Coupling :-

Coupling is nothing but the dependency of one class on the other. If one object in a code uses the other object in the program, it is called loose coupling in Java. In coupling, two classes or objects collaborate and work with each other to complete a pre-defined task.

**1) Loose Coupling in Java**

When two classes, modules, or components have low dependencies on each other, it is called loose coupling in Java. Loose coupling in Java means that the classes are independent of each other. The only knowledge one class has about the other class is what the other class has exposed through its interfaces in loose coupling

class Volume {

   public static void main(String args[]) {

        Cylinder b = new Cylinder(25, 25, 25);

           System.out.println(b.getVolume());

   }

}

final class Cylinder {

    private int volume;

    Cylinder(int length, int width, int height) {

             this.volume = length \* width \* height;

    }

    public int getVolume() {

             return volume;

    }

}

### Tight Coupling

When two classes are highly dependent on each other, it is called tight coupling. It occurs when a class takes too many responsibilities or where a change in one class requires changes in the other class. In tight coupling, an object (parent object) creates another object (child object) for its usage.

class Volume {

   public static void main(String args[]) {

        Cylinder b = new Cylinder(15, 15, 15);

           System.out.println(b.volume);//volume class depend on cylinder class if we make it

   }} // volume private then we cant access volume

 class Cylinder {

   public int volume;

   Cylinder(int length, int width, int height) {

           this.volume = length \* width \* height;  }}

Spring

UI Layer----------------🡪Service Layer--------------🡪Dao Layer-----🡪Database(MYSQL/MongoDB)

ProductController<--------ProductServic<-----------ProductDao<---------MySQL

Dependency injection 🡪

dependency injection (DI) is the process of supplying a resource that a given piece of code requires. The required resource, which is often a component of the application itself, is called a dependency.

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.

Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

Inversion of Control :Predefined container

container --manages life cycle of spring beans

(spring bean --- java obj whose life cycle completely managed by SC(spring container)

It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML)

**Two Types of IOC Container:-**

**1.Bean Factory 2.ApplicationContext**

| Feature | BeanFactory | ApplicationContext |
| --- | --- | --- |
| Annotation Support | No | Yes |
| Bean Instantiation/Wiring | Yes | Yes |
| Enterprise Services | No | Yes |
| ApplicationEvent publication | No | Yes |
| Automatic BeanPostProcessor  registration | No | Yes |
| Loading Mechanism | Lazy loading | Aggressive loading |

Both BeanFactory and ApplicationContext are Interface implement by XMLBeanFactory and ClassPathXMLApplicationContext class respectively

**1:BEAN FACTORY**

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

Syntax :

Resource resource=**new** ClassPathResource("applicationContext.xml");

BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

#### 2.ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

Syntax :

ApplicationContext context =

**new** ClassPathXmlApplicationContext("applicationContext.xml");

This applicationContext.xml file will help to manage all bean

<beans>

<bean class=”{Path}” name=”object name”>

</bean>

</beans>

Dependency injection happen via 2-way :Constructor based and setter based

Setter Based Injection :

<bean id=obj class=”com.app.ClassName>

<property name=”mem\_var”>

<value>value</value>

</property>

</bean>

Constructor Based Injection :

<bean id=”obj” class=”com.app.ClassName”>

<constructor-arg value=”value” type=”type” />

</bean>

If we do not provide type bydefault it assigned value to string type of variable,if we have 2 constructor having int and string datatype respectively or we do not provide type of constructor then it call string type constructor causes ambiguity problem

Difference Between Setter based and Constructor based

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

**AutoWiring** :

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Syntax--------- ***autowire=”Type”***----where you want to inject dependencies

Types :

**No**: It is the default autowiring mode. It means no autowiring bydefault.

**byName**: it check bean name in xml file and class ref name if it found matches then it autowire,property name and bean name must be same

**byType**: it check type of ref and type of bean if it found similarity then it autowire,Type of property and Type of bean must be same.

**constructor**:it check constructor in bean and ref but if having multiple constructor then it goes with higher argument constructor

*Ambiguity Error in constructor injection :-*

*Some cases multiple constructor present with same number of argument but differnt types but while configuring bean if you did not mention type of value then bydefault it take string to overcome this ambiguity you need to mention type of argument in constructor*

**LifeCycle of SpringBean :-**

[*https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/*](https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/)

**1)ioc container start------🡪2)bean instance created---🡪3)Dependencies injected-------->4)init method----🡪5)utility method--🡪6)Destroy method**

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.

**We can customize this init() and destroy() method to execute some customize code**

**Two-ways –** 1.by XML approach 2.by Programmatic approach 3.By Annotations

1)XML Approach :- init-method=”method name” and destroy-method=”method name” attribute used in XML while defining bean.

2)Programmatic Approach:- To provide the facility to the created bean to invoke custom **init()** method on the startup of a spring container and to invoke the custom **destroy()** method on closing the container, we need to implement our bean with two interfaces namely **InitializingBean**, **DisposableBean**

3)By Annotations : To provide the facility to the created bean to invoke custom **init()** method on the startup of a spring container and to invoke the custom **destroy()** method on closing the container, we need annotate **init()** method by **@PostConstruct** annotation and **destroy()** method by **@PreDestroy** annotation.

Spring Bean Scope :

**Bean Scopes** refers to the lifecycle of Bean that means when the object of Bean will be instantiated, how long does that object live, and how many objects will be created for that bean throughout. Basically, it controls the instance creation of the bean and it is managed by the spring container.

Types

**Singleton**

**Prototype**

**Request**

**Session**

**Application**

**Singleton :**

**scope=”singleton”**

 Only one instance will be created for a single bean definition per Spring IoC container and the same object will be shared for each request made for that bean.

<bean> one instance

<bean>

**Prototype:**

**Scope=”prototype”**

A new instance will be created for a single bean definition in IOC every time a request is made for that bean.

<bean> one instance

<bean> 2nd Instance

**Request :** A new instance will be created for a single bean definition every time an HTTP request is made for that bean. But Only valid in the context of a web-aware Spring ApplicationContext.

**Session:** Scopes a single bean definition to the lifecycle of an HTTP Session. But Only valid in the context of a web-aware Spring ApplicationContext.

**Application:** When beans are application scoped, the same instance of the bean is shared across multiple servlet-based applications running in the same ServletContext, while singleton scoped beans are scoped to a single application context only.

**Annotations Covered :**

@Postconstruct :this annotation used over init method in lifecycle of bean which executed before ioc container create instance of bean

@Predestroy:This annotation used over destroy method if user want excuted some code before destroying of bean instance

*With This annotation no need to configure init and destroy method in XML file*

@Component: @Component is an annotation that **allows Spring to automatically detect our beans**. In other words, without having to write any explicit code, Spring will: Scan our application for classes annotated with @Component. Instantiate them and inject any specified dependencies into them.

@Value: It is commonly used for injecting values into configuration variables

**Spring Expression Language** : The Spring Expression Language (SpEL for short) is a powerful expression language that supports querying and manipulating an object graph at runtime

Syntax : -

@Value(“#{expression}”)

Example

@Value(“#{8>7?8:7}”)

int data;

@Value(“#{class,constructor,reference,primitive data}”)

**Spring IOC Configuration without XML : Total Annotation based configuration**

**@Configuration:** One of the most important annotations in spring is **@Configuration annotation which indicates that the class has @Bean definition methods**. So Spring container can process the class and generate Spring Beans to be used in the application. This annotation is part of the spring core framework.

**@ComponentScan:** **@ComponentScan which is used along with the @Configuration annotation to specify the packages that we want to be scanned.** @ComponentScan without arguments tells Spring to scan the current package and all of its sub-packages.

Syntax : @ComponentScan(basePackages = "com.app")

So configuration class search base package class for bean

**@Autowire :** The @Autowired annotation marks a Constructor, Setter method, Properties and Config() method as to be autowired that is ‘injecting beans'(Objects) at runtime by Spring Dependency Injection mechanism.**@Autowired** annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform **byType** autowiring on the method.

**@Lazy(“true”):** The annotation delaying loading of object until we demand it,simply it delaying from instancetiation until we demand it