**Spring is a framework:**

**Spring Core**: Can be used with a desktop application

**Loose coupling**: When one layer/ class does not get affected by changes in another layer/class.

How do u achieve loose coupling: In plain java, we use coding to interfaces, we use the reference of the interface. But still we have to pass in the name of the implementation class to create the object.

In spring: spring provides us the container in the form of a library, which can create objects for the beans/ java classes and do the dependency injection

There are 2 ways of setting the object of B

1) Using the constructor to set the object

class A{

B b;

A(B b){

this.b= b;//b object is needed

}

}

Here, spring container will create the objects of B class, and it can also can also create the object of A

by the calling the constructor and passing this B object. It is doing dependency injection also i.e. injecting the object of B in A class

//**Constructor dependency injection**

2) Using the setter to set the dependency

class A{ //A needs B object, B is the dependency

B b;

public void setB(B b){//pass in the B object

this.b=b;

}

}

//Spring container will create B object, call the default constructor of class A, can create the object of A, if u tell spring to inject this B object into A class by calling the setter method, it can do that also.

//**Setter dependency injection**

Spring core module requires some **libraries** like: spring-core, spring-bean, spring-context, spring-context-support, spring-expression.

Till now we were downloading libraries and we were adding it to the project.

1) U have the remember all of the libraries you need to add..

2) If it is web application, we had to add these libraries to Deployment descriptor to ensure they were deployed to your server. Otherwise these libraries would not be packaged as part of ur war file.

3) If u forget to add any library to deployment descriptor , it will throw NoClassDefFoundError (i.e. class was present at the compile time but was not present at the time when you were running your application)

Class.forName("com.java.A")//means, A may not be present at compile time, we are trying to load the class dynamically at runtime. SO if A is not present at the classpath during the runtime, it will throw ClassNotFoundException (CheckedException)

To overcome these 3 major problems we are going to use a build tool. There are many build tools which are available: MAven, Ant, Gradle..

Maven: Maven works on the concept of global repository (https://mvnrepository.com/), local repository

(it will create a folder with the name .m2 on your hard disk and it will store the libraries you have used previously. First time u tell maven to download a library, it will download it from the global repository and it will store it in the .m2 folder. Next time u tell maven to use the same library, it will now get it from your local repository. So even if you are running your application on server and you have a internet connection, if the library is not present, it will download the libary from global repository.

Can prevent NoClassDefFoundError .

You need to add **dtd/ xsd to** your project to make sure only spring valid tags are added to spring configuration file.

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- bean definitions here -->

</beans>

The equivalent file in the XML Schema-style would be…​

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

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

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

xsi:schemaLocation="

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

<!-- bean definitions here -->

</beans>

If you do not specify the version for xsd, it will use the latest version available.

In spring, if for spring-core you are using 5.2 version, spring-context: u cannot use a diff version for it

Spring has given you a **container** in the form of a library. Used for **instantiating, configuring (scope, lifecycle callbacks, , and assembling the beans using the spring configuration file.**

For desktop application there are **2 spring containers** which are available

1) **BeanFactory**: Impl classes: XmlBeanFactory

BeanFactory factory= new XmlBeanFactory(new ClassPathResource("spring-config.xml"));

When the container starts, and u make a request to a bean: create the object, give u the object

2) **ApplicationContext**: It can be used for Internationalization, support for BeanPostProcessors etc..

Both of these containers are interfaces and to start the spring container, we need to create the object of the impl classes. ClassPathXmlApplicationContext, AnnotationConfigApplicationContext etc..

ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");

When the container starts, it will read ur config file,and create objects of all beans specified in config file as singleton scope.

ApplicationContext is a sub-interface of BeanFactory. It adds:

1) Easier integration with Spring’s AOP features

2) Message resource handling (for use in internationalization)

3) Event publication

4) Application-layer specific contexts such as the WebApplicationContext for use in web applications.

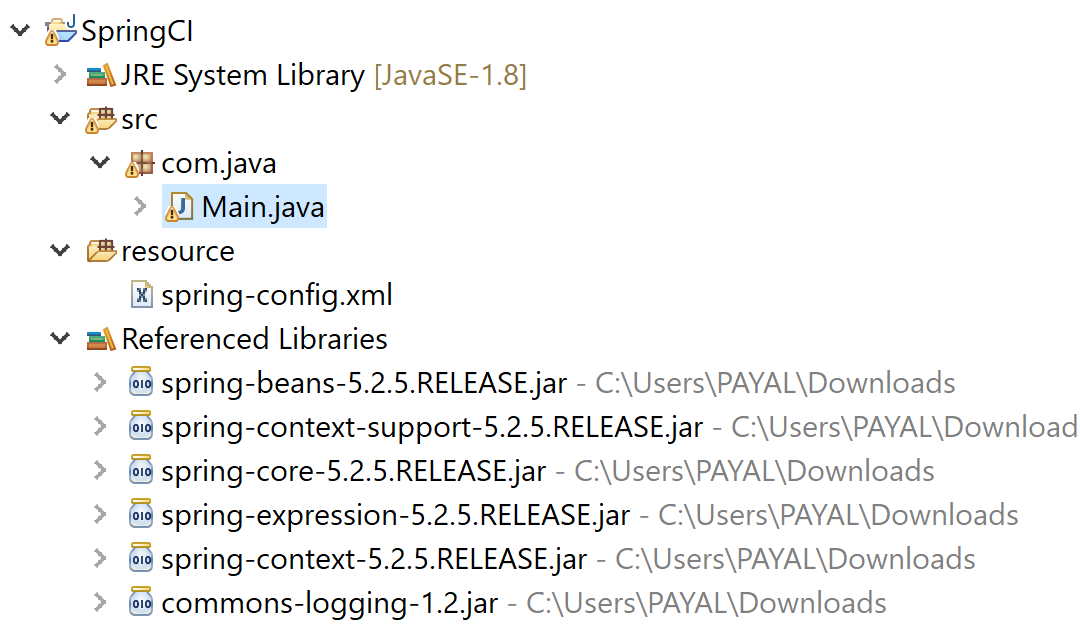
In short, the BeanFactory provides the basic functionality, and the ApplicationContext adds more enterprise-specific functionality

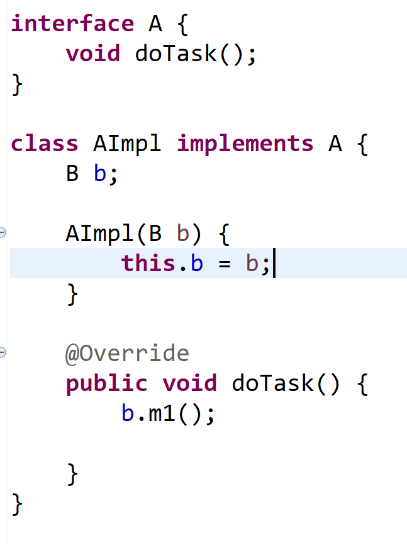
A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

**IoC (inversion of control) is also known as dependency injection (DI).** It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the **Service Locator pattern.**

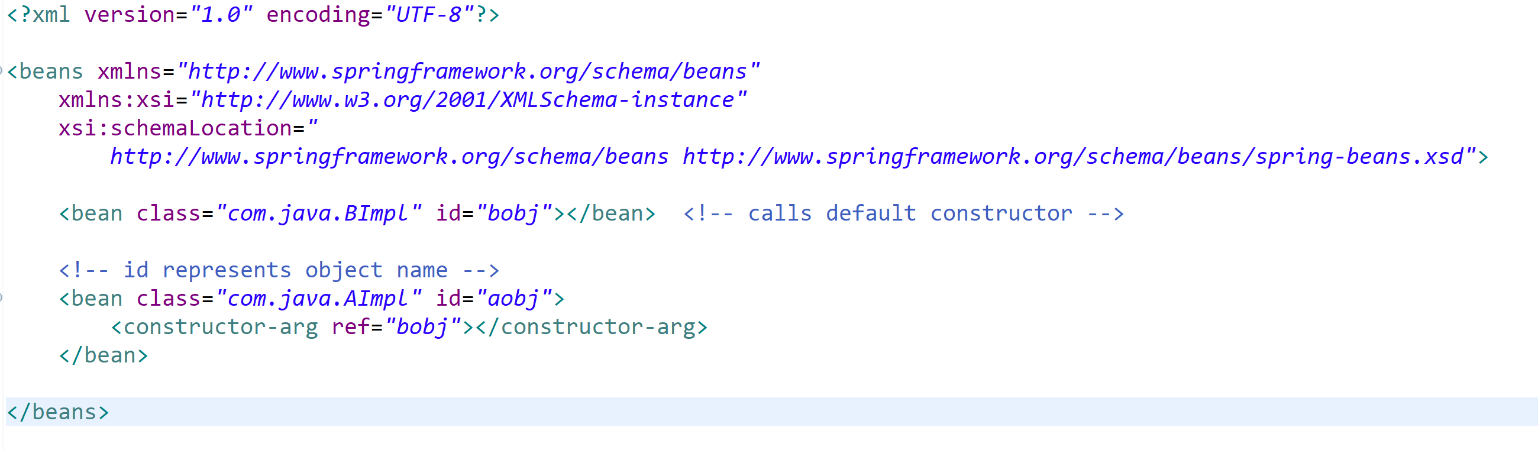
**Example 1:**

Dependencies:









For constructor injection: use constructor-arg

For setter injection: property

Tags in spring-config file:

1. Import: to import another xml config file

<import resource="services.xml"/>

file:C:/config/services.xml or classpath:/config/services.xml

1. Class: Mandatory unless using factory method or bean definition interface
2. Instantiating Beans:

Inner static class: if you have a class called SomeThing in the com.example package, and this SomeThing class has a static nested class called OtherThing, the value of the class attribute on a bean definition would be com.example.SomeThing$OtherThing.

Static factory method:

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

Instance factory method:

*<!-- the factory bean, which contains a method called createInstance() -->*

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

*<!-- the bean to be created via the factory bean -->*

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

**Constructor argument type**

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

**Constructor argument index**

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

**Constructor argument name**

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

1. Name: used to create alias for bean other than id. Can have them separated by comma or semi-colon or whitespace. Can have only 1 id but

Can also use <alias name="fromName" alias="toName"/> outside bean tag to create an alias.

In this case, a bean (in the same container) named fromName may also, after the use of this alias definition, be referred to as toName.

1. Naming Beans
2. Scope
3. Bean Scopes
4. Constructor arguments
5. Dependency Injection
6. Properties
7. Dependency Injection
8. Autowiring mode
9. Autowiring Collaborators
10. Lazy initialization mode
11. Lazy-initialized Beans
12. Initialization method
13. Initialization Callbacks
14. Destruction method

Use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the [@Required](https://docs.spring.io/spring/docs/5.2.5.RELEASE/spring-framework-reference/core.html#beans-required-annotation) annotation on a setter method can be used to make the property be a required dependency; however, constructor injection with programmatic validation of arguments is preferable. Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later

<bean id="theTargetBean" class="..." />

<bean id="client" class="...">

<property name="targetName" value="theTargetBean"/>

</bean>

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

<property name="email">

<null/>

</property>

<bean id="something" class="things.ThingOne">

<property name="fred.bob.sammy" value="123" />

</bean>

The something bean has a fred property, which has a bob property, which has a sammy property, and that final sammy property is being set to a value of 123. In order for this to work, the fred property of something and the bob property of fred must not be null after the bean is constructed. Otherwise, a NullPointerException is thrown.

<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />

To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute (commas, whitespace, and semicolons are valid delimiters):

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

**Singleton scope vs singleton class**

In Spring, we can define multiple bean tags in ur config file for a particular class, it will create that many objects. So it is no longer a singleton class, but each object is of a singleton scope. That means if u call getBean for same tag, it will always return you the same object.

Singleton class: A class for which we can only create 1 object

Class A{

}

<bean class=”A” id=”a1” >

<bean class=”A” id=”a2” scope=”singleton”>

Bean A objects a1, a2 are of singleton scope but A is not a singleton class.

A obj1=Factory.getBean(a1);

A obj2=Factory.getBean(a1);

A obj3=Factory.getBean(a2);

Obj1 is equal to obj2 but not equal to obj3

<bean class=”A” id=”a3” scope=”prototype”>

A obj1=Factory.getBean(a3);

A obj2=Factory.getBean(a3);

obj1==obj2: false

Bydefault the scope for every bean is singleton

**Inner Object**

Class A{

B b;

//setter for B

}

Class B{}

<bean class=”A” id=”a”>

<property name=”b>

<bean class=”B”></bean>

</property>

<bean>

This b object is an **inner object**, so it cannot be accessed elsewhere. It can only be accessed inside ur a object

<bean class=”A” id=”a2”>

<property name=”b” ref=”b”> //This is wrong

**Autowiring:**

Class A{

B b;

//setter for B

}

Class B{}

<bean class=”B” id=”b”>

<bean class=”A” id=”a” autowire=”none”>

<-- <property name=”b” ref=”b”> - ->

</bean>

**By-default it is autowire=none**. You need to manually inject the object of one class into the other using either ur property tag or ur constructor-arg.

2) **autowire=byType**

<bean class=”B” id=”b”>

<bean class=”A” id=”a” autowire=”byType”></bean>

Here container sees A has a property of type B, it will automatically inject Bean of type B in ur A object

Sysout(a.b)=> B object

3) **autowire=byName**

If you have more than one bean of same type, autowire=type will give no unique bean of type found. Expected 1 but found multiple. Then u should autowire by name.

<bean class=”B” id=”b1”>

<bean class=”B” id=”b”>

<bean class=”A” id=”a” autowire=”byType”></bean> //Runtime Exception

<bean class=”A” id=”a” autowire=”byName”></bean> //In this case will inject object with id b

4) **autowire=”constructor”**

This is used for Constructor injection

A(B b): inject type B

**Init / Destroy Method**

1. <bean class=”A” id=”a” init-method=”init1”/>

Class A{

A(){

Sysout(2);

}

Public void init1(){  
Sysout(1);

}

}

// 2 1

If we want init method in more than 1 bean: I can define it for all the beans

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

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

xsi:schemaLocation=*"*

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

default-init-method=*"init2"*>

If any bean has this method, it would work as a init method

Similarly u can define ur destroy methods

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

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

xsi:schemaLocation=*"*

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

default-init-method=*"init2"* default-destroy-method=*"d2"*>

<bean class=*"com.java.BImpl"* id=*"bobj"* destroy-method=*"d1"*></bean>

3)

Spring also provides u InitializingBean and DisposableBean interfaces to define ur init and destroy methods in java class

**public** **class** Test **implements** InitializingBean, DisposableBean{

Test(){

System.***out***.println(1);

}

@Override

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

System.***out***.println(2); //init

}

@Override

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

System.***out***.println(3);

}

}

4) Jdk provides u some annotations which can use to set ur method to be as the init/ destroy method

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**public** **class** Test {

Test() {

System.***out***.println(1);

}

@PostConstruct

**public** **void** i1() {

System.***out***.println(2);

}

@PreDestroy

**public** **void** d1() {

System.***out***.println(3);

}

}

If using BeanFactory, u have to register these annotations with spring bcoz benaFcatory does not provide support for BeanPostProcessor

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

Activity 1

U have to use all the types of init method, destroy method:

Constructor

Init-method, @PostConstruct, Initializing, default-init-method

Which gets executed first/ later

Usig applicationContext:

ApplicationContext factory= **new** ClassPathXmlApplicationContext("spring-config.xml");

AImpl obj = factory.getBean("aobj", AImpl.**class**);

obj.doTask();

But these do not have close method

So instead you can use ur AbstractApplicationContext

**public** **class** Main {

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

AbstractApplicationContext factory= **new** ClassPathXmlApplicationContext("spring-config.xml");

AImpl obj = factory.getBean("aobj", AImpl.**class**);

obj.doTask();

factory.**registerShutdownHook(**);// it tells spring to shutdown ur container: destroy all bean object

}

}

//destroy methods can only be seen getting invoked here

**BeanFactory does not provide support for BeanPostPorcessors**

beanPostProcessor: CommonAnnotationBeanPostProcessor

When ur bean is created, it will check for annotations in class using java annotations. It will enable it.

Activity:

**class** B **implements** BeanPostProcessor{

@Override

**public** java.lang.Object postProcessBeforeInitialization(java.lang.Object bean, java.lang.String beanName) {

System.***out***.println(bean + "object getting created " );

//validatios

**return** bean;

}

@Override

**public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {

System.***out***.println(beanName+" got created");

**return** bean;

}

}

Add these to ur init method code and find the sequence:

**Bean lifecycle**

Constructor

PostProcessBeforeInitialization

afterPropertiesSet: InitializingBean

init-method

PostProcessAfterInitialization

//bean methods

Disposable bean

Destroy method

**Factory methods: instance/ static**

Method which is being used to return the object : Factory method

1. Instance
2. Static: when it is returning the object of the same class

Class A{

A obj= new A();

public static A getInstance(){

return obj;

}}

<bean class=”A” id=”a”/>//calling constructor

<bean class=”A” factory-method=”getInstance”/> //call ur static factory method with the name getInstance in A class

Eg2:

**class** A{

List<B> pool= **new** ArrayList<>();//pool of b objects

{

**for**(**int** i=0; i<20; i++)

pool.add(**new** B());

}

**public** B getB(){ //instance factory method

**if**(!pool.isEmpty())

**return** pool.get(0);

**return** **null**;

}

}

**class** B{

}

<bean class=”B” id=”obj”/>//constructor of B, not valid

<bean class=”A” id=”a”/>

<bean factory-bean=”a” factory-method=”getB” id=”b” name=”b1”/>

<alias name=”b” alias=”b2”/>

So b can be referred even using b1 & b2.

//since getB is a instance method, so using object of A to call this method.

**Java Configuration**

We were using xml to configure you bean (object) which are being managed by spring

XML is an old way of doing configuration

**class** A{

}

@Configuration

**public** **class** SpringConfig {

@Bean

@Scope("prototype")

**public** A getA() {

**return** **new** A();

}

}

Q)

@Configuration

**public** **class** SpringConfig {

@Bean

//@Scope("prototype")

**public** A getA() {

A obj= **new** A(getB());//new B();

obj.setX(1);

obj.setY(2);

**return** obj;

}

@Bean

**public** B getB() {

**return** **new** B();

}

}

**class** A{

B b;//CI

**int** x;//setter

**int** y; //setter

A(B b){

**this**.b= b;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

**class** B{

}

**public** **class** Main {

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

AbstractApplicationContext ctx = **new** **AnnotationConfigApplicationContext**(SpringConfig.**class**);

ctx.getBean(A.**class**);

}

}

Q)

@Configuration

**public** **class** SpringConfig {

@Bean(initMethod = "init", destroyMethod = "dest", ~~autowire~~ = Autowire.***BY\_TYPE***)

**public** A getA() {

A obj = **new** A();// new B();

obj.setX(1);

obj.setY(2);

**return** obj;

}

@Bean(autowireCandidate = **false**)

**public** B getB1() {

**return** **new** B();

}

@Bean

**public** B getB2() {

**return** **new** B();

}

}

**class** A {

B b;// CI

**int** x;// setter

**int** y; // setter

**public** **void** setB(B b) {

**this**.b = b;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

**class** B {

}

Importing configuration

@Configuration

@Scope("prototype")

**public** **class** SpringConfig {

@Bean(initMethod = "init", destroyMethod = "dest", ~~autowire~~ = Autowire.***BY\_TYPE***)

**public** A getA() {

A obj = **new** A();// new B();

obj.setX(1);

obj.setY(2);

**return** obj;

}

}

@Configuration

**class** SpringConfig2{

@Bean(autowireCandidate = **false**)

**public** B getB1() {

**return** **new** B();

}

@Bean

**public** B getB2() {

**return** **new** B();

}

}

**class** A {

B b;// CI

**int** x;// setter

**int** y; // setter

**public** **void** setB(B b) {

**this**.b = b;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

**class** B {

}

**public** **class** Main {

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

AbstractApplicationContext ctx = **new** AnnotationConfigApplicationContext(SpringConfig.**class**,SpringConfig2.**class**);

ctx.getBean(A.**class**);

}

}

OR

@Configuration

@Scope("prototype")

**@Import(SpringConfig2.class)**

**public** **class** SpringConfig {

@Bean(initMethod = "init", destroyMethod = "dest", ~~autowire~~ = Autowire.***BY\_TYPE***)

**public** A getA() {

A obj = **new** A();// new B();

obj.setX(1);

obj.setY(2);

**return** obj;

}

}

@Configuration

**class** SpringConfig2{

@Bean(autowireCandidate = **false**)

**public** B getB1() {

**return** **new** B();

}

@Bean

**public** B getB2() {

**return** **new** B();

}

}

**public** **class** Main {

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

AbstractApplicationContext ctx = **new** AnnotationConfigApplicationContext(SpringConfig.**class**);

ctx.getBean(A.**class**);

}

}

OR

@Configuration

@Scope("prototype")

@Import(SpringConfig2.**class**)

**@ImportResource(locations = "spring-config.xml")**

**public** **class** SpringConfig {

@Bean(initMethod = "init", destroyMethod = "dest", ~~autowire~~ = Autowire.***BY\_TYPE***)

**public** A getA() {

A obj = **new** A();// new B();

obj.setX(1);

obj.setY(2);

**return** obj;

}

}

@Configuration

**class** SpringConfig2{

@Bean(autowireCandidate = **false**)

**public** B getB1() {

**return** **new** B();

}

@Bean

**public** B getB2() {

**return** **new** B();

}

}

In xml, to import other xml:

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

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

xsi:schemaLocation=*"*

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

default-init-method=*"init2"* default-destroy-method=*"d2"*>

**<import resource=*"spring-config2.xml"*/>**

<bean class=*"com.java.A"* id=*"a1"*/>

<bean class=*"com.java.A"* id=*"a2"*/>

</beans>

**OR**

**public** **class** Main {

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

AbstractApplicationContext ctx = **new** ClassPathXmlApplicationContext("spring-config.xml","spring-config2.xml");

ctx.getBean(A.**class**);

}

}

**Annotation based configuration:**

1. Stereotype annotations: To tell spring that u need the object of these classes, or to configure them as a bean in spring-context
   1. @Component
   2. @Repository
   3. @Service
   4. @Controller

<bean.> @Bean

@Component

Class A{

}

It will create the object of class A with id= a. But u can always define ur own name

@Component(“a1”)

In your project, you many have many classes, and many packages. SO you have to tell spring in which package to look for ur stereotype annotations

@Configuration

@ComponentScan(basePackage=”com.java”)

Class SpringConfig

{

}

Example:

@Configuration

@Scope("prototype")

@Import(SpringConfig2.**class**)

@ImportResource(locations = "spring-config.xml")

**public** **class** SpringConfig {

/\*

\* @Bean(initMethod = "init", destroyMethod = "dest", autowire =

\* Autowire.BY\_TYPE) public A getA() { A obj = new A();// new B(); obj.setX(1);

\* obj.setY(2); return obj; }

\*/

}

@Configuration

@ComponentScan(basePackages = "com.java")

**class** SpringConfig2{

/\*

\* @Bean(autowireCandidate = false) public B getB1() { return new B(); }

\*

\* @Bean public B getB2() { return new B(); }

\*/

}

@Component

**class** A {

@Autowired //does not need constructor/ setter

B b;// CI

@Value("1")

**int** x;// setter

@Value("1")

**int** y; // setter

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

@Component

**class** B {

}

**public** **class** Main {

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

AbstractApplicationContext ctx= **new** AnnotationConfigApplicationContext(SpringConfig2.**class**);

}

}

Example2:

@Configuration

@ComponentScan(basePackages = "com.java")

**class** SpringConfig2{

@Bean(name = "a2", ~~autowire~~ = Autowire.***BY\_TYPE***)

**public** A getA() {

A obj= **new** A();

obj.setX(2);

obj.setY(2);

**return** obj;

}

}

@Component("a")

**class** A {

@Autowired //does not need constructor/ setter

B b;// CI

**public** **void** setB(B b) {

**this**.b = b;

}

@Value("1")

**int** x;// setter

@Value("1")

**int** y; // setter

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

}

@Component

**class** B {

}

@Autowired bydefualt is done byType, but if multiple beans of same type, uses byName.

If bean not found with that property name, you should use @Qualifier (“beanName”), specifying the name of the bean you want to inject.

If you have one constructor and it takes in another object as an argument, there is no need to explicitly use @Autowired

**Depends-on**

maven