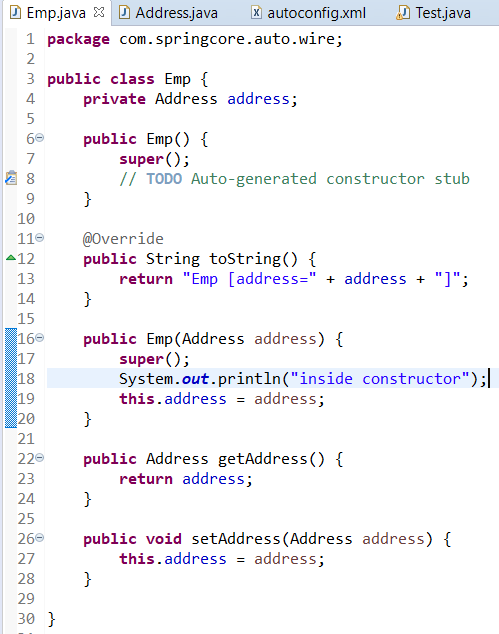
**AutoWiring can’t be used to inject primitive & string values**. It works with reference only. (eg Object)

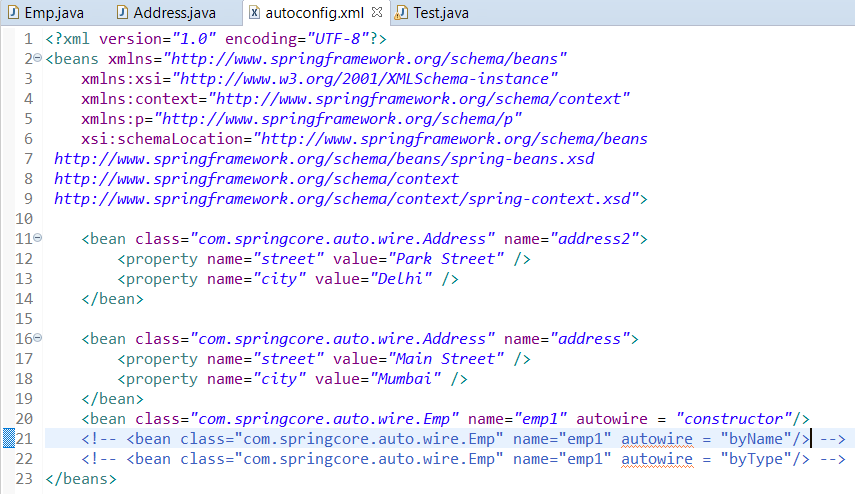
AutoWiring can be done using

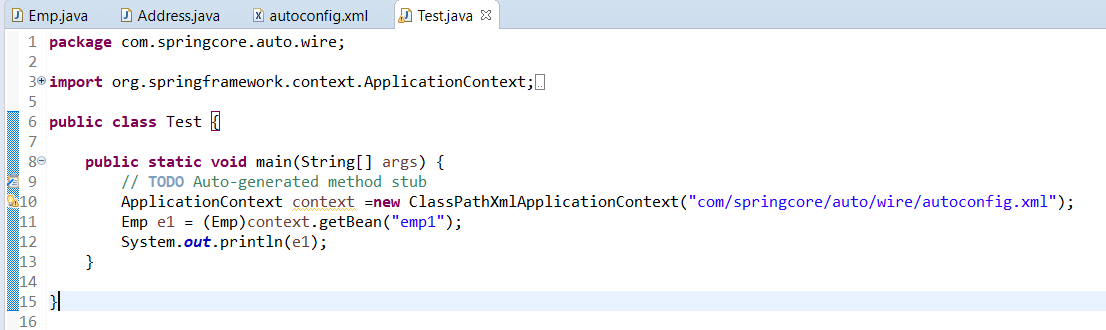
1. XML
2. Annotations (@AutoWired)



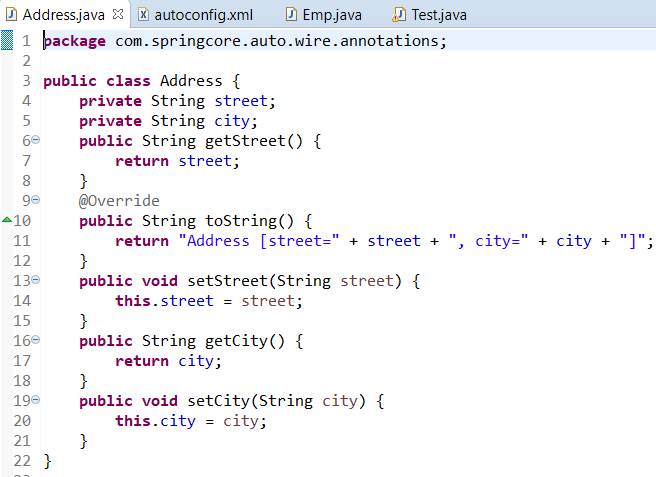
**Auto Wiring By XML**

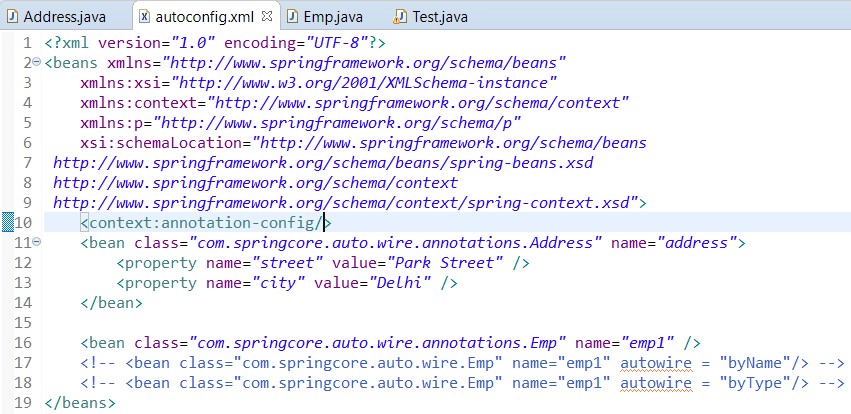


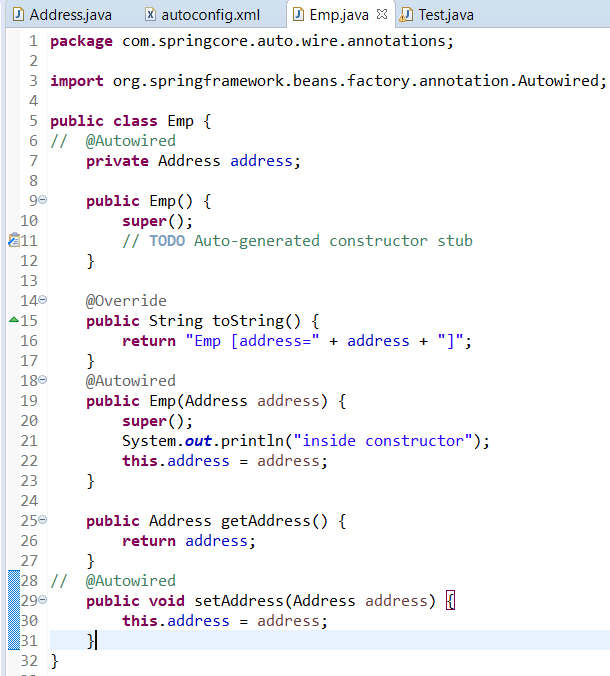


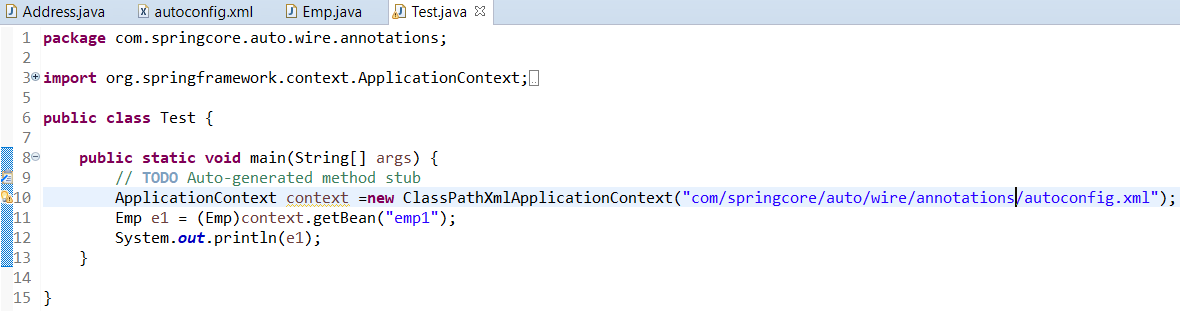


**Autowiring by Annotation**

@Autowired can be used with setter method, with a variable or with constructor.





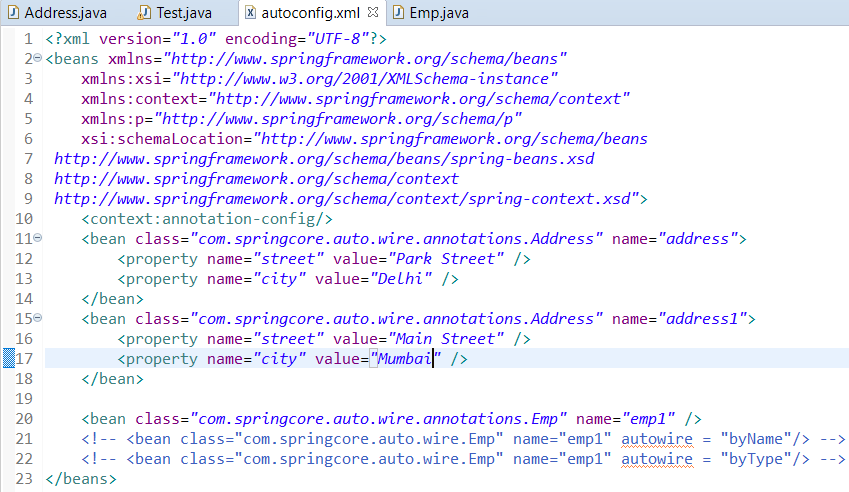


**@Qualifier() annotations**

@Qualifier(bean name)

It is used with @Autowired annotation.

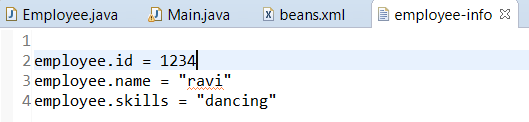
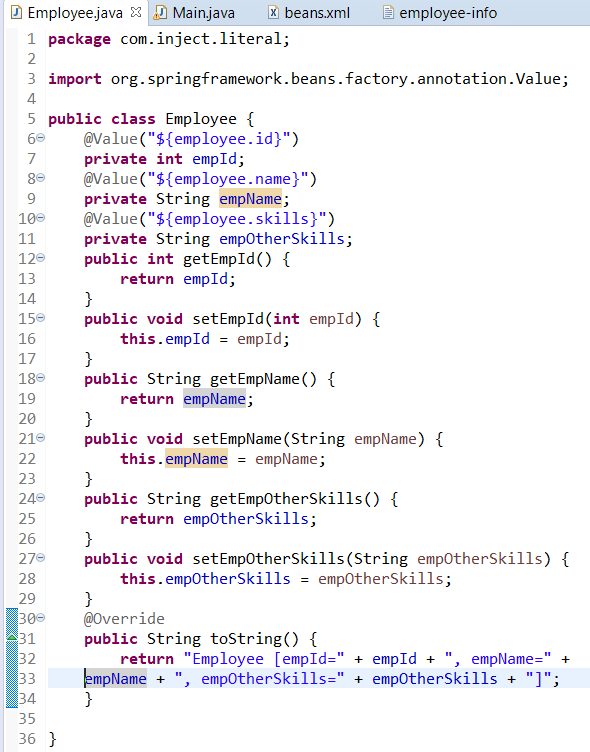
@Qualifier is used to specify the bean name.

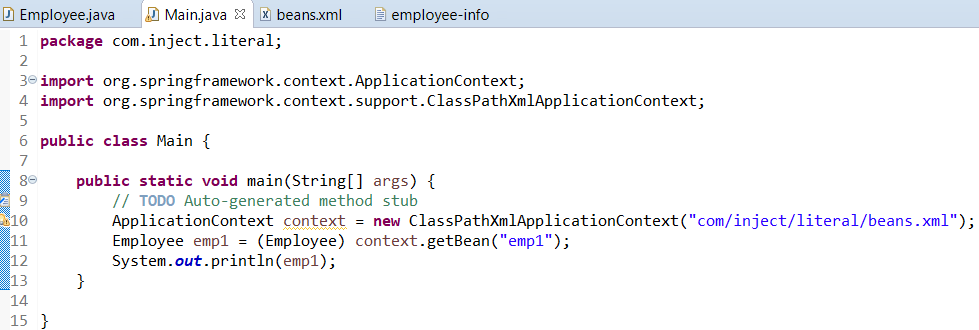


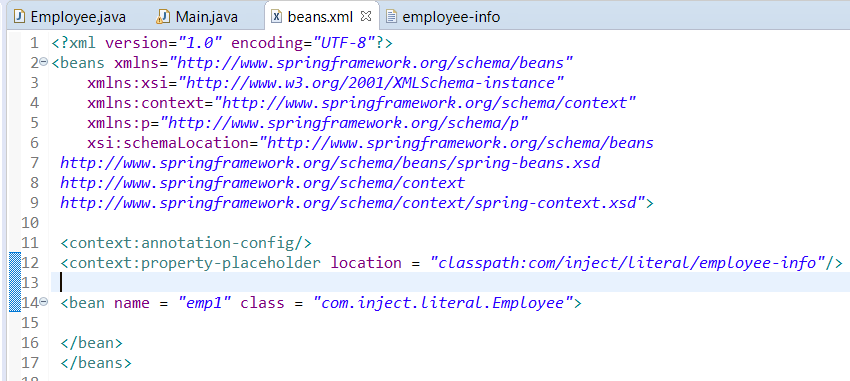


Address.java and test.java are same as in Autowired

**Injecting value from properties file Using @Value Annotation**



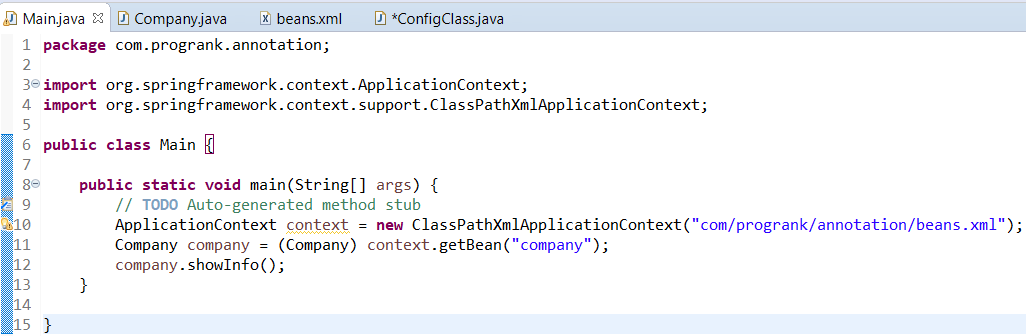


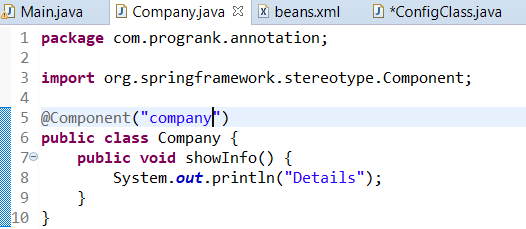


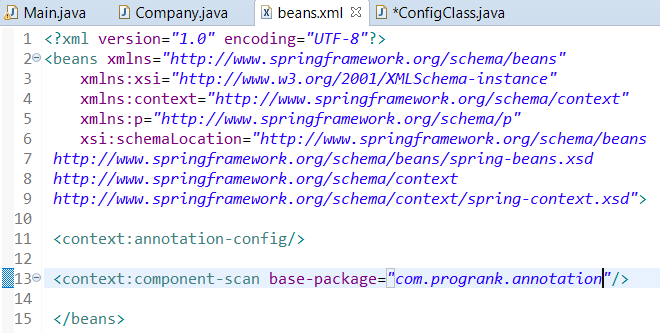
We need <context:property-placeholder> for injecting using @Values

**Injecting value without using <bean> in xml configuration file and using @Component Annotation**

We need to add <context:component-scan base-package=”package-name”/>. It activates @Component annotation.



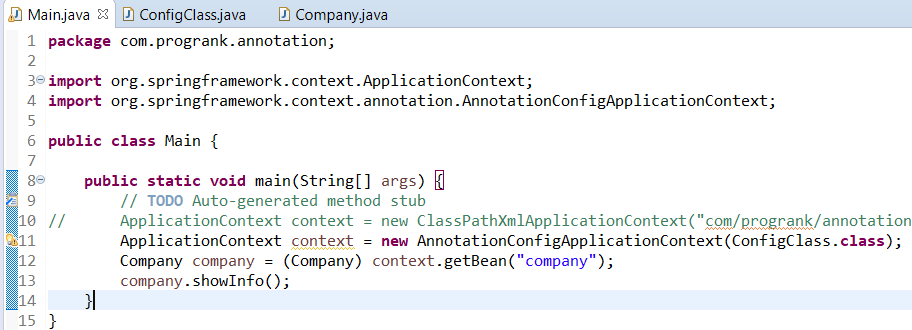


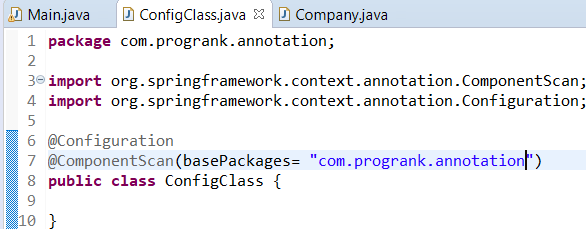
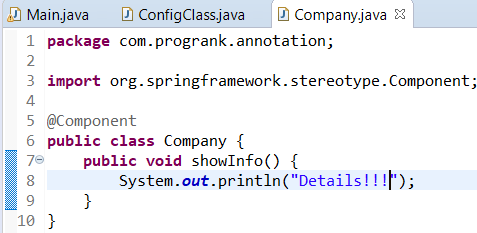


Note : We have not used <Bean> tag in xml file. Instead we have added @Component Annotation

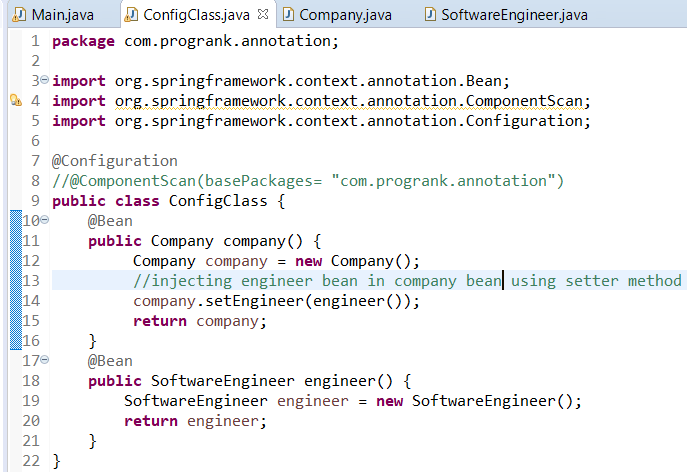
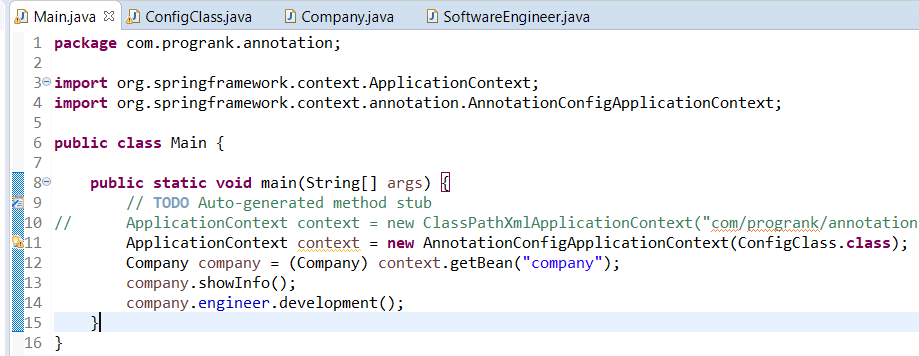
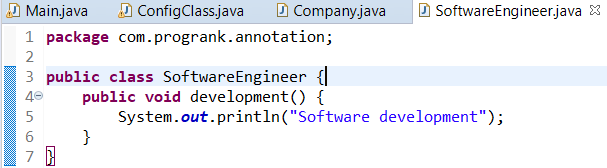
**Injecting value without xml configuration file and using @Configuration & @ComponentScan Annotation**

We will create a config class which will create bean and inject it where ever required

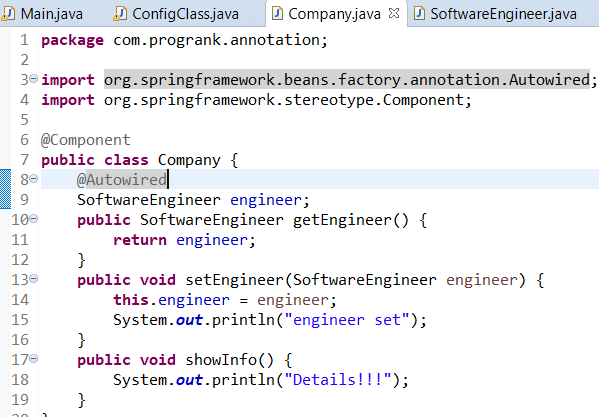
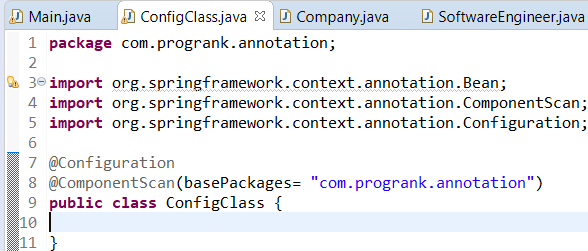


**Injecting Reference Dependency without using xml config file**

**Using @AutoWIred for injecting value**

Rest 2 classes(Main.java & SoftwareEngineer.java) remains same.

Here we are using only one dependency to inject, Suppose if we use 2 dependencies(Software Engineer & Hardware Engineer), then there will be an ambiguity which dependency to inject. To resolve this we use @Qualifier annotation. See the next screen shots. ConfigClass.java, main.java remains same.

