

Spring & SpringBoot Framework

By –

Dilip Singh

 **dilipsingh1306@gmail.com**

 **dilipsingh1306**

Spring Core Module

Before starting with Spring framework, we should understand more about **Programming Language vs Framework**. The difference between a programming language and a framework is obviously need for a programmer. I will try to list the few important things that students should know about programming languages and frameworks.

What is a programming language?

Shortly, it is a set of keywords and rules of their usage that allows a programmer to tell a computer what to do. From a technical point of view, there are many ways to classify languages - compiled and interpreted, functional and object-oriented, low-level and high-level, etc..

do we have only one language in our project?

Probably not. Majority of applications includes at least two elements:

- **The server part.** This is where all the "heavy" calculations take place, background API interactions, Database write/read operations, etc.
Languages Used : Java, .net, python etc..
- **The client part.** For example, the interface of your website, mobile applications, desktop apps, etc.
Languages Used : HTML, Java Script, Angular, React etc.

Obviously, there can be much more than two languages in the project, especially considering such things as SQL used for database operations.

What is a Framework?

When choosing a technology stack for our project, we will surely come across such as framework. A framework is a set of ready-made elements, rules, and components that simplify the process and increase the development speed. Below are some popular frameworks as an example:

- JAVA : Spring, SpringBoot, Struts, Hibernate, Quarkus etc..
- PHP Frameworks: Laravel, Symfony, Codeigniter, Slim, Lumen
- JavaScript Frameworks: ReactJs, VueJs, AngularJs, NodeJs
- Python Frameworks: Django, TurboGears, Dash

What kind of tasks does a framework solve?

Frameworks can be general-purpose or designed to solve a particular type of problems. In the case of web frameworks, they often contain out-of-the-box components for handling:

- Routing URLs
- Security
- Database Interaction,
- caching
- Exception handling, etc.

Do I need a framework?

- **It will save time.** Using premade components will allow you to avoid reinventing the logics again and writing from scratch those parts of the application which already exist in the framework itself.
- **It will save you from making mistakes.** Good frameworks are usually well written. Not always perfect, but on average much better than the code your team will deliver from scratch, especially when you're on a short timeline and tight budget.
- **Opens up access to the infrastructure.** There are many existing extensions for popular frameworks, as well as convenient performance testing tools, CI/CD, ready-to-use boilerplates for creating various types of applications.

Conclusion:

While a programming language is a foundation, a framework is an add-on, a set of components and additional functionality which simplifies the creation of applications. In My opinion - using a modern framework is in **95%** of cases a good idea, and it's **always** a great idea to create an applications with a framework rather than raw language.

Spring Introduction

The Spring Framework is a popular Java-based application framework used for building enterprise-level applications. It was developed by Rod Johnson in 2003 and has since become one of the most widely used frameworks in the Java ecosystem. The term "Spring" means different things in different contexts.



The framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications, with support for features such as dependency injection, aspect-oriented programming, data access, and transaction management. Spring handles the infrastructure so you can focus on your application. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

One of the key features of the Spring Framework is its ability to promote loose coupling between components, making it easier to develop modular, maintainable, and scalable applications. The framework also provides a wide range of extensions and modules that can be used to integrate with other technologies and frameworks, such as Hibernate, Struts, and JPA.

Overall, the Spring Framework is widely regarded as a powerful and flexible framework for building enterprise-level applications in Java.

The Spring Framework provides a variety of features, including:

- **Dependency Injection:** Spring provides a powerful dependency injection mechanism that helps developers write code that is more modular, flexible, and testable.
- **Inversion of Control:** Spring also provides inversion of control (IoC) capabilities that help decouple the application components and make it easier to manage and maintain them.
- **AOP:** Spring's aspect-oriented programming (AOP) framework helps developers modularize cross-cutting concerns, such as security and transaction management.
- **Spring MVC:** Spring MVC is a popular web framework that provides a model-view-controller (MVC) architecture for building web applications.
- **Integration:** Spring provides integration with a variety of other popular Java technologies, such as Hibernate, JPA, JMS.

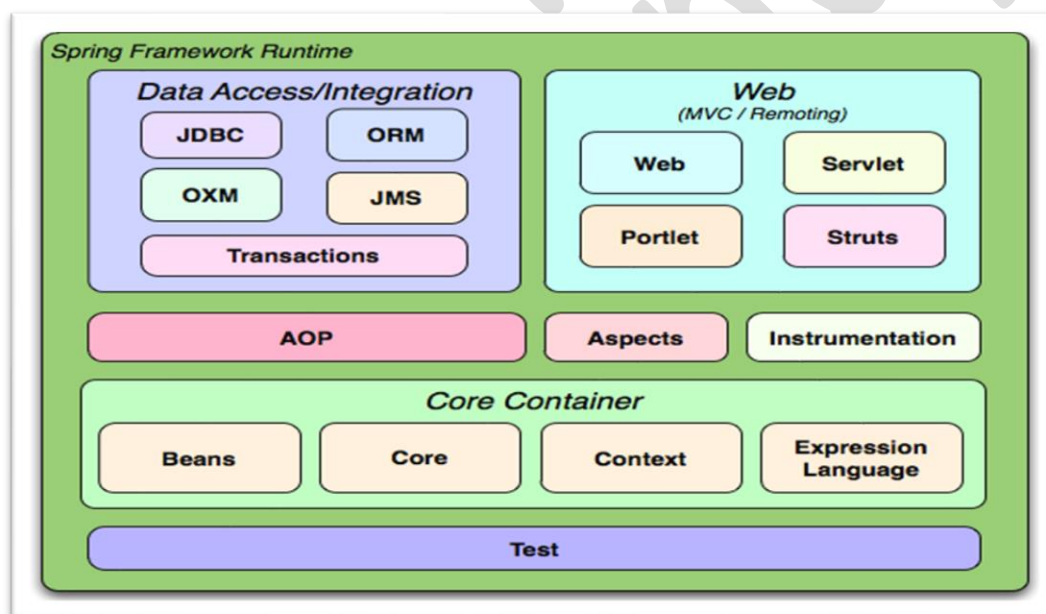
Overall, Spring Framework has become one of the most popular Java frameworks due to its ease of use, modularity, and extensive features. It is widely used in enterprise applications, web applications, and other types of Java-based projects.

Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others.

Spring Framework architecture is an arranged layered architecture that consists of different modules. All the modules have their own functionalities that are utilized to build an application.

The Spring Framework includes several modules that provide a range of services:

- **Spring Core Container:** this is the base module of Spring and provides spring containers (BeanFactory and ApplicationContext).
- **Aspect-oriented programming:** enables implementing cross-cutting concerns.
- **Data access:** working with relational database management systems on the Java platform using Java Database Connectivity (JDBC) and object-relational mapping tools and with NoSQL databases
- **Authentication and authorization:** configurable security processes that support a range of standards, protocols, tools and practices via the Spring Security sub-project.
- **Model–View–Controller:** an HTTP- and servlet-based framework providing hooks for web applications and RESTful (representational state transfer) Web services.
- **Testing:** support classes for writing unit tests and integration tests



Spring Release Version History:

Version	Date	Notes
0.9	2003	
1.0	March 24, 2004	First production release.
2.0	2006	
3.0	2009	
4.0	2013	
5.0	2017	
6.0	November 16, 2022	Current/Latest Version

Advantages of Spring Framework:

The Spring Framework is a popular open-source application framework for developing Java applications. It provides a number of advantages that make it a popular choice among developers. Here are some of the key advantages of the Spring Framework:

1. **Lightweight:** Spring is a lightweight framework, which means it does not require a heavy runtime environment to run. This makes it faster and more efficient than other frameworks.
2. **Inversion of Control (IOC):** The Spring Framework uses IOC to manage dependencies between different components in an application. This makes it easier to manage and maintain complex applications.
3. **Dependency Injection (DI):** The Spring Framework also supports DI, which allows you to inject dependencies into your code at runtime. This makes it easier to write testable and modular code.
4. **Modular:** Spring is a modular framework, which means you can use only the components that you need. This makes it easier to develop and maintain applications.
5. **Loose Coupling:** The Spring applications are loosely coupled because of dependency injection.
6. **Integration:** The Spring Framework provides seamless integration with other frameworks and technologies such as Hibernate, Struts, and JPA.
7. **Aspect-Oriented Programming (AOP):** The Spring Framework supports AOP, which allows you to separate cross-cutting concerns from your business logic. This makes it easier to develop and maintain complex applications.
8. **Security:** The Spring Framework provides robust security features such as authentication, authorization, and secure communication.
9. **Transaction Management:** The Spring Framework provides robust transaction management capabilities, which make it easier to manage transactions across different components in an application.
10. **Community Support:** The Spring Framework has a large and active community, which provides support and contributes to its development. This makes it easier to find help and resources when you need them.

Overall, the Spring Framework provides a number of advantages that make it a popular choice among developers. Its lightweight, modular, and flexible nature, along with its robust features for managing dependencies, transactions, security, and integration, make it a powerful tool for developing enterprise-level Java applications.

Why do we use Spring in Java?

- Works on POJOs (Plain Old Java Object) which makes your application lightweight.
- Provides predefined templates for JDBC, Hibernate, JPA etc., thus reducing your effort of writing too much code.
- Because of dependency injection feature, your code becomes loosely coupled.
- Using Spring Framework, the development of Java Enterprise Edition (JEE) applications became faster.
- It also provides strong abstraction to Java Enterprise Edition (JEE) specifications.
- It provides declarative support for transactions, validation, caching and formatting.

What is the difference between Java and Spring?

The below table represents the differences between Java and Spring:

Java	Spring
Java is one of the prominent programming languages in the market.	Spring is a Java-based open-source application framework.
Java provides a full-highlighted Enterprise Application Framework stack called Java EE for web application development	Spring Framework comes with various modules like Spring MVC, Spring Boot, Spring Security which provides various ready to use features for web application development.
Java EE is built upon a 3-D Architectural Framework which are Logical Tiers, Client Tiers and Presentation Tiers.	Spring is based on a layered architecture that consists of various modules that are built on top of its core container.

Since its origin till date, Spring has spread its popularity across various domains.

Spring Core Module

Core Container:

Spring Core Module has the following three concepts:

1. **Spring Core:** This module is the core of the Spring Framework. It provides an implementation for features like IoC (Inversion of Control) and Dependency Injection with a singleton design pattern.
2. **Spring Bean:** This module provides an implementation for the factory design pattern through BeanFactory.
3. **Spring Context:** This module is built on the solid base provided by the Core and the Beans modules and is a medium to access any object defined and configured.

Spring Bean:

Beans are java objects that are configured at run-time by Spring IoC Container. In Spring, the objects of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by Spring container.

Dependency Injection in Spring:

Dependency Injection is the concept of an object to supply dependencies of another object. Dependency Injection is one such technique which aims to help the developer code easily by providing dependencies of another object. 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 container** rather than by the objects themselves.

When we hear the term dependency, what comes on to our mind? Obviously, something relying on something else for support right? Well, that’s the same, in the case of programming also.

Dependency in programming is an approach where a class uses specific functionalities of another class. So, for example, If you consider two classes A and B, and say that class A using functionalities of class B, then its implied that class A has a dependency of class B i.e. A depends on B. Now, if we are coding in Java then you must know that, you have to create an instance/Object of class B before the functionalities are being used by class A.

Dependency Injection in Spring can be done through constructors, setters or fields. Here's how we would create an object dependency in traditional programming:

Employee.java

```
public class Employee {  
    private String ename;  
    private Address addr;  
  
    public Employee() {  
        this.addr = new Address();  
    }  
    // setter & getter methods  
}
```

Address.java

```
public class Address {  
    private String cityName;  
    // setter & getter methods  
}
```

In the example above, we need to instantiate an implementation of the Address within the *Employee* class itself.

By using DI, we can rewrite the example without specifying the implementation of the *Address* that we want:

```
public class Employee {  
    private String ename;  
    private Address addr;  
  
    public Employee(Address addr) {  
        this.addr = addr;  
    }  
}
```

```
}  
}
```

In the next sections, we'll look at how we can provide the implementation of *Address* through metadata. Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

Spring Container / IOC Container:

An IoC container is a common characteristic of frameworks that implement IoC principle in software engineering.

Inversion of Control:

Inversion of Control is a principle in software engineering, which transfers the control of objects of a program to a container or framework. We most often use it in the context of object-oriented programming.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets information's from the XML file or Using annotations and works accordingly.

The main tasks performed by IoC container are:

- to instantiate the application java classes
- to configure data with the objects
- to assemble the dependencies between the objects internally

As I have mentioned above Inversion of Control is a principle based on which, Dependency Injection is made. Also, as the name suggests, Inversion of Control is basically used to invert different kinds of additional responsibilities of a class rather than the main responsibility.

If I have to explain you in simpler terms, then consider an example, wherein you have the ability to cook. According to the IoC principle, you can invert the control, so instead of you cooking food, you can just directly order from outside, wherein you receive food at your doorstep. Thus the process of food delivered to you at your doorstep is called the Inversion of Control.

You do not have to cook yourself, instead, you can order the food and let a delivery executive, deliver the food for you. In this way, you do not have to take care of the additional responsibilities and just focus on the main work.

Spring IOC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects.

Spring provides two types of Container Implementations namely as follows:

1. **BeanFactory Container**
2. **ApplicationContext Container**

Spring IoC container is the program that injects dependencies into an object and make it ready for our use. Spring IoC container classes are part of **org.springframework.beans** and **org.springframework.context** packages from spring framework. Spring IoC container provides us different ways to decouple the object dependencies. **BeanFactory** is the root interface of Spring IoC container. **ApplicationContext** is the child interface of BeanFactory interface. These Interfaces are having many implementation classes in same packages to create IOC container in execution time.

Spring Framework provides a number of useful **ApplicationContext** implementation classes that we can use to get the spring context and then the Spring Bean. Some of the useful **ApplicationContext** implementations that we use are.

- **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
- **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
- **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

spring is actually a container and behaves as a factory of Beans.

Spring – BeanFactory:

This is the simplest container providing the basic support for DI and defined by the **org.springframework.beans.factory.BeanFactory** interface. BeanFactory interface is the simplest container providing an advanced configuration mechanism to instantiate, configure and manage the life cycle of beans. **BeanFactory** represents a basic IoC container which is a parent interface of **ApplicationContext**. **BeanFactory** uses Beans and their dependencies metadata i.e. what we configured in XML file to create and configure them at run-time. BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file(XML) or the beans can be directly returned when required using Java Configuration.

Spring ApplicationContext:

The **org.springframework.context.ApplicationContext** interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. Several implementations of the ApplicationContext interface are supplied with Spring. In standalone applications, it is common to create an instance of **ClassPathXmlApplicationContext** or **FileSystemXmlApplicationContext**. While XML has been the traditional format for defining configuration of spring bean classes. We can instruct the

container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

The following diagram shows a high-level view of how Spring Container works. Your application bean classes are combined with configuration metadata so that, after the **ApplicationContext** is created and initialized, you have a fully configured and executable system or application.

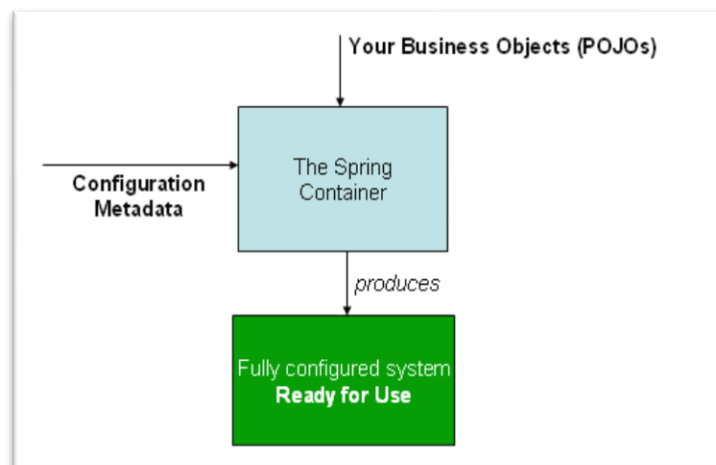


Figure :The Spring IoC container

Configuration Metadata:

As diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application. Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container. These days, many developers choose Java-based configuration for their Spring applications.

Instantiating a Container:

The location path or paths supplied to an **ApplicationContext** constructor are resource Strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java CLASSPATH, and so on. The Spring provides **ApplicationContext** interface: **ClassPathXmlApplicationContext** and **FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations. Here's one way to manually instantiate a container:

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Difference Between BeanFactory Vs ApplicationContext:

BeanFactory	ApplicationContext
It is a fundamental container that provides the basic functionality for managing beans.	It is an advanced container that extends the BeanFactory that provides all basic functionality and adds some advanced features.
It is suitable to build standalone applications.	It is suitable to build Web applications, integration with AOP modules, ORM and distributed applications.
It supports only Singleton and Prototype bean scopes.	It supports all types of bean scopes such as Singleton, Prototype, Request, Session etc.
It does not support Annotation based configuration.	It supports Annotation based configuration in Bean Autowiring.
This interface does not provide messaging (i18n or internationalization) functionality.	ApplicationContext interface extends MessageSource interface, thus it provides messaging (i18n or internationalization) functionality.
BeanFactory will create a bean object when the getBean() method is called thus making it Lazy initialization.	ApplicationContext loads all the beans and creates objects at the time of startup only thus making it Eager initialization.

NOTE: Usually, if we are working on Spring MVC application and our application is configured to use Spring Framework, Spring IoC container gets initialized when the application started or deployed and when a bean is requested, the dependencies are injected automatically. However, for a standalone application, you need to initialize the container somewhere in the application and then use it to get the spring beans.

Create First Spring Core module Application:

NOTE: We can Create Spring Core Module Project in 2 ways.

1. Manually Downloading Spring JAR files and Copying/Configuring Build Path
2. By Using Maven Project Setup, Configuring Spring JAR files in Maven. This is preferred in Real Time practice.

In Maven Project, JAR files are always configured with **pom.xml** file i.e. we should not download manually JAR files in any Project. In First Approach we are downloading JAR files manually from Internet into our computer and then setting class Path to those jar file, this is not recommended in Real time projects.

Creation of Project with Downloaded Spring Jar Files :

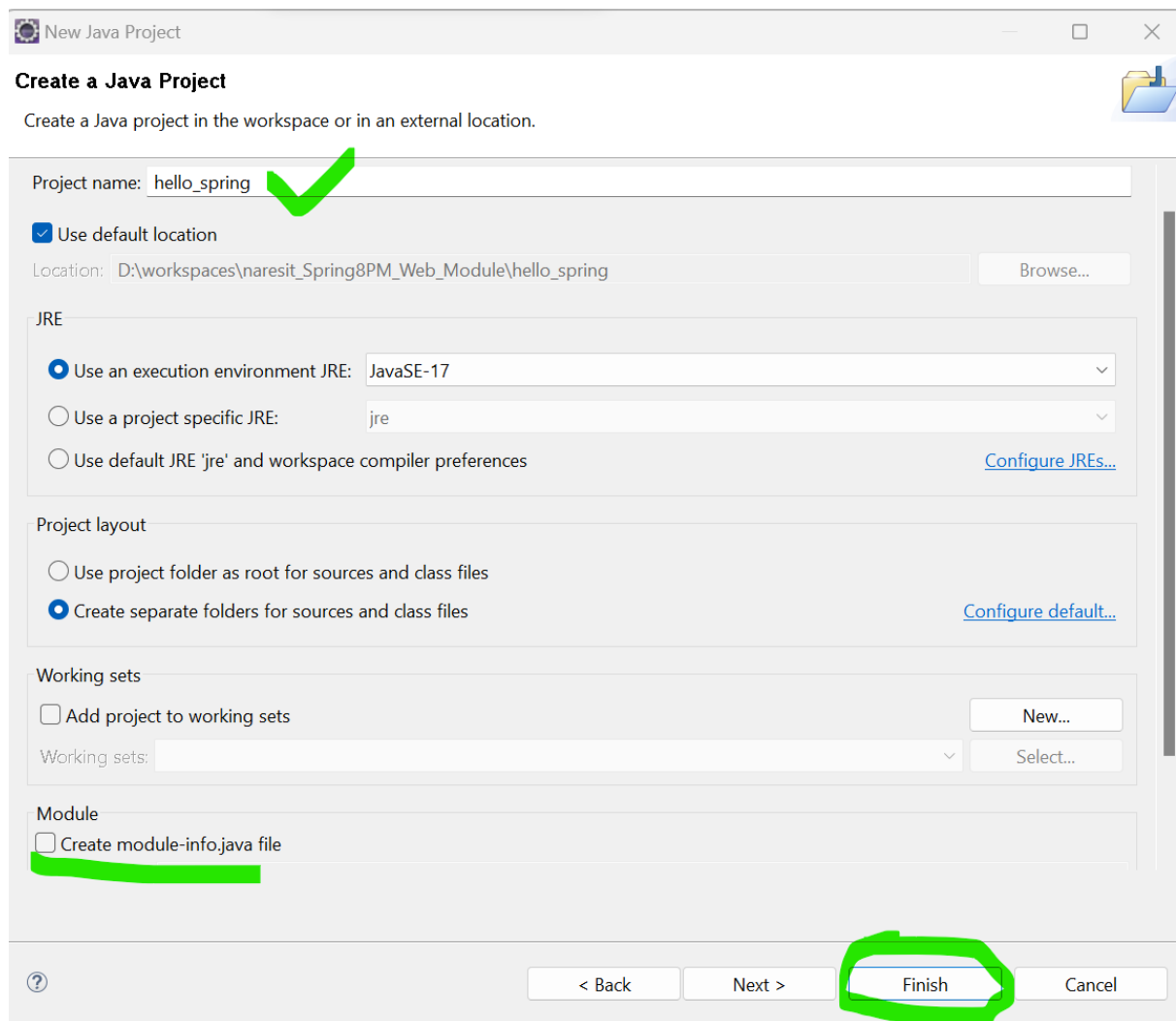
1. Open Eclipse

File -> new -> Project -> Java Project

Enter Project Name

Un-Select Create Module-Info

Click Finish.



2. Now Download Spring Framework Libraries/jar files from Online/Internet.

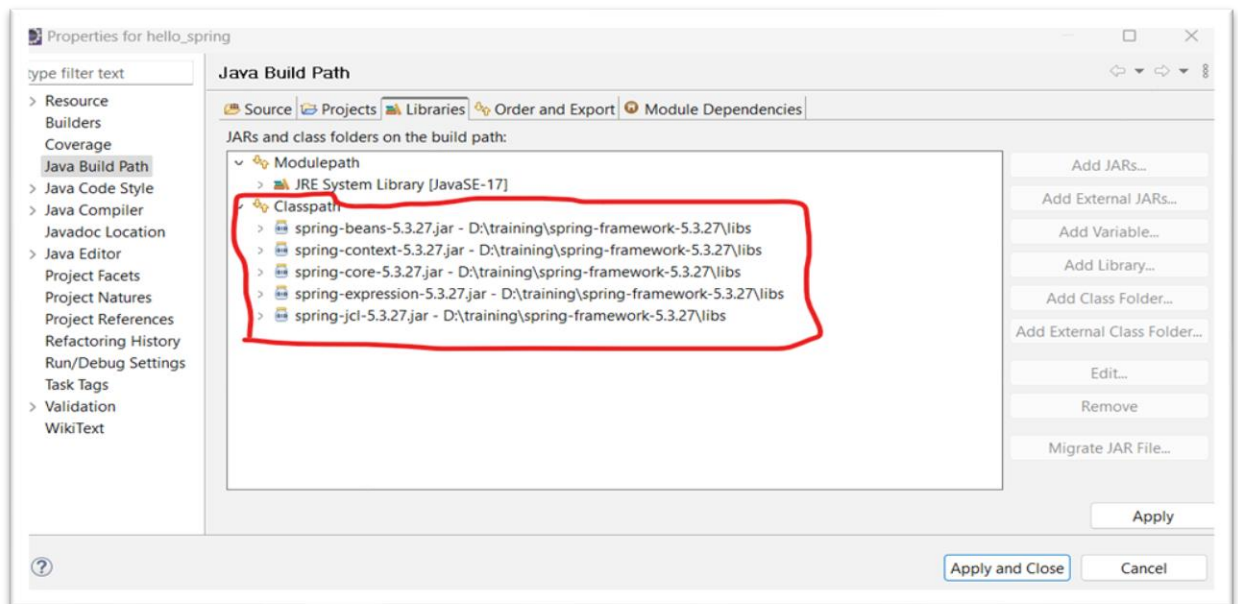
I have uploaded copy of all Spring JAR files uploaded in Google Drive. You can download from below link directly.

https://drive.google.com/file/d/1FnbtP3yqjTN5arIEGeoUHCrlJcdcBgM7/view?usp=drive_link

After Download completes, Please extract .zip file.

- Now Please set build path to java project with Spring core jar files from lib folder in downloaded in step 2, which are shown in image.

Right Click on Project -> Build Path -> Configure Build Path -> Libraries -> ClassPath

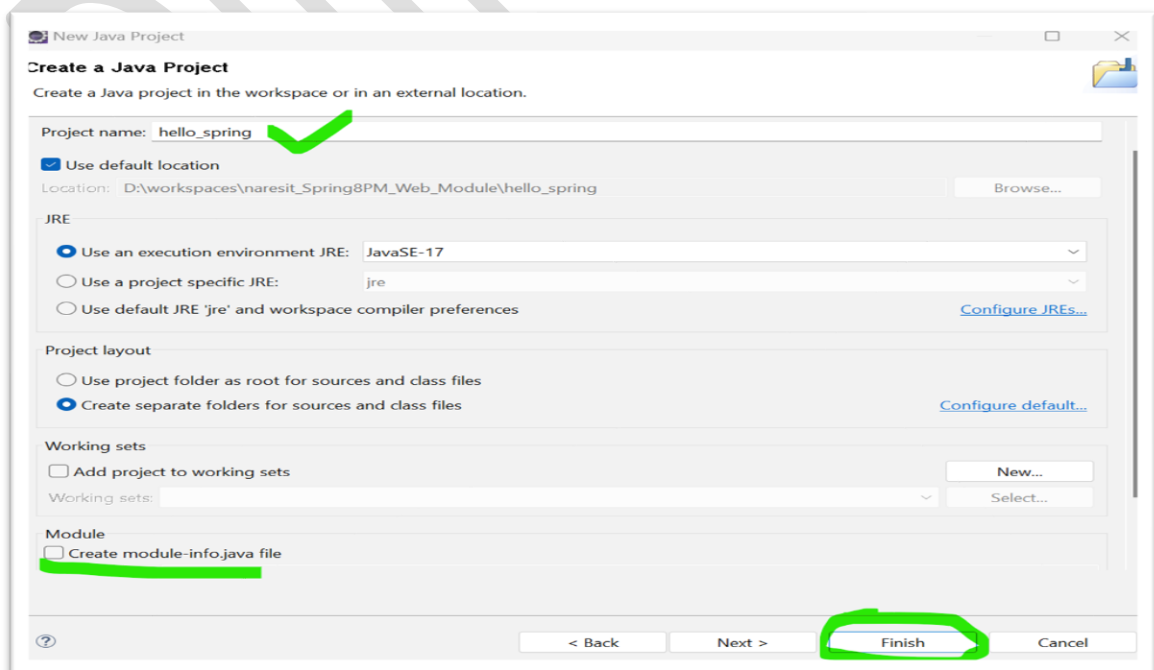


With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

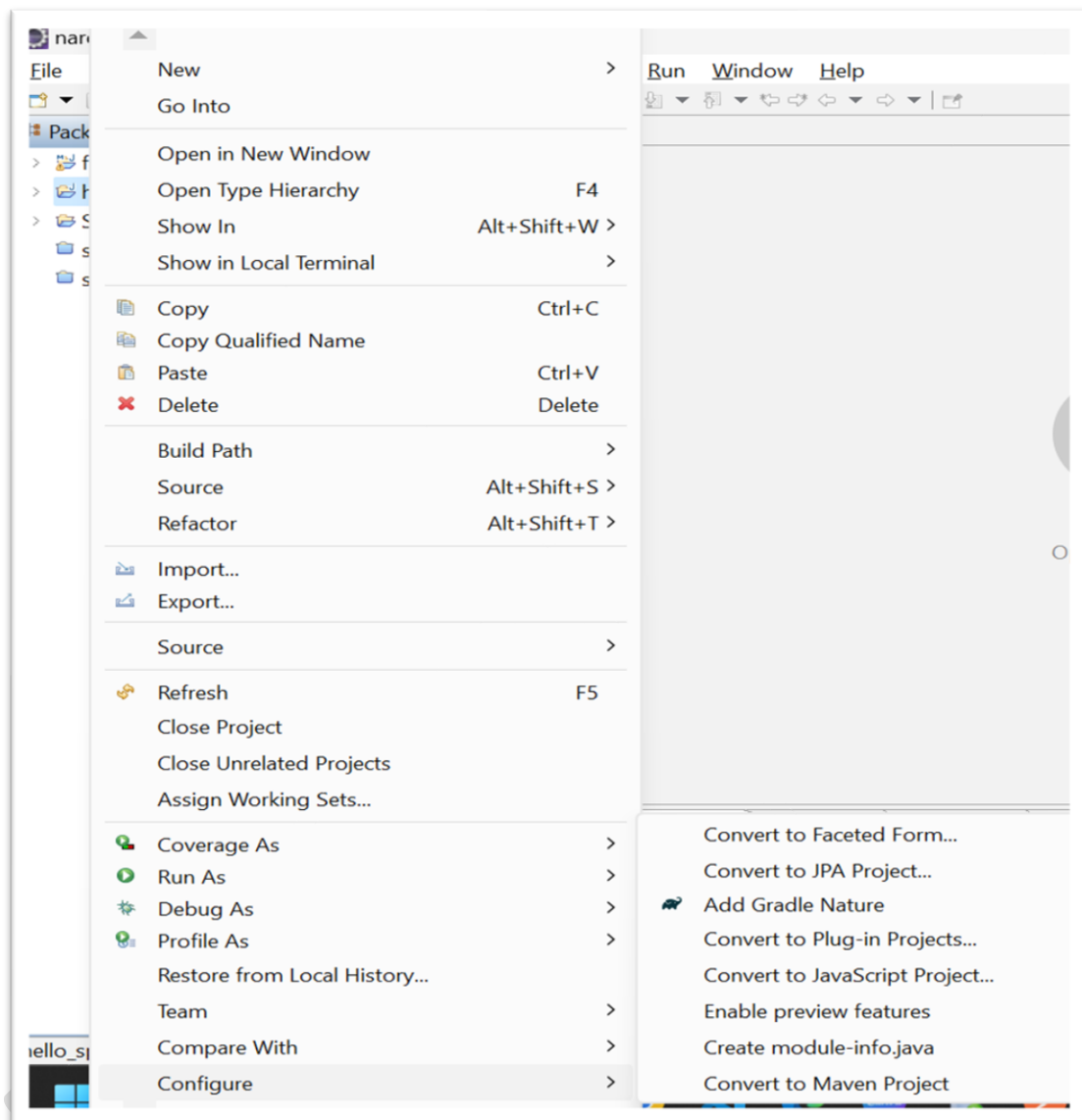
Creating Spring Core Project with Maven Configuration:

1. Create Java Project.

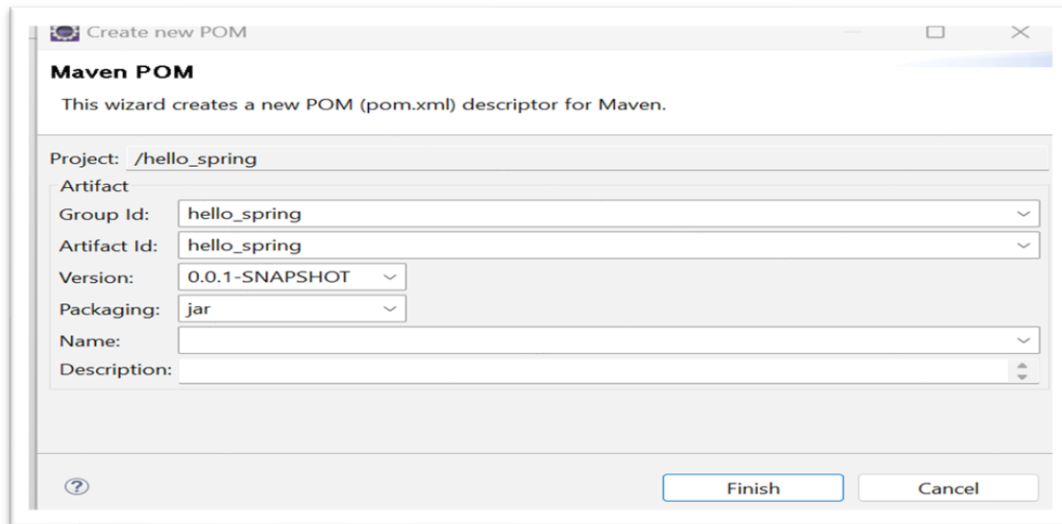
**Open Eclipse -> File -> new -> Project -> Java Project -> Enter Project Name
-> Un-Select Create Module-Info -> Click Finish.**



2. Now Right Click On Project and Select Configure -> Convert to Maven Project.



Immediately It will show below details and click on Finish.



3. Now With Above Step, Java Project Supporting Maven functionalities. Created a default pom.xml as well. Project Structure shown as below.



Now Open pom.xml file, add Spring Core JAR Dependencies to project and save it.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

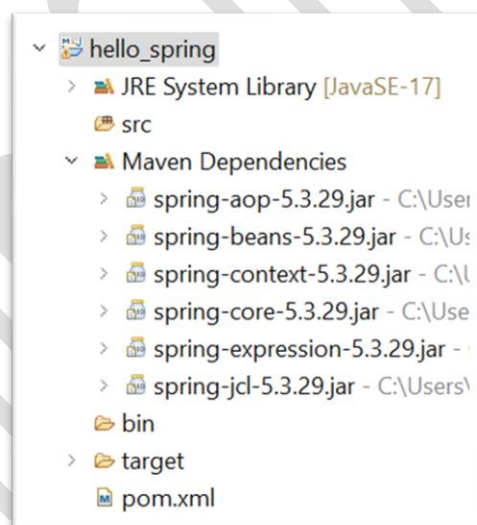
  <groupId>hello_spring</groupId>
  <artifactId>hello_spring</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.3.29</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.29</version>
    </dependency>
  </dependencies>
</project>
```

```

<build>
  <sourceDirectory>src</sourceDirectory>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <release>17</release>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

After adding Dependencies, Maven downloads all Spring Core Jar files with internal dependencies of jars at the same time configures those as part of Project automatically. As a Developer we no need to configure of jars in this approach. Now we can See Downloaded JAR files under Maven Dependencies Section as shown in below.



With This Our Java Project is Supporting Spring Core Module Functionalities. We can Continue with Spring Core Module Functionalities.

NOTE: Below Steps are now common across our Spring Core Project created by either Manual Jar files or Maven Configuration.

4. Now Create a java POJO Class in src inside package.

```

package com.naresh.hello;

public class Student {

```

```

private String studnetName;

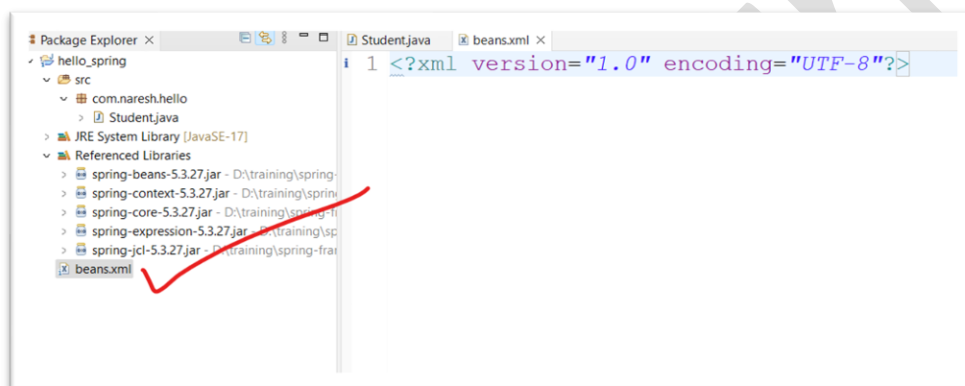
public String getStudnetName() {
    return studnetName;
}

public void setStudnetName(String studnetName) {
    this.studnetName = studnetName;
}
}

```

5. Now create a xml file with any name in side our project root folder:

Ex: **beans.xml**



6. Now Inside **beans.xml**, and paste below XML Shema content to configure all our bean classes.

```

<?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">

    <!-- Configure Our Bean classes Here -->

</beans>

```

We can get above content from below link as well.

<https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html>

7. Now configure our POJO class **Student** in side **beans.xml** file.

```

<?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 id="stu" class="com.naresh.hello.Student">    </bean>
</beans>

```

From above Configuration, Points to be Noted:

- Every class will be configured with **<bean>** tag, we can call it as Bean class.
 - The **id** attribute is a string that identifies the individual bean name in Spring IOC Container i.e. similar to Object Name or Reference.
 - The **class** attribute is fully qualified class name our class i.e. class name with package name.
8. Now create a main method class for testing.

Here we are getting the object of Student class from the Spring IOC container using the **getBean()** method of **BeanFactory**. Let's see the code

```

package com.naresh.hello;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;

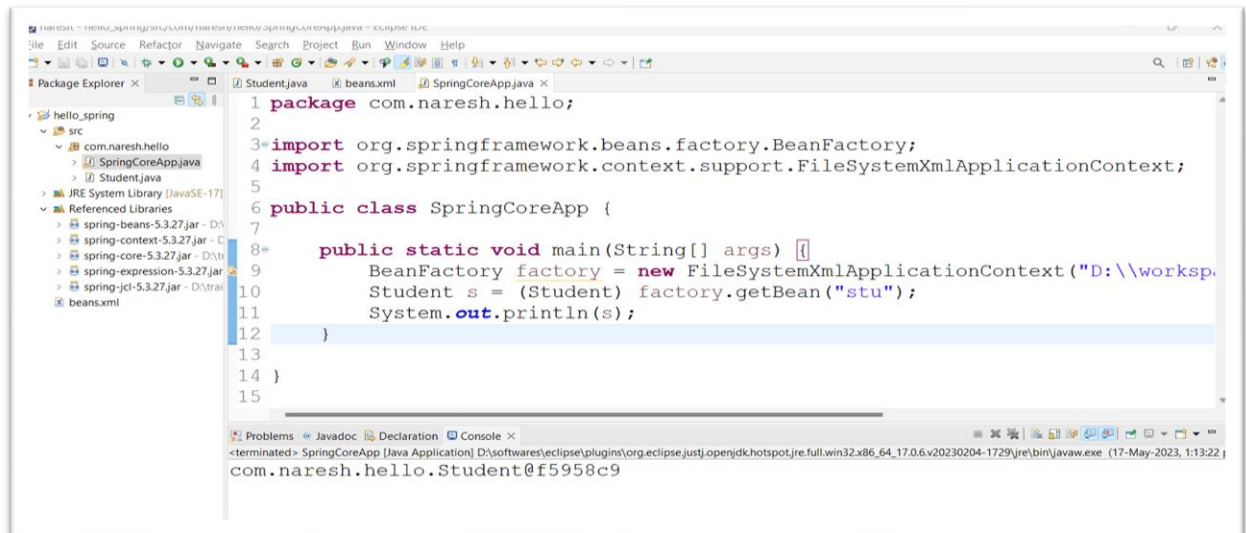
public class SpringCoreApp {
    public static void main(String[] args) {

        // BeanFactory Object is called as IOC Container.
        BeanFactory factory = new
        FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\hello_spring\\beans.xml");

        Student s = (Student) factory.getBean("stu");
        System.out.println(s);
    }
}

```

9. Now Execute Your Program : Run as Java Application.



In above example Student Object Created by Spring IOC container and we got it by using **getBean()** method. If you observe, we are not written code for Student Object Creation i.e. using new operator.

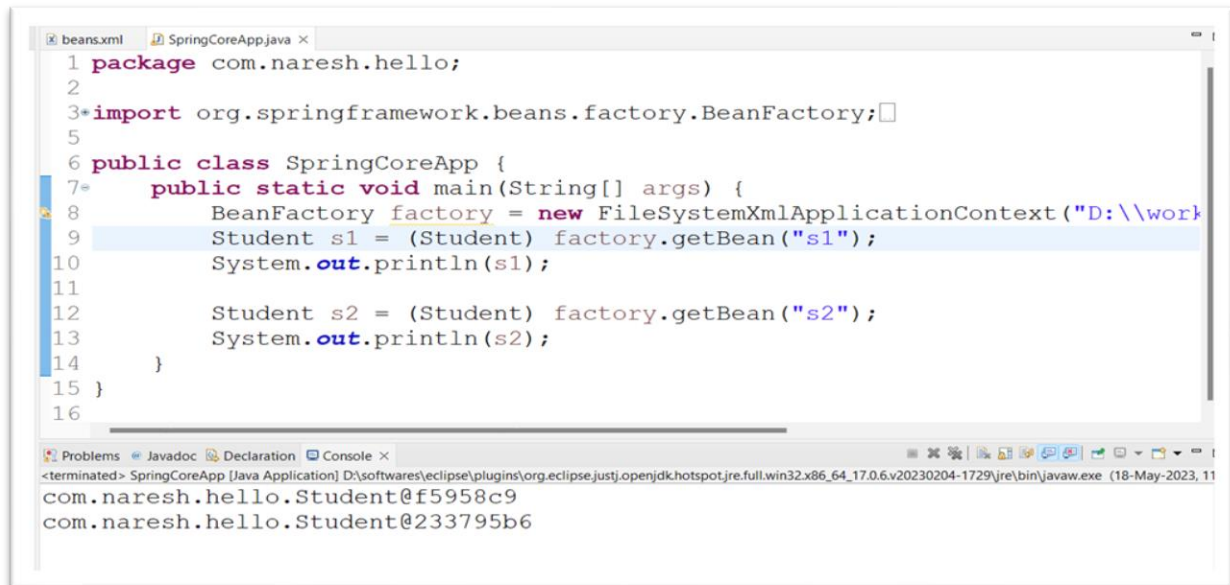
- We can create multiple Bean Objects for same Bean class with multiple bean configurations in xml file.

```
<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 id="s1" class="com.naresh.hello.Student"> </bean>
    <bean id="s2" class="com.naresh.hello.Student"> </bean>

</beans>
```

- Now In Main Application class, Get Second Object.



```
1 package com.naresh.hello;
2
3 import org.springframework.beans.factory.BeanFactory;
4
5
6 public class SpringCoreApp {
7     public static void main(String[] args) {
8         BeanFactory factory = new FileSystemXmlApplicationContext("D:\\work\\
9         Student s1 = (Student) factory.getBean("s1");
10        System.out.println(s1);
11
12        Student s2 = (Student) factory.getBean("s2");
13        System.out.println(s2);
14    }
15 }
16
```

Problems Javadoc Declaration Console X

<terminated> SpringCoreApp [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (18-May-2023, 11

com.naresh.hello.Student@f5958c9

com.naresh.hello.Student@233795b6

So we can provide multiple configurations and create multiple Bean Objects for a class.

Bean Overview:

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML `<bean/>` definitions). Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier. However, if it requires more than one, the extra ones can be considered aliases. In XML-based configuration metadata, you use the `id` attribute, the `name` attribute, or both to specify bean identifiers. The `id` attribute lets you specify exactly one id.

Bean Naming Conventions:

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include **accountManager**, **accountService**, **userDao**, **loginController**.

Instantiating Beans:

A bean definition is essentially a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the **class** attribute of the `<bean/>` element. This **class** attribute (which, internally, is a Class property on a BeanDefinition instance) is usually mandatory.

Types of Dependency Injection:

Dependency Injection (DI) is a design pattern that allows us to decouple the dependencies of a class from the class itself. This makes the class more loosely coupled and easier to test. In Spring, DI can be achieved through constructors, setters, or fields.

1. Setter Injection
2. Constructor Injection
3. Filed Injection

There are many benefits to using dependency injection in Spring. Some of the benefits include:

- **Loose coupling:** Dependency injection makes the classes in our application loosely coupled. This means that the classes are not tightly coupled to the specific implementations of their dependencies. This makes the classes more reusable and easier to test.
- **Increased testability:** Dependency injection makes the classes in our application more testable. This is because we can inject mock implementations of dependencies into the classes during testing. This allows us to test the classes in isolation, without having to worry about the dependencies.
- **Increased flexibility:** Dependency injection makes our applications more flexible. This is because we can change the implementations of dependencies without having to change the classes that depend on them. This makes it easier to change the underlying technologies in our applications.

Dependency injection is a powerful design pattern that can be used to improve the design and testability of our Spring applications. By using dependency injection, we can make our applications more loosely coupled, increase their testability, and improve their flexibility.

Setter Injection:

Setter injection is another way to inject dependencies in Spring. In this approach, we specify the dependencies in the class setter methods. The Spring container will then create an instance of the class and then call the setter methods to inject the dependencies.

The **<property>** sub element of **<bean>** is used for setter injection. Here we are going to inject

- primitive and String-based values
- Dependent object (contained object)
- Collection values etc.

Now Let's take example for setter injection.

1. Create a class.

```
package com.naresh.first.core;
```

```

public class Student {

    private String studentName;
    private String studentId;
    private String clgName;
    public String getClgName() {
        return clgName;
    }
    public void setClgName(String clgName) {
        this.clgName = clgName;
    }
    public String getStudentName() {
        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public String getStudentId() {
        return studentId;
    }
    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }
    public void printStudentDeatils() {
        System.out.println("This is Student class");
    }
    public double getAvgOfMArks() {
        return 456 / 6;
    }
}

```

2. Configure bean in beans xml file :

```

<?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 id="s1" class="com.naresh.first.core.Student">
        <property name="clgName" value="ABC College " />
        <property name="studentName" value="Dilip Singh " />
        <property name="studentId" value="100" />
    </bean>

```


</beans>

From above configuration, <property> tag referring to setter injection i.e. injecting value to a variable or property of Bean Student class.

<property> tag contains some attributes.

name: Name of the Property i.e. variable name of Bean class

value: Real/Actual value of Variable for injecting/storing

3. Now get the bean object from Spring Container and print properties values.

```
package com.naresh.first.core;
```

```
import org.springframework.beans.factory.BeanFactory;
```

```
import org.springframework.context.support.FileSystemXmlApplicationContext;
```

```
public class SpringApp {
```

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

```
        BeanFactory factory = new  
        FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_first\\beans.xml");
```

```
        // Requesting Spring Container for Student Object
```

```
        Student s1 = (Student) factory.getBean("s1");
```

```
        System.out.println(s1.getId());
```

```
        System.out.println(s1.getName());
```

```
        System.out.println(s1.getClgName());
```

```
        s1.printStudentDeatils();
```

```
        System.out.println(s1.getAvgOfMArks());
```

```
    }
```

```
}
```

Output:

```
100  
Dilip Singh  
ABC College  
This is Student class  
76.0
```

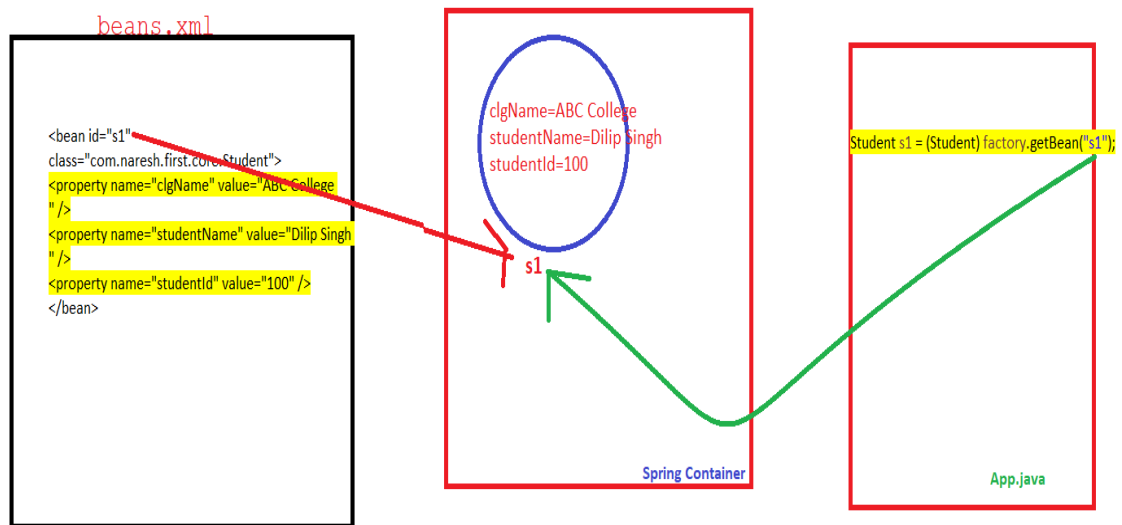
Internal Workflow/Execution of Above Program.

1. From the Below Line execution, Spring will create Spring IOC container and Loads our beans xml file in JVM memory and Creates Bean Objects inside Spring Container.

```
BeanFactory factory = new  
FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_first\\beans.xml");
```

2. Now from below line, we are getting bean object of Student class configured with bean id : s1

```
Student s1 = (Student) factory.getBean("s1");
```



3. Now we can use **s1** object and call our method as usual.

Injecting primitive and String Data properties:

Now we are injecting/configuring primitive and String data type properties into Spring Bean Object.

- Define a class, with different primitive datatypes and String properties.

```
package com.naresh.hello;
```

```
public class Student {
```

```
    private String stuName;  
    private int studId;  
    private double avgOfMarks;  
    private short passedOutYear;  
    private boolean isSelected;
```

```
    public String getStuName() {  
        return stuName;  
    }
```

```

    }
    public void setStuName(String stuName) {
        this.stuName = stuName;
    }
    public int getStudId() {
        return studId;
    }
    public void setStudId(int studId) {
        this.studId = studId;
    }
    public double getAvgOfMarks() {
        return avgOfMarks;
    }
    public void setAvgOfMarks(double avgOfMarks) {
        this.avgOfMarks = avgOfMarks;
    }
    public short getPassedOutYear() {
        return passedOutYear;
    }
    public void setPassedOutYear(short passedOutYear) {
        this.passedOutYear = passedOutYear;
    }
    public boolean isSelected() {
        return isSelected;
    }
    public void setSelected(boolean isSelected) {
        this.isSelected = isSelected;
    }
}

```

- Now configure above properties in spring beans xml file.

```

<bean id="studentOne" class="com.naresh.hello.Student">
    <property name="stuName" value="Dilip"></property>
    <property name="studId" value="101"></property>
    <property name="avgOfMarks" value="99.88"></property>
    <property name="passedOutYear" value="2022"></property>
    <property name="isSelected" value="true"></property>
</bean>

```

- For primitive and String data type properties of bean class, we can use both **name** and **value** attributes.
- Now let's test values injected or not from above bean configuration.

```
import org.springframework.context.ApplicationContext;
```

```

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context = new
        FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_notes\\beans.xml");
        Student s1 = (Student) context.getBean("studentOne"); // get it from
        container

        System.out.println(s1.getStudId());
        System.out.println(s1.getStuName());
        System.out.println(s1.getPassedOutYear());
        System.out.println(s1.getAvgOfMarks());
        System.out.println(s1.isSelected());
    }
}

```

Output:

```

101
Dilip
2022
99.88
True

```

Injecting Collection Data Types properties:

Now we are injecting/configuring Collection Data Types like List, Set and Map properties into Spring Bean Object.

- For **List** data type property, Spring Provided <list> tag, sub tag of <property>.

```

<list>
    <value> ... </value>
    <value>... </value>
    <value> .. </value>
    .....
</list>

```

- For **Set** data type property, Spring Provided <list> tag, sub tag of <property>.

```

<set>
    <value> ... </value>
    <value>... </value>
    <value> .. </value>
    .....
</set>

```

- For **Map** data type property, Spring Provided <list> tag, sub tag of <property>.

```

<map>
    <entry key="..." value="..." />
    <entry key="..." value="..." />

```

```

        <entry key="..." value="..." />
        .....
    </map>

```

🚦 Created Bean class with List, Set and Map properties.

```
package com.naresh.hello;
```

```
import java.util.List;
```

```
import java.util.Map;
```

```
import java.util.Set;
```

```
public class Student {
```

```
    private String stuName;
```

```
    private int studId;
```

```
    private double avgOfMarks;
```

```
    private short passedOutYear;
```

```
    private boolean isSelected;
```

```
    private List<String> emails;
```

```
    private Set<String> mobileNumbers;
```

```
    private Map<String, String> subMarks;
```

```
    public List<String> getEmails() {
```

```
        return emails;
```

```
    }
```

```
    public void setEmails(List<String> emails) {
```

```
        this.emails = emails;
```

```
    }
```

```
    public Set<String> getMobileNumbers() {
```

```
        return mobileNumbers;
```

```
    }
```

```
    public void setMobileNumbers(Set<String> mobileNumbers) {
```

```
        this.mobileNumbers = mobileNumbers;
```

```
    }
```

```
    public Map<String, String> getSubMarks() {
```

```
        return subMarks;
```

```
    }
```

```
    public void setSubMarks(Map<String, String> subMarks) {
```

```

        this.subMarks = subMarks;
    }

    public String getStuName() {
        return stuName;
    }

    public void setStuName(String stuName) {
        this.stuName = stuName;
    }

    public int getStudId() {
        return studId;
    }

    public void setStudId(int studId) {
        this.studId = studId;
    }

    public double getAvgOfMarks() {
        return avgOfMarks;
    }

    public void setAvgOfMarks(double avgOfMarks) {
        this.avgOfMarks = avgOfMarks;
    }

    public short getPassedOutYear() {
        return passedOutYear;
    }

    public void setPassedOutYear(short passedOutYear) {
        this.passedOutYear = passedOutYear;
    }

    public boolean isSelected() {
        return isSelected;
    }

    public void setIsSelected(boolean isSelected) {
        this.isSelected = isSelected;
    }
}

```

🔧 Now configure in beans xml file.

```
<?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 id="studentOne" class="com.naresh.hello.Student">
        <property name="stuName" value="Dilip"></property>
        <property name="studId" value="101"></property>
        <property name="avgOfMarks" value="99.88"></property>
        <property name="passedOutYear" value="2022"></property>
        <property name="isSelected" value="true"></property>
        <property name="emails">
            <list>
                <value>dilip@gmail.com</value>
                <value>laxmi@gmail.com</value>
                <value>dilip@gmail.com</value>
            </list>
        </property>
        <property name="mobileNumbers">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+1234567890</value>
            </set>
        </property>
        <property name="subMarks">
            <map>
                <entry key="maths" value="88" />
                <entry key="science" value="66" />
                <entry key="english" value="44" />
            </map>
        </property>
    </bean>
</beans>

```

- Now let's test values injected or not from above bean configuration.

package com.naresh.hello;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

```

public class SpringCoreApp {
    public static void main(String[] args) {

```

```

        ApplicationContext context = new
FileSystemXmlApplicationContext("D:\\workspaces\\naresit\\spring_notes\\beans.xml");
        Student s1 = (Student) context.getBean("studentOne"); // get it from
container

        System.out.println(s1.getStudId());
        System.out.println(s1.getStuName());
        System.out.println(s1.getPassedOutYear());
        System.out.println(s1.getAvgOfMarks());
        System.out.println(s1.isSelected());
        System.out.println(s1.getEmails()); // List Values
        System.out.println(s1.getMobileNumbers()); // Set Values
        System.out.println(s1.getSubMarks()); //Map values
    }
}

```

Output:

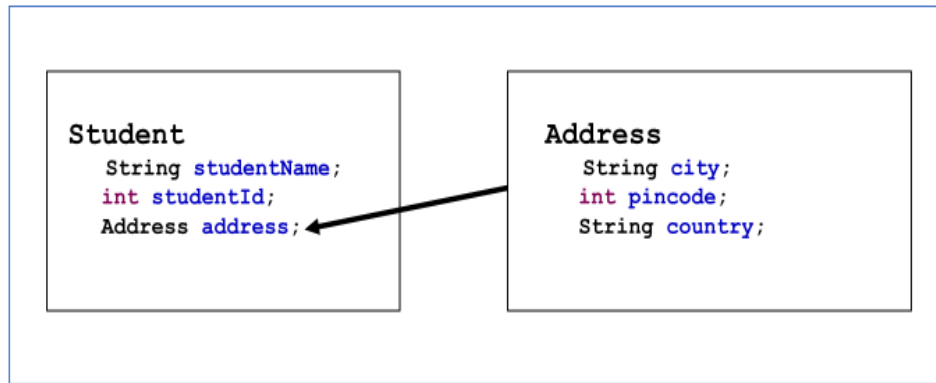
```

101
Dilip
2022
99.88
true
[dilip@gmail.com, laxmi@gmail.com, dilip@gmail.com]
[8826111377, +1234567890]
{maths=88, science=66, english=44}

```

Injecting Dependency's of Other Bean Objects:

Now we are injecting/configuring other Bean Objects into another Spring Bean Object.



- Now Create a class : **Address.java**

```
package com.naresh.training.spring.core;

public class Address {

    private String city;
    private int pincode;
    private String country;

    public Address() {
        System.out.println("Address instance/constructed ");
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
}
```

- Now Create another class with Address Type Property: **Student.java**

```

package com.naresh.training.spring.core;

public class Student {

    private String studentName;
    private int studentId;
    private Address address;

    public Student() {
        System.out.println("Student Constructor executed.");
    }
    public String getStudentName() {
        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}

```

- Now Configure Address and Student Bean classes. Here, Address Bean Object is dependency of Student object i.e. Address should be referred inside Student. To achieve this collaboration, Spring provided **ref** element/attribute.

ref attribute / <ref> element:

The **ref** element is the element inside a <property/> or <constructor-arg/> element. Here, you set the value of the specified property of a bean to be a referenced to another bean (a collaborator) managed by the container. Sometimes we can use **ref** attribute as part of <property/> or <constructor-arg/>. We will provide bean Id for **ref** element which should be injected into target Object. Please refer below, how to inject Bean Objects via **ref** element or attribute.

- Configure Bean classes now inside **beans.xml** 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 id="universityAddress"
        class="com.naresh.training.spring.core.Address">
        <property name="city" value="Bangalore"></property>
        <property name="country" value="India"></property>
        <property name="pincode" value="400066"></property>
    </bean>
    <!-- Student Bean Objects -->
    <bean id="student1" class="com.naresh.training.spring.core.Student">
        <property name="studentName" value="Dilip Singh"></property>
        <property name="studentId" value="100"></property>
        <property name="address" ref="universityAddress"></property>
    </bean>
    <bean id="student2" class="com.naresh.training.spring.core.Student">
        <property name="studentName" value="Naresh"></property>
        <property name="studentId" value="101"></property>
        <property name="address">
            <ref bean="universityAddress"/>
        </property>
    </bean>
</beans>

```

- Now let's test values and references injected or not from above bean configuration.

```

package com.naresh.training.spring.core;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringSetterInjectionDemo {

    public static void main(String[] args) {

        BeanFactory factory = new FileSystemXmlApplicationContext(
            "D:\\Spring\\spring-
            injection\\beans.xml");

        System.out.println("***** Student1 Data *****");
        Student stu1 = (Student) factory.getBean("student1");
        System.out.println(stu1.getStudentId());
        System.out.println(stu1.getStudentName());
        Address stu1Address = stu1.getAddress();
    }
}

```

```

        System.out.println(stu1Address.getCity());
        System.out.println(stu1Address.getCountry());
        System.out.println(stu1Address.getPincode());

        System.out.println("***** Student2 Data *****");
        Student stu2 = (Student) factory.getBean("student2");
        System.out.println(stu2.getId());
        System.out.println(stu2.getName());
        System.out.println(stu2.getAddress().getCity());
        System.out.println(stu2.getAddress().getCountry());
        System.out.println(stu2.getAddress().getPincode());
    }
}

```

Output:

```

Address instance/constructed
Student Constructor executed.
Student Constructor executed.
***** Student1 Data *****
100
Dilip Singh
Bangalore
India
400066
***** Student2 Data *****
101
Naresh
Bangalore
India
400066

```

From above output, same **universityAddress** bean Object is injected by Spring Container internally inside both **student1** and **student2** Bean Objects.

Constructor Injection:

Constructor injection is a form of dependency injection where dependencies are provided to a class through its constructor. It is a way to ensure that all required dependencies are supplied when creating an object. In constructor injection, the class that requires dependencies has one or more parameters in its constructor that represent the dependencies. When an instance of the class is created, the dependencies are passed as arguments to the constructor. Constructor injection is often considered a best practice in Spring because it helps ensure that the dependencies required for an object to function are provided at the time of its creation. This can lead to more maintainable and testable code.

In XML configuration, Spring provided a child tag `<constructor-arg>` of `<bean>` to achieve or configure Constructor Injection

Example: Defining Bean Class With Primitive and String Data type.

```
package com.naresh.spring.di.ci;

public class Product {

    private int productId;
    private String productName;
    private double price;

    // All Params Constructor
    public Product(int productId, String productName, double price) {
        super();
        System.out.println("Product All Param's Constructor Executed");
        this.productId = productId;
        this.productName = productName;
        this.price = price;
    }
    public int getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
    public double getPrice() {
        return price;
    }
    public void setProductId(int productId) {
        System.out.println("setProductId is called");
        this.productId = productId;
    }
    public void setProductName(String productName) {
        System.out.println("setProductName is called");
        this.productName = productName;
    }
    public void setPrice(double price) {
        System.out.println("setPrice is called");
        this.price = price;
    }
}
```

- Now Configure Bean Data With Constructor Injection.

```
<bean id="iphone" class="com.naresh.spring.di.ci.Product">
```

```
<constructor-arg value="1000"></constructor-arg>
<constructor-arg value="Apple Iphone 15 "></constructor-arg>
<constructor-arg value="150000.00"></constructor-arg>
</bean>
```

- From the above Bean Configuration, Spring Container Internally passes values to constructor of our class while creating Bean Object.

Testing:

```
package com.naresh.spring.di.ci;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

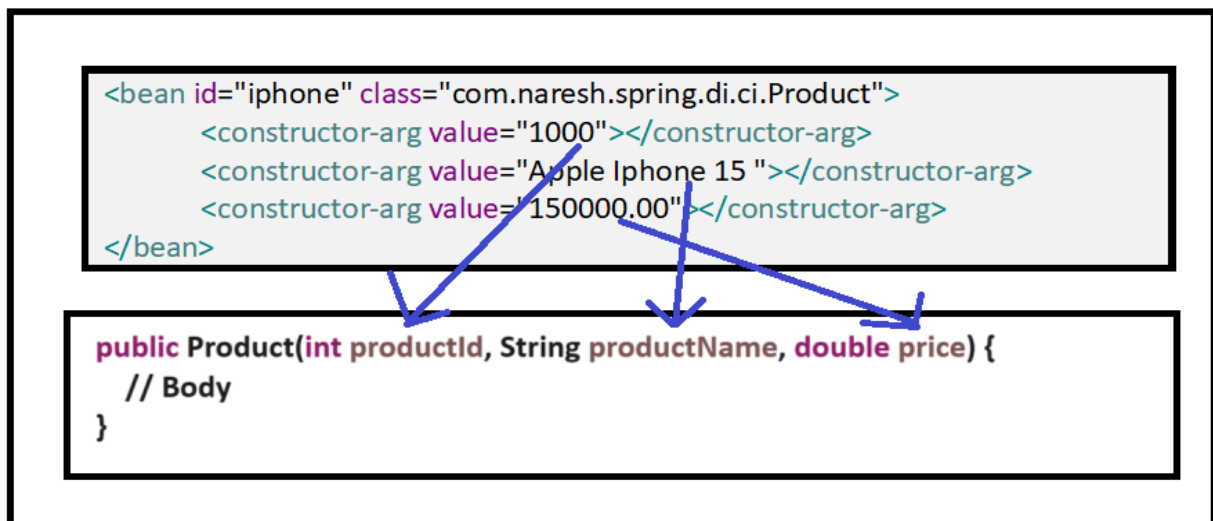
public class SpringConstructorInjectionDemo {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\spring-
beans.xml");
        Product product= (Product) context.getBean("iphone");
        System.out.println(product.getProductid());
        System.out.println(product.getProductName());
        System.out.println(product.getPrice());
    }
}
```

Output:

```
Product All Param's Constructor Executed
1000
Apple Iphone 15
150000.0
```

- ✚ Finally, values are injected into properties of bean Object by executing constructor.

NOTE: When we are defining `<constructor-arg>` and values in Beans XML Configuration, we should follow same order w.r.to Constructor Parameters.



- If we are not following same order in vice versa, then we will get below Exception while Creating Bean Object.

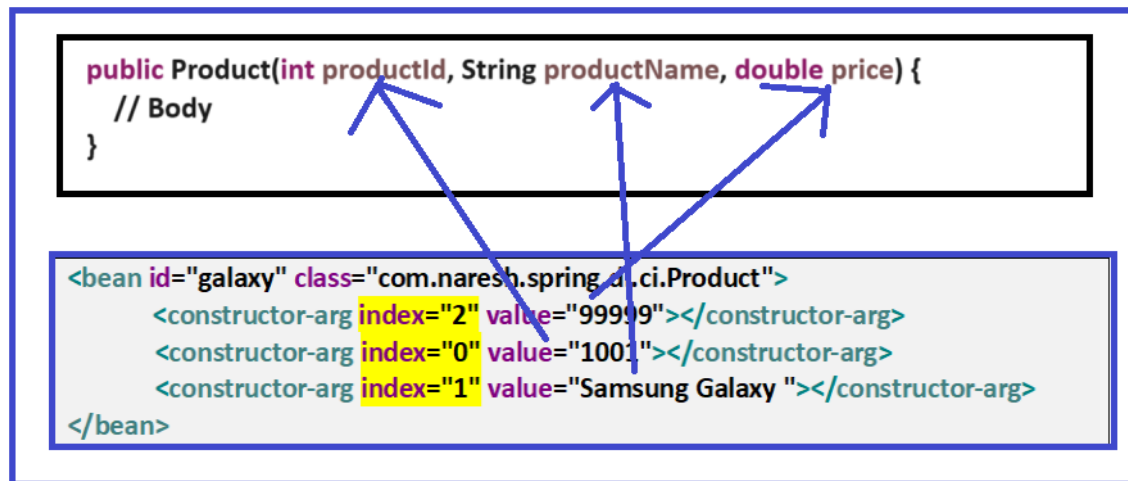
Exception in thread "main"

[org.springframework.beans.factory.UnsatisfiedDependencyException:](#)

Question: In any case, if we don't want to follow same order in beans configuration, do we have any alternative solution?

Answer: Yes, Spring provided an attribute **index** as part of **<constructor-arg>** tag i.e. we should provide index value for every property w.r.to Constructor Parameters Order. Here, Index starts from **0** always.

```
<bean id="galaxy" class="com.naresh.spring.di.ci.Product">
  <constructor-arg index="2" value="99999"></constructor-arg>
  <constructor-arg index="0" value="1001"></constructor-arg>
  <constructor-arg index="1" value="Samsung Galaxy "></constructor-arg>
</bean>
```



Question: Do we need to configure all values for all constructor parameters in Spring Bean Configuration?

Answer: Yes, We should configure every constructor parameter value inside bean configuration in Spring i.e. From above example, Constructor Defined with 3 parameters in Product class, so we should configure 3 values of `<constructor-arg>`.

If we are not configured same number of values respectively Spring will create an Exception while creating Bean Object for that Bean Configuration.

Exception in thread "main"

`org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'laptop' defined in file`

- we can create many constructors in a Spring Bean class, and we should configure values respectively for every bean Object with Constructor Injection.

Constructor Injection: Example with Collection Data Type and Another Object Reference.

- Define AccountDetails Bean class.

```
package com.naresh.hello;  
  
import java.util.Set;  
  
public class AccountDetails {  
  
    private String name;  
    private double balance;  
    private Set<String> mobiles;  
    private Address customerAddress;
```



```

    public AccountDetails(String name, double balance, Set<String> mobiles,
                           Address customerAddress) {

        super();
        this.name = name;
        this.balance = balance;
        this.mobiles = mobiles;
        this.customerAddress = customerAddress;
    }

    public AccountDetails() {

    }

    public Address getCustomerAddress() {
        return customerAddress;
    }

    public void setCustomerAddress(Address customerAddress) {
        this.customerAddress = customerAddress;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public Set<String> getMobiles() {
        return mobiles;
    }

    public void setMobiles(Set<String> mobiles) {
        this.mobiles = mobiles;
    }

}

```

- Define Dependency Class Address.

```

package com.naresh.hello;

public class Address {

    private int flatNo;
    private String houseName;
    private long mobile;
}

```

```

    public int getFlatNo() {
        return flatNo;
    }
    public void setFlatNo(int flatNo) {
        this.flatNo = flatNo;
    }
    public String getHouseName() {
        return houseName;
    }
    public void setHouseName(String houseName) {
        this.houseName = houseName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
}

```

- Define Bean Configuration

```

<?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 id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"></property>
        <property name="houseName" value="Lotus Homes"></property>
        <property name="mobile" value="91822222"></property>
    </bean>
    <bean id="accountDeatils"
        class="com.naresh.hello.AccountDetails">
        <constructor-arg name="name" value="Dilip" />
        <constructor-arg name="balance" value="500.00" />
        <constructor-arg name="mobiles">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+91-888888888</value>
                <value>+2323888888888</value>
            </set>
        </constructor-arg>
    </bean>

```

```
        <constructor-arg name="customerAddress" ref="addr" />
    </bean>
</beans>
```

- **Now Test Constructor Injection Beans and Configuration.**

```
package com.naresh.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringCoreApp {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\workspaces\\naresit\\spring_notes\\beans.xml");

        AccountDetails details = (AccountDetails) context.getBean("accountDeatils");

        System.out.println(details.getName());
        System.out.println(details.getBalance());
        System.out.println(details.getMobiles());
        System.out.println(details.getCustomerAddress().getFlatNo());
        System.out.println(details.getCustomerAddress().getHouseName());
    }
}
```

Output:

```
Dilip
500.0
[8826111377, +91-888888888, +232388888888]
333
Lotus Homes
```

Differences Between Setter and Constructor Injection.

Setter injection and constructor injection are two common approaches for implementing dependency injection. Here are the key differences between them:

1. Dependency Resolution: In setter injection, dependencies are resolved and injected into the target object using setter methods. In contrast, constructor injection resolves dependencies by passing them as arguments to the constructor.

2. Timing of Injection: Setter injection can be performed after the object is created, allowing for the possibility of injecting dependencies at a later stage. Constructor injection, on the other hand, requires all dependencies to be provided at the time of object creation.

3. Flexibility: Setter injection provides more flexibility because dependencies can be changed or modified after the object is instantiated. With constructor injection, dependencies are typically immutable once the object is created.

4. Required Dependencies: In setter injection, dependencies may be optional, as they can be set to null if not provided. Constructor injection requires all dependencies to be provided during object creation, ensuring that the object is in a valid state from the beginning.

5. Readability and Discoverability: Constructor injection makes dependencies more explicit, as they are declared as parameters in the constructor. This enhances the readability and discoverability of the dependencies required by a class. Setter injection may result in a less obvious indication of required dependencies, as they are set through individual setter methods.

6. Testability: Constructor injection is generally favored for unit testing because it allows for easy mocking or substitution of dependencies. By providing dependencies through the constructor, testing frameworks can easily inject mocks or stubs when creating objects for testing. Setter injection can also be used for testing, but it may require additional setup or manipulation of the object's state.

The choice between setter injection and constructor injection depends on the specific requirements and design considerations of your application. In general, constructor injection is recommended when dependencies are mandatory and should be set once during object creation, while setter injection provides more flexibility and optional dependencies can be set or changed after object instantiation.

Bean Wiring in Spring:

Bean wiring, also known as bean configuration or bean wiring configuration, is the process of defining the relationships and dependencies between beans in a container or application context. In bean wiring, you specify how beans are connected to each other, how dependencies are injected, and how the container should create and manage the beans. This wiring process is typically done through configuration files or annotations.

```
package com.naresh.hello;

public class AreaDeatils {

    private String street;
    private String pincode;

    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
}
```

```

    public String getPincode() {
        return pincode;
    }
    public void setPincode(String pincode) {
        this.pincode = pincode;
    }
}

```

- Create Another Bean : **Address**

```

package com.naresh.hello;

public class Address {

    private int flatNo;
    private String houseName;
    private long mobile;
    private AreaDeatils area; // Dependency Of Another
Class

    public AreaDeatils getArea() {
        return area;
    }
    public void setArea(AreaDeatils area) {
        this.area = area;
    }
    public int getFlatNo() {
        return flatNo;
    }
    public void setFlatNo(int flatNo) {
        this.flatNo = flatNo;
    }
    public String getHouseName() {
        return houseName;
    }
    public void setHouseName(String houseName) {
        this.houseName = houseName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
}

```

- Create Another Bean : **AccountDetails**

```

package com.naresh.hello;

```

```

import java.util.Set;

public class AccountDetails {

    private String name;
    private double balance;
    private Set<String> mobiles;
    private Address customerAddress;
    public AccountDetails(String name, double balance, Set<String> mobiles,
                           Address customerAddress) {

        super();
        this.name = name;
        this.balance = balance;
        this.mobiles = mobiles;
        this.customerAddress = customerAddress;
    }

    public AccountDetails() {

    }

    public Address getCustomerAddress() {
        return customerAddress;
    }

    public void setCustomerAddress(Address customerAddress) {
        this.customerAddress = customerAddress;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public Set<String> getMobiles() {
        return mobiles;
    }

    public void setMobiles(Set<String> mobiles) {
        this.mobiles = mobiles;
    }

}

```

- **Beans Configuration in spring xml file. With “ref” attribute we are configuring bean object each other internally.**

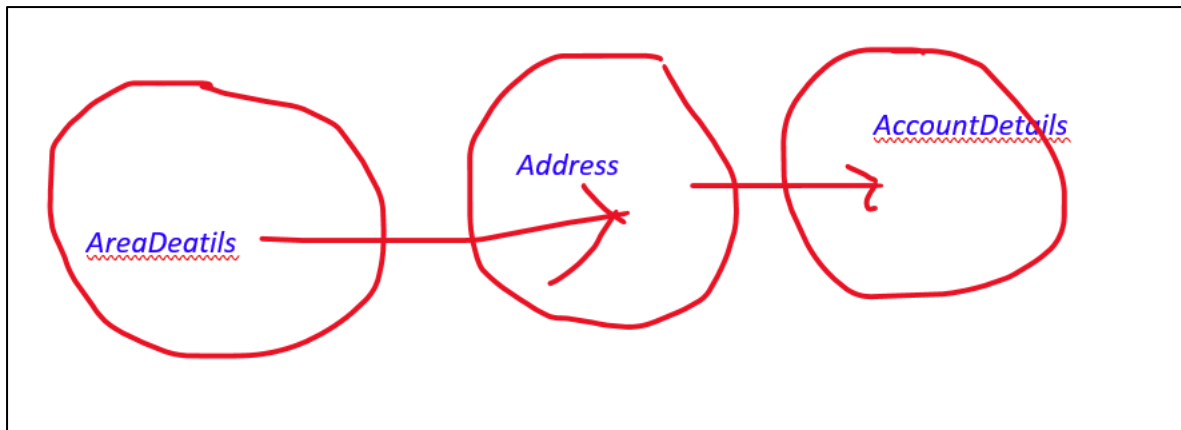
```
<?xml version="1.0" encoding="UTF-8"?>

<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 id="areaDetails" class="com.naresh.hello.AreaDeatils">
        <property name="street" value="Naresh It road"></property>
        <property name="pincode" value="323232"></property>
    </bean>

    <bean id="addr" class="com.naresh.hello.Address">
        <property name="flatNo" value="333"></property>
        <property name="houseName" value="Lotus Homes"></property>
        <property name="mobile" value="91822222"></property>
        <property name="area" ref="areaDetails"></property>
    </bean>

    <bean id="accountDeatils" class="com.naresh.hello.AccountDetails">
        <constructor-arg name="name" value="Dilip" />
        <constructor-arg name="balance" value="500.00" />
        <constructor-arg name="customerAddress" ref="addr" />
        <constructor-arg name="mobiles">
            <set>
                <value>8826111377</value>
                <value>8826111377</value>
                <value>+91-888888888</value>
                <value>+232388888888</value>
            </set>
        </constructor-arg>
    </bean>
</beans>
```



Testing of Bean Configuration:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringCoreApp {
    public static void main(String[] args) {

        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\workspaces\\naresit\\spring_notes\\beans.xml");

        AccountDetails details
            = (AccountDetails) context.getBean("accountDeatils");

        System.out.println(details.getName());
        System.out.println(details.getBalance());
        System.out.println(details.getMobiles());
        //Get Address Instance
        System.out.println(details.getCustomerAddress().getFlatNo());
        //Get Area Instance
        System.out.println(details.getCustomerAddress().getArea().getPincode());
    }
}
```

Output:

```
Dilip
500.0
[8826111377, +91-88888888, +232388888888]
333
323232
```

AutoWiring in Spring:

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

“**autowire**” attribute is used in XML <bean> configuration files to enable automatic dependency injection. It allows Spring to automatically wire dependencies between beans without explicitly specifying them in the XML file.

To use autowiring in an XML bean configuration file, you need to define the “**autowire**” attribute for a bean definition. The “**autowire**” attribute accepts different values to determine how autowiring should be performed. There are many autowiring modes.

1. **no** : This is the default value. It means no autowiring will be performed, and you need to explicitly specify dependencies using the appropriate XML configuration using property of constructor tags.
2. **byName** : The byName mode injects the object dependency according to name of the bean i.e. Bena ID. In such case, property name of class and bean ID must be same. It internally calls setter method. If a match is found, the dependency will be injected.
3. **byType**: The byType mode injects the object dependency according to type i.e. Data Type of Property. So property/variable name and bean name can be different int this case. It internally calls setter method. If a match is found, the dependency will be injected. If multiple beans are found, an exception will be thrown.
4. **constructor**: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

Here’s an examples of using the “**autowire**” attribute in an XML bean configuration file.

autowire=no:

For example, Define Product Class.

```
public class Product {  
  
    private String productId;  
    private String productName;  
  
    public String getProductId() {  
        return productId;  
    }  
    public void setProductId(String productId) {  
        this.productId = productId;  
    }  
    public String getProductName() {
```

```

        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
}

```

- Now Define Class **Order** which is having dependency of **Product** Object i.e. **Product** bean object should be injected to **Order**.

```

Package com.flipkart.orders;

import com.flipkart.product.Product;

public class Order {

    private String orderId;
    private double orderValue;
    private Product product;

    public Order() {
        System.out.println("Order Object Created by IOC");
    }
    public Order(String orderId, double orderValue, Product product) {
        super();
        this.orderId = orderId;
        this.orderValue = orderValue;
        this.product = product;
    }
    public String getOrderId() {
        return orderId;
    }
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
    public double getOrderValue() {
        return orderValue;
    }
    public void setOrderValue(double orderValue) {
        this.orderValue = orderValue;
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
}

```

```
}
```

- Now let's configure both Product and Order in side beans xml file.

```
<beans>
  <bean id="product" class="com.flipkart.product.Product">
    <property name="productid" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
  </bean>
  <bean id="order" class="com.flipkart.orders.Order" autowire="no">
    <property name="orderid" value="order1234"></property>
    <property name="orderValue" value="33000.00"></property>
  </bean>
</beans>
```

- Now Try to request Object of Order from IOC Container.

```
package com.flipkart.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import com.flipkart.orders.Order;
import com.flipkart.orders.OrdersManagement;
import com.flipkart.product.Product;

public class AutowiringDemo {
    public static void main(String[] args) {

        // IOC Container
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");

        Order order = (Order) context.getBean("order");
        // Product Object: Getting product Id
        System.out.println(order.getProduct().getProductId());
    }
}
```

- Now we got an exception, as shown below.

```
Product Object Created by IOC
Order Object Created by IOC
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"com.flipkart.product.Product.getProductId()" because the return value of
"com.flipkart.orders.Order.getProduct()" is null at
com.flipkart.main.AutowiringDemo.main(AutowiringDemo.java:22)
```

Means, Product and Order Object created by Spring but not injected product bean object automatically in side Order Object by IOC Container with setter injection internally. Because we used **autowire** mode as **no**. **By Default autowire value is "no"** i.e. Even if we are not given autowire attribute internally Spring Considers it as **autowire="no"**.

autowire="byName":

Now configure **autowire="byName"** in side beans xml file for order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this autowire mode, We are expecting dependency injection of objects by Spring instead of we are writing bean wiring with either using `<property>` and `<constructor-arg>` tags by using **ref** attribute. Means, eliminating logic of reference configurations.

As per **autowire="byName"**, Spring internally checks for a dependency bean objects which is matched to property names of Object. As per our example, Product is dependency for Order class.

Product class Bean ID = Property Name of Order class

```
<bean id="product" class="com.flipkart.product.Product">
  <property name="productId" value="101"></property>
  <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

```
public class Order {
    private String orderId;
    private double orderValue;
    private Product product;

    // setters , getters , constructors
}
```

- Internally Spring comparing as shown in above and injected product bean object to Order object.
- Beans Configuration:**

```
<beans>
  <bean id="product" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
  </bean>
  <bean id="order" class="com.flipkart.orders.Order" autowire="byName">
    <property name="orderId" value="order1234"></property>
    <property name="orderValue" value="33000.00"></property>
  </bean>
</beans>
```

- Now test our application.**

```
1 package com.flipkart.orders;
2
3 public class OrdersManagement {
4
5     private int noOfOrders;
6     private double totalAmount;
7     private Order order;
8
9     public OrdersManagement(int noOfOrders, double totalAmount, Order order)
10    {
11        super();
12        this.noOfOrders = noOfOrders;
13        this.totalAmount = totalAmount;
14        this.order = order;
15    }
16 }
```

Console ×
<terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-2023, 1:16:04)
Product Object Created by IOC
Order Object Created by IOC
101

So Internally Spring injected **Product object** by name of bean and **property name** of Order class.

Question: If property name and Bean ID are different, then Spring will not inject Product object inside Order Object. Now I made bean id as prod for Product class.

```
<bean id="prod" class="com.flipkart.product.Product">
  <property name="productId" value="101"></property>
  <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

Not Matching

```
public class Order {
    private String orderId;
    private double orderValue;
    private Product product;

    // setters , getters, constructors
}
```

Test Our application and check Spring injected Product object or not inside Order.

Product Object Created by IOC

Order Object Created by IOC

Exception in thread "main" [java.lang.NullPointerException](#): Cannot invoke "com.flipkart.product.Product.getProductId()" because the return value of "com.flipkart.orders.Order.getProduct()" is null at com.flipkart.main.AutowiringDemo.main([AutowiringDemo.java:19](#))

autowire="byType":

Now configure **autowire="byType"** in side beans xml file for Order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this

autowire mode, We are expecting dependency injected by Spring instead of we are writing bean wiring with either using <property> and <constructor-arg> tags by using **ref** attribute. Means, eliminating logic of reference configurations.

As per **autowire="byType"**, Spring internally checks for a dependency bean objects, which are matched with Data Type of property and then that bean object will be injected. In this case Bean ID and Property Names may be different. As per our example, Product is dependency for Order class.

Bean Data Type i.e. class Name = Data type of property of Order class

```
<bean id="prod" class="com.flipkart.product.Product">
  <property name="productId" value="101"></property>
  <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

```
public class Order {

    private String orderId;
    private double orderValue;
    private Product product;

    // setters , getters, constructors
}
```

- **Beans Configuration:**

```
<beans>
  <bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
  </bean>
  <bean id="order" class="com.flipkart.orders.Order" autowire="byType">
    <property name="orderId" value="order1234"></property>
    <property name="orderValue" value="33000.00"></property>
  </bean>
</beans>
```

- **Test Our Application Now:** Dependency Injected Successfully, because only One Product Object available.

```

3 import org.springframework.context.ApplicationContext;
9
10 public class AutowiringDemo {
11
12     public static void main(String[] args) {
13         ApplicationContext context = new FileSystemXmlApplicationContext(
14             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
15
16         Order order = (Order) context.getBean("order");
17         System.out.println(
18             order.getProduct() // Product Object
19             .getProductId() // Product object : Getting product Id
20         );
21     }
22 }
23

```

Console x

<terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-2023, 1:38:27 pm -

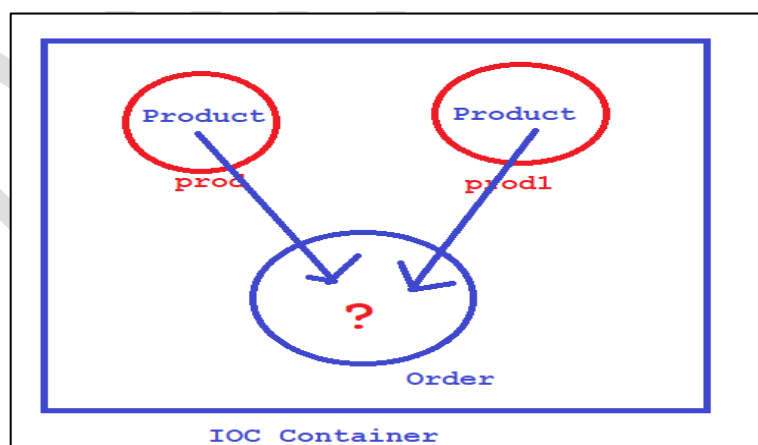
Product Object Created by IOC
Order Object Created by IOC
101

Question: If Product Bean configured more than one time inside beans configuration, then which Product Bean Object will be injected in side Order ?

```

<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>

```



Test Our Application: We will get Exception while trying to inject Product Object because of ambiguity between 2 Objects.

Product Object Created by IOC
Product Object Created by IOC
Order Object Created by IOC

Jul 11, 2023 1:56:23 PM org.springframework.context.support.AbstractApplicationContext refresh

WARNING: Exception encountered during context initialization - cancelling refresh attempt: [org.springframework.beans.factory.UnsatisfiedDependencyException](#): Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#): No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

Exception in thread "main"
[org.springframework.beans.factory.UnsatisfiedDependencyException](#): Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#): No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

autowire="constructor":

Now configure **autowire="constructor"** in side beans xml file for Order Bean configuration, because internally Product bean object should be injected to Order Bean Object. In this autowire mode, We are expecting dependency injected by Spring instead of we are writing bean wiring with either using <property> or <constructor-arg> tags by using **ref** attribute. Means, eliminating logic of reference configurations.

As per **autowire="constructor"**, Spring internally checks for a dependency bean objects, which are matched with **constructor argument** of same data type and then that bean object will be injected. Means, constructor autowire mode works with constructor injection not setter injection. In this case, injected Bean Type and Constructor Property Name should be same.

As per our example, Product is dependency for Order and Order class defined a constructor with parameter contains Product type.


```
<bean id="prod" class="com.flipkart.product.Product">
  <property name="productId" value="101"></property>
  <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="order" class="com.flipkart.orders.Order" autowire="constructor">
  <constructor-arg index="0" value="order1234"></constructor-arg>
  <constructor-arg index="1" value="33000"></constructor-arg>
</bean>
```

```
public class Order {

    private String orderId;
    private double orderValue;
    private Product product;

    public Order(String orderId, double orderValue, Product product) {
        super();
        this.orderId = orderId;
        this.orderValue = orderValue;
        this.product = product;
    }

}
```

Beans Configuration:

```
<bean id="prod" class="com.flipkart.product.Product">
  <property name="productId" value="101"></property>
  <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="order" class="com.flipkart.orders.Order" autowire="constructor">
  <constructor-arg index="0" value="order1234"></constructor-arg>
  <constructor-arg index="1" value="33000"></constructor-arg>
</bean>
```

- **Test Our Application: Now Product Object Injected via Constructor.**

```
10 public class AutowiringDemo {
11
12     public static void main(String[] args) {
13         ApplicationContext context = new FileSystemXmlApplicationContext(
14             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
15
16         Order order = (Order) context.getBean("order");
17         System.out.println(
18             order.getProduct() // Product Object
19             .getProductId() // Product object : Getting product Id
20         );
21     }
22 }
23
```

Console ×

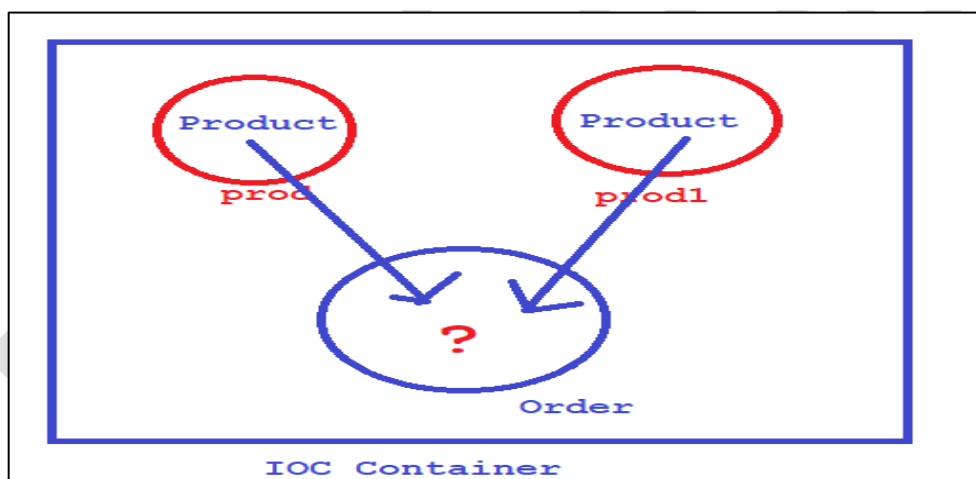
```
<terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-20
Product Object Created by IOC
Order Object Created: With Params Constructor
101
```

Question: If Product Bean configured more than one time inside beans configuration, then which Product Bean Object will be injected in side Order?

when **autowire =constructor**, spring internally checks out of multiple bean ids dependency object which is matching with property name of Order class. If matching found then that specific bean object will be injected. If not found then we will get ambiguity exception as following.

As per our below configuration, both bean ids of Product are not matching with Order class property name of Product type.

```
<bean id="prod" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```



- **Test Our Application:** We will get Exception while trying to inject Product Object because of ambiguity between 2 Objects.

Product Object Created by IOC

Product Object Created by IOC

Jul 11, 2023 1:56:23 PM org.springframework.context.support.AbstractApplicationContext refresh

WARNING: Exception encountered during context initialization - cancelling refresh attempt:

[org.springframework.beans.factory.UnsatisfiedDependencyException](#): Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#): No qualifying bean

of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2
Exception in thread "main"
[org.springframework.beans.factory.UnsatisfiedDependencyException](#): Error creating bean with name 'order' defined in file [D:\workspaces\naresit\bean-wiring\beans.xml]: Unsatisfied dependency expressed through bean property 'product'; nested exception is [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#): No qualifying bean of type 'com.flipkart.product.Product' available: expected single matching bean but found 2: prod,prod2

Now if we configure one bean object of **Product** class with bean id which is matching with property name of **Order** class. Then that Specific Object will be injected. From following configuration **Product** object of bean id "**product**" will be injected.

```
<bean id="product" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
<bean id="prod2" class="com.flipkart.product.Product">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

• Test Our Application :

```
10 public class AutowiringDemo {
11
12     public static void main(String[] args) {
13         ApplicationContext context = new FileSystemXmlApplicationContext(
14             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
15
16         Order order = (Order) context.getBean("order");
17         System.out.println(
18             order.getProduct() // Product Object
19             .getProductId() // Product object : Getting product Id
20         );
21     }
22 }
23
```

Console ×

<terminated> AutowiringDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (11-Jul-2023)

Product Object Created by IOC

Order Object Created: With Params Constructor

101

Advantage of Autowiring:

- It requires less code because we don't need to write the code to inject the dependency explicitly.

Disadvantages of Autowiring:

- No control of the programmer.

- It can't be used for primitive and string values.

Bean Scopes in Spring:

When you start a Spring application, the Spring Framework creates beans for you. These Spring beans can be application beans that you have defined or beans that are part of the framework. When the Spring Framework creates a bean, it associates a scope with the bean. **A scope defines the life cycle and visibility of that bean within runtime application context which the bean instance is available.**

The Latest Spring Framework supports 5 scopes, last four are available only if you use Web aware of ApplicationContext i.e. inside Web applications.

1. singleton
2. prototype
3. request
4. session
5. application
6. websocket

Defining Scope of beans syntax:

In XML configuration, we will use an attribute “**scope**”, inside **<bean>** tag as shown below.

```
<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

singleton:

This is **default scope** of a bean configuration i.e. even if we are not provided scope attribute as part of any bean configuration, then spring container internally considers as **scope="singleton"**.

If Bean **scope=singleton**, then Spring Container creates only one object in side Spring container overall application level and Spring Container returns same instance reference always for every IOC container call i.e. `getBean()`.

```
<bean id="productOne" class="com.flipkart.product.Product" scope="singleton">
```

Above line is equal to following because default is scope="singleton"

```
<bean id="productOne" class="com.flipkart.product.Product">
```

Now create Bean class and call IOC container many times with same bean ID.

Product.java

```
public class Product {
    private String productId;
    private String productName;

    public Product() {
        System.out.println("Product Object Created by IOC");
    }

    //setters and getters methods
}
```

- Now configure above class in side beans xml file.

```
<bean id="product" class="com.flipkart.product.Product" scope="singleton">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>
```

- Now call Get Bean from IOC Container for Product Bean Object. In Below, we are calling IOC container 3 times.

```
public class BeanScopesDemo {

    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
                                                                    "D:\\beans.xml");

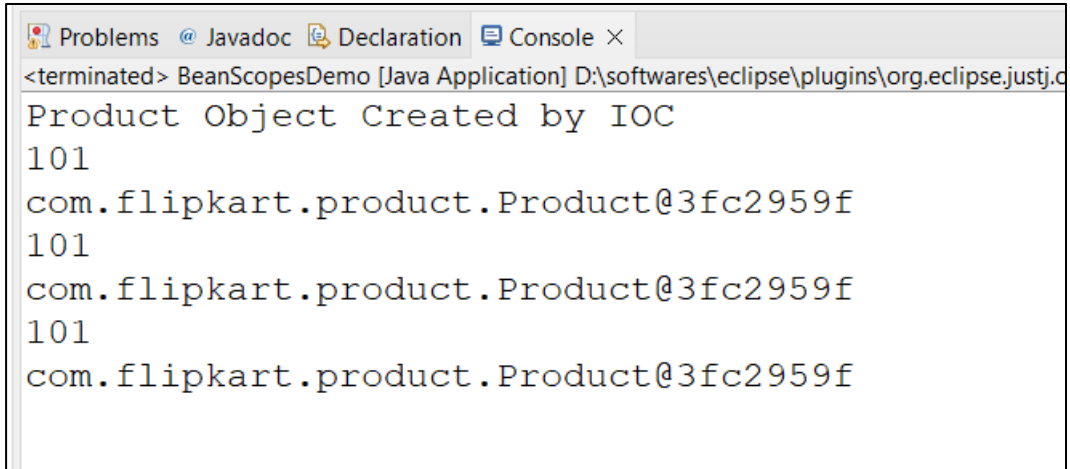
        // 1. Requesting/Calling IOC Container for Product Object
        Product p1 = (Product) context.getBean("product");
        System.out.println(p1.getProductId());
        System.out.println(p1);

        // 2. Requesting/Calling IOC Container for Product Object
        Product p2 = (Product) context.getBean("product");
        System.out.println(p2.getProductId());
        System.out.println(p2);

        // 3. Requesting/Calling IOC Container for Product Object
        Product p3 = (Product) context.getBean("product");
        System.out.println(p3.getProductId());
        System.out.println(p3);
    }
}
```

```
}  
}
```

Output:



```
<terminated> BeanScopesDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.c  
Product Object Created by IOC  
101  
com.flipkart.product.Product@3fc2959f  
101  
com.flipkart.product.Product@3fc2959f  
101  
com.flipkart.product.Product@3fc2959f
```

From above output, Spring Container created only one Object and same passed for every new container call with `getBean()` by passing bean id. Means, Singleton Object created for bean ID **product** in IOC container.

NOTE: If we created another bean id configuration for same class, then previous configuration behaviour will not applicable to current configuration i.e. every individual bean configuration or Bean Object having it's own behaviour and functionality in Spring Framework.

Create one more bean configuration of Product.

```
<bean id="product" class="com.flipkart.product.Product" scope="singleton">  
    <property name="productId" value="101"></property>  
    <property name="productName" value="Lenevo Laptop"></property>  
</bean>  
<bean id="productTwo" class="com.flipkart.product.Product">  
    <property name="productId" value="102"></property>  
    <property name="productName" value="HP Laptop"></property>  
</bean>
```

- **Testing:** In below we are requesting 2 different bean objects of Product.

```
public class BeanScopesDemo {  
    public static void main(String[] args) {  
  
        ApplicationContext context = new FileSystemXmlApplicationContext(  
            "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");  
  
        // 1. Requesting/Calling IOC Container for Product Object  
        Product p1 = (Product) context.getBean("product");  
        System.out.println(p1.getProductId());  
    }  
}
```

```
System.out.println(p1);
```

```
// 2. Requesting/Calling IOC Container for Product Object
```

```
Product p2 = (Product) context.getBean("product");  
System.out.println(p2.getProductId());  
System.out.println(p2);
```

```
// 3. Requesting/Calling IOC Container for Product Object
```

```
Product p3 = (Product) context.getBean("productTwo");  
System.out.println(p3.getProductId());  
System.out.println(p3);
```

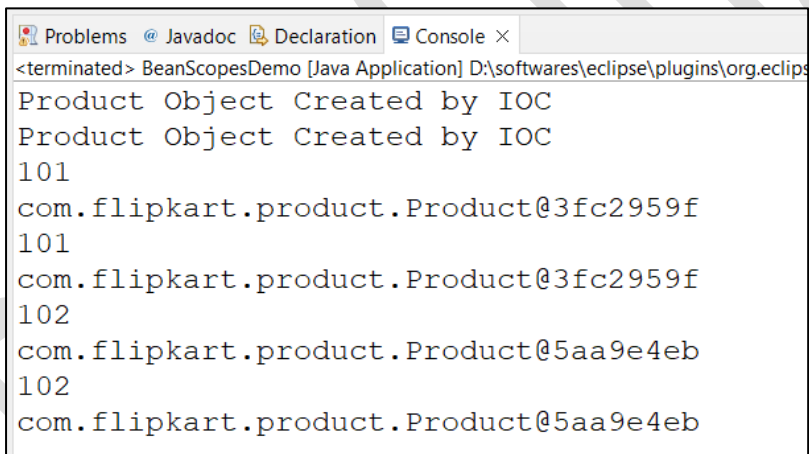
```
// 4. Requesting/Calling IOC Container for Product Object
```

```
Product p4 = (Product) context.getBean("productTwo");  
System.out.println(p4.getProductId());  
System.out.println(p4);
```

```
}
```

```
}
```

Output:



```
<terminated> BeanScopesDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse  
Product Object Created by IOC  
Product Object Created by IOC  
101  
com.flipkart.product.Product@3fc2959f  
101  
com.flipkart.product.Product@3fc2959f  
102  
com.flipkart.product.Product@5aa9e4eb  
102  
com.flipkart.product.Product@5aa9e4eb
```

For 2 Bean configurations of Product class, 2 individual Singleton Bean Objects created.

prototype:

If Bean scope defined as “**prototype**”, a new instance of the bean is created every time it is requested from the container. It is not cached, so each request/call to IOC container for the bean will return in a new instance.

Bean class: Product.java

```
public class Product {  
  
    private String productId;  
    private String productName;
```

```

    public Product() {
        System.out.println("Product Object Created by IOC");
    }

    //setters and getters
}

```

XML Bean configuration:

```

<bean id="product" class="com.flipkart.product.Product" scope="prototype">
    <property name="productId" value="101"></property>
    <property name="productName" value="Lenevo Laptop"></property>
</bean>

```

Testing :

```

1  /
2  /
3  /
4  /
5  /
6  /
7  /
8  public class BeanScopesDemo {
9      public static void main(String[] args) {
10         ApplicationContext context = new FileSystemXmlApplicationContext(
11             "D:\\workspaces\\naresit\\bean-wiring\\beans.xml");
12         // 1. Requesting/Calling IOC Container for Product Object
13         Product p1 = (Product) context.getBean("product");
14         System.out.println(p1);
15         System.out.println(p1.getProductId());
16         // 2. Requesting/Calling IOC Container for Product Object
17         Product p2 = (Product) context.getBean("product");
18         System.out.println(p2);
19         System.out.println(p2.getProductId());
20     }
21 }

```

Problems | Javadoc | Declaration | Console x

```

<terminated> BeanScopesDemo [Java Application] D:\softwares\ eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (12
Product Object Created by IOC
com.flipkart.product.Product@5542c4ed
101
Product Object Created by IOC
com.flipkart.product.Product@1573f9fc
101

```

Now Spring Container created and returned every time new Bean Object for every Container call `getBean()` for same bean ID.

request:

When we apply scope as request, then for every new HTTP request Spring will crates new instance of configured bean. Only valid in the context of a web-aware Spring ApplicationContext i.e. in web/MVC applications.

session:

When we apply scope as session, then for every new HTTP session creation in server side Spring will create new instance of configured bean. Only valid in the context of a web-aware Spring ApplicationContext i.e. in web/MVC applications.

application:

Once you have defined the application-scoped bean, Spring will create a single instance of the bean per web application context. Any requests for this bean within the same web application will receive the same instance.

It's important to note that the application scope is specific to web applications and relies on the lifecycle of the web application context. Each web application running in a container will have its own instance of the application-scoped bean.

You can use application-scoped beans to store and share application-wide state or resources that need to be accessible across multiple components within the same web application.

websocket:

This is used as part of socket programming. We can't use it in our Servlet based MVC application level.

Java/Annotation Based Beans Configuration:

So far we have seen how to configure Spring beans using XML configuration file. Java-based configuration option enables you to write most of your Spring configuration without XML but with the help of annotations.

@Configuration & @Bean Annotations:

@Configuration:

In Java Spring, the **@Configuration** annotation is used to indicate that this class is a configuration class of Beans. A configuration class is responsible for defining beans and their dependencies in the Spring application context. Beans are objects that are managed by the Spring IOC container. Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IOC container as a source of bean definitions.

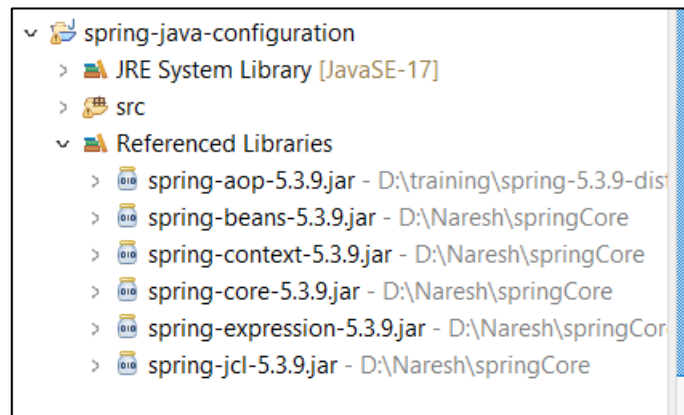
Create a Java class and annotate it with **@Configuration**. This class will serve as our configuration class.

@Bean:

In Spring, the **@Bean** annotation is used to declare a method as a bean definition method within a configuration class. The **@Bean** annotation tells Spring that a method annotated with **@Bean** will return an object that should be registered as a bean in the Spring

application context. The method annotated with **@Bean** is responsible for creating and configuring an instance of a bean that will be managed by the Spring IoC (Inversion of Control) container.

- Now Create a Project and add below jars to support Spring Annotations of Core Module.



NOTE: Added one extra jar file comparing with previous project setup. Because internally Spring core module using AOP functionalities to process annotations.

- Now Create a Bean class : **UserDetails**

```
package com.amazon.users;

public class UserDetails {

    private String firstName;
    private String lastName;
    private String emailId;
    private String password;
    private long mobile;

    //setters and getters

}
```

Now Create a Beans Configuration class. i.e. Class Marked with an annotation **@Configuration**. In side this configuration class, we will define multiple bean configurations with **@Bean** annotation methods.

Configuration class would be as follows:

```
package com.amazon.config;
```

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.amazon.users.UserDetails;

@Configuration
public class BeansConfiguration {
    @Bean("userDetails")
    UserDetails getUserDetails() {
        return new UserDetails();
    }
}

```

The above code will be equivalent to the following XML bean configuration –

```

<beans>
    <bean id = "userDetails" class = "com.amazon.users.UserDetails" />
</beans>

```

Here, the method is annotated with `@Bean("userDetails")` works as bean ID is `userDetails` and Spring creates bean object with that bean ID and returns the same bean object when we call `getBean()`. Your configuration class can have a declaration for more than one `@Bean`. Once your configuration classes are defined, you can load and provide them to Spring container using `AnnotationConfigApplicationContext` as follows .

```

package com.amazon;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.users.UserDetails;

public class SpringBeanMainApp {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(BeansConfiguration.class);

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);
        context.close();
    }
}

```

Output: Prints hash code of Object

`com.amazon.users.UserDetails@34bde49d`

Understand From above code :

AnnotationConfigApplicationContext:

AnnotationConfigApplicationContext is a class provided by the Spring Framework that serves as an implementation of the **ApplicationContext** interface. It is used to create an application context container by reading Java-based configuration metadata. In Spring, there are multiple ways to configure the application context, such as XML-based configuration or Java-based configuration using annotations.

AnnotationConfigApplicationContext is specifically used for Java-based configuration. It allows you to bootstrap the Spring container by specifying one or more Spring configuration classes that contain **@Configuration** annotations.

We will provide configuration classes as Constructor parameters of **AnnotationConfigApplicationContext** or spring provided **register()** method also. We will have example here after.

- **context.getBean()**, Returns an instance, which may be shared or independent, of the specified bean.
- **context.close()**, Close this application context, destroying all beans in its bean factory.

Multiple Configuration Classes and Beans:

Now we can configure multiple bean classes inside multiple configuration classes as well as Same bean with multiple bean id's.

Now I am crating one more Bean class in above application.

```
package com.amazon.products;

public class ProductDetails {

    private String pname;
    private double price;

    public String getPname() {
        return pname;
    }
    public void setName(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

```
}
```

Configuring above POJO class as Bean class inside Beans Configuration class.

```
package com.amazon.config;

import org.springframework.context.annotation.Bean;
import com.amazon.products.ProductDetails;

public class BeansConfigurationTwo {

    @Bean("productDetails")
    ProductDetails productDetails() {
        return new ProductDetails();
    }

}
```

➤ Testing Bean Object Created or not. Below Code loading Two Configuration classes.

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.config.BeansConfigurationTwo;
import com.amazon.products.ProductDetails;
import com.amazon.users.UserDetails;
public class SpringBeanMainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class); //UserDetails Bean Config.
        context.register(BeansConfigurationTwo.class); //ProductDetails Bean Config.
        context.refresh();

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);
        ProductDetails product = (ProductDetails) context.getBean("productDetails");
        System.out.println(product);
        context.close();
    }

}
```

Now Create multiple Bean Configurations for same Bean class.

➤ **Inside Configuration class:** Two Bean configurations for **ProductDetails** Bean class.

```
import org.springframework.context.annotation.Bean;
import com.amazon.products.ProductDetails;

public class BeansConfigurationTwo {

    @Bean("productDetails")
    ProductDetails productDetails() {
        return new ProductDetails();
    }

    @Bean("productDetailsTwo")
    ProductDetails productTwoDetails() {
        return new ProductDetails();
    }

}
```

➤ **Now get Both bean Objects of ProductDetails.**

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfiguration;
import com.amazon.config.BeansConfigurationTwo;
import com.amazon.products.ProductDetails;
import com.amazon.users.UserDetails;

public class SpringBeanMainApp {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.register(BeansConfigurationTwo.class);
        context.refresh();

        UserDetails user = (UserDetails) context.getBean("userDetails");
        System.out.println(user);

        ProductDetails product = (ProductDetails) context.getBean("productDetails");
        System.out.println(product);

    }

}
```

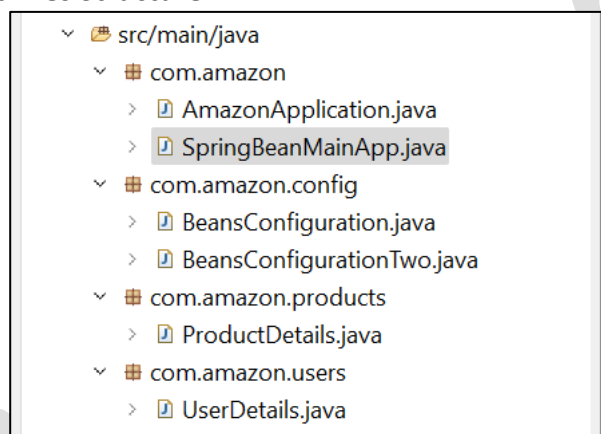
```
ProductDetails productTwo = (ProductDetails) context.getBean("productDetailsTwo");
System.out.println(productTwo);
context.close();
}
}
```

Output:

```
com.amazon.users.UserDetails@7d3e8655
com.amazon.products.ProductDetails@7dfb0c0f
com.amazon.products.ProductDetails@626abbd0
```

From above Output Two **ProductDetails** bean objects created by Spring Container.

Above Project Files Structure:



@Component Annotation :

Before we can understand the value of **@Component**, we first need to understand a little bit about the Spring **ApplicationContext**.

Spring **ApplicationContext** is where Spring holds instances of objects that it has identified to be managed and distributed automatically. These are called beans. Some of Spring's main features are bean management and dependency injection. Using the Inversion of Control principle, **Spring collects bean instances from our application and uses them at the appropriate time**. We can show bean dependencies to Spring without handling the setup and instantiation of those objects.

However, the base/regular spring bean definitions are explicitly defined in the XML file or configured in configuration class with **@Bean**, while the annotations drive only the dependency injection. This section describes an option for implicitly/internally detecting the candidate components by scanning the classpath. Components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, **@Component**) to select which classes have bean definitions registered with the container.

We should take advantage of Spring's automatic bean detection by using stereotype annotations in our classes.

@Component: This annotation that allows Spring to detect our custom beans automatically.

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
- Inject them wherever needed

We have other more specialized stereotype annotations like **@Controller**, **@Service** and **@Repository** to serve this functionality derived, we will discuss then in MVC module level.

Define Spring Components :

1. Create a java Class and provide an annotation **@Component** at class level.

```
package com.tek.teacher;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
public class Product {

    private String pname;
    private double price;

    public Product(){
        System.out.println("Product Object Created.");
    }
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

➤ Now Test in Main Class.


```

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

        context.scan("com.tek. teacher");
        context.refresh();

        Product details = (Product) context.getBean(Product.class);
        System.out.println(details);

    }
}

```

From Above program, **context.scan()** method, Perform a scan for @Component classes to instantiate Bean Objects within the specified base packages. We can pass many package names wherever we have @Component Classes. Note that, **scan(basePackages)** method will scans @Configuraion classes as well from specified package names. Note that **refresh()** must be called in order for the context to fully process the new classes.

Spring Provided One more way which used mostly in Real time applications is using **@ComponentScan** annotation. To enable auto detection of Spring components, we shou use another annotation **@ComponentScan**.

@ComponentScan:

Before we rely entirely on *@Component*, we must understand that it's only a plain annotation. The annotation serves the purpose of differentiating beans from other objects, such as domain objects. However, Spring uses the *@ComponentScan* annotation to gather all component into its **ApplicationContext**.

@ComponentScan annotation is used to specify packages for spring to scan for annotated components. Spring needs to know which packages contain beans, otherwise you would have to register each bean individually. Hence **@ComponentScan** annotation is a supporting annotation for **@Configuration** annotation. Spring instantiate Bean Objects of components from specified packages for those classes annotated with **@Component**.

So create a beans configuration class i.e. **@Configuration** annotated class and provide **@ComponentScan** with base package name.

Ex: When we have to scan multiple packages we can pass all package names as String array with attribute **basePackages** .

```

@ComponentScan (basePackages =
    { "com.hello.spring.*", "com.hello.spring.boot.*" } )

```

Or If only one base package and it's sub packages should be scanned, then we can directly pass package name.

```
@ComponentScan("com.hello.spring.*")
```

Test our component class:

- Create A **@Configuration** class with **@ComponentScan** annotation.

```
@Configuration
//making sure scanning all packages starts with com.tek.teacher
@ComponentScan("com.tek.teacher.*")
public class BeansConfiguration {

}
```

- Now Load/pass above configuration class to Application Context i.e. Spring Container.

```
package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfiguration.class);
        context.refresh();
        Product details = (Product) context.getBean(Product.class);
        System.out.println(details);
    }
}
```

- Now Run your Main class application.

Output: **ProductDetails [pname=null, price=0.0]**

From above, Spring Container detected **@Component** classes from all packages and instantiated as Bean Objects.

Now Add One More @Component class:

```
package com.tek.teacher;

import org.springframework.stereotype.Component;
```

@Component

```
public class UserDetails {  
  
    private String firstName;  
    private String lastName;  
    private String emailId;  
    private String password;  
    private long mobile;  
  
    public UserDetails(){  
        System.out.println("UserDetails Object Created");  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
    public String getLastName() {  
        return lastName;  
    }  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
    public String getEmailId() {  
        return emailId;  
    }  
    public void setEmailId(String emailId) {  
        this.emailId = emailId;  
    }  
    public String getPassword() {  
        return password;  
    }  
    public void setPassword(String password) {  
        this.password = password;  
    }  
    public long getMobile() {  
        return mobile;  
    }  
    public void setMobile(long mobile) {  
        this.mobile = mobile;  
    }  
}
```

➤ Now get UserDetails from Spring Container and Test/Run our Main class.

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.refresh();
        //UserDetails Component
        UserDetails userDetails = context.getBean(UserDetails.class);
        System.out.println(userDetails);
    }
}

```

Output: com.tek.teacher.UserDetails@5e17553a

NOTE: In Above Logic, used **getBean(Class<UserDetails> requiredType)**, Return the bean instance that uniquely matches the given object type, if any. Means, when we are not configured any component name or don't want to pass bean name from **getBean()** method.

We can use any of overloaded method **getBean()** to get Bean Object as per our requirement or functionality demanding.

Can we Pass Bean Scope to @Component Classes?

Yes, Similar to **@Bean** annotation level however we are assigning scope type , we can pass in same way with **@Component** class level because Component is nothing but Bean finally.

Ex : From above example, requesting another Bean Object of type UserDetails without configuring scope at component class level.

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.refresh();

        // UserDetails Component
    }
}

```

```

        UserDetails userDetails = context.getBean(UserDetails.class);
        System.out.println(userDetails);

        UserDetails userTwo = context.getBean(UserDetails.class);
        System.out.println(userTwo);
    }
}

```

Output:

```

com.tek.teacher.UserDetails@5e17553a
com.tek.teacher.UserDetails@5e17553a

```

So we can say by default component classes are instantiated as singleton bean object, when there is no scope defined. Means, Internally Spring Container considering as **singleton scope**.

Question : Can we create @Bean configurations for @Component class?

Yes, We can create Bean Configurations in side Spring Configuration classes. With That Bean ID, we can request from Application Context, as usual.

Inside Configuration Class:

```

package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.tek.*")
public class BeansConfiguration {

    @Bean("user")
    UserDetails getUserDetails() {
        return new UserDetails();
    }
}

```

➤ **Testing from Main Class:**

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
    }
}

```

```

        context.register(BeansConfiguration.class);
        context.refresh();

        // UserDetails Component
        UserDetails userThree = (UserDetails) context.getBean("user");
        System.out.println(userThree);
    }
}

```

Output: com.tek.teacher.UserDetails@3eb91815

Question : How to pass default values to @Component class properties?

We can pass/initialize default values to a component class instance with **@Bean** method implementation inside Spring Configuration classes.

Inside Configuration Class:

```

package com.tek.teacher;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.tek.*")
public class BeansConfiguration {

    @Bean("user")
    UserDetails getUserDetails() {
        UserDetails user = new UserDetails();
        user.setEmailId("dilip@gmail.com");
        user.setMobile(8826111377l);
        return user;
    }
}

```

Main App:

```

package com.tek.teacher;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();
    }
}

```

```

        context.register(BeansConfiguration.class);
        context.refresh();

        // UserDetails Component
        UserDetails userThree = (UserDetails) context.getBean("user");
        System.out.println(userThree.getEmailId());
        System.out.println(userThree.getMobile());
    }
}

```

Output:

```

dilip@gmail.com
8826111377

```

Defining Scope of beans with Annotations:

In XML configuration, we will use an attribute “**scope**”, inside **<bean>** tag as shown below.

```

<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>

```

Annotation Based Scope Configuration:

We will use **@Scope** annotation will be used to define scope type.

@Scope: A bean’s scope is set using the **@Scope** annotation. By default, the Spring framework creates exactly one instance for each bean declared in the IoC container. This instance is shared in the scope of the entire IoC container and is returned for all subsequent **getBean()** calls and bean references.

Example: Create a bean class and configure with Spring Container : **ProductDetails.java**

```

package com.amazon.products;

public class ProductDetails {
    private String pname;
    private double price;
    public String getPname() {
        return pname;
    }
    public void setName(String pname) {
        this.pname = pname;
    }
    public double getPrice() {
        return price;
    }
}

```

```

    }
    public void setPrice(double price) {
        this.price = price;
    }
    public void printProductDetails() {
        System.out.println("Product Details Are : .....");
    }
}

```

Now Inside Configuration class, Define Bean Creation and Configure scope value.

Singleton Scope:

A single Bean object instance created and returns same Bean instance for each Spring IoC container call i.e. **getBean()**. In side Configuration class, **scope** value defined as **singleton**.

NOTE: If we are not defined any scope value for any Bean Configuration, then Spring Container by default considers scope as **singleton**.

```

package com.amazon.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import com.amazon.products.ProductDetails;

@Configuration
public class BeansConfigurationThree {
    @Scope("singleton")
    @Bean("productDetails")
    ProductDetails getProductDetails() {
        return new ProductDetails();
    }
}

```

- Now Test Bean **ProductDetails** Object is singleton or not. Request multiple times **ProductDetails** Object from Spring Container by passing bean id **productDetails**.

```

package com.amazon;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfigurationThree;
import com.amazon.products.ProductDetails;

public class SpringBeanScopeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
    }
}

```



```

context.register(BeansConfigurationThree.class);
context.refresh();
ProductDetails productOne = (ProductDetails) context.getBean("productDetails");
System.out.println(productOne);
ProductDetails productTwo = (ProductDetails) context.getBean("productDetails");
System.out.println(productTwo);
context.close();
}
}

```

Output:

```

com.amazon.products.ProductDetails@58e1d9d
com.amazon.products.ProductDetails@58e1d9d

```

From above output, we can see same hash code printed for both `getBean()` calls on Spring Container. Means, Container created singleton instance for bean id "`productDetails`".

Prototype Scope: In side Configuration class, scope value defined as **prototype**.

```

package com.amazon.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import com.amazon.products.ProductDetails;

@Configuration
public class BeansConfigurationThree {
    @Scope("singleton")
    @Bean("productDetails")
    ProductDetails getProductDetails() {
        return new ProductDetails();
    }

    @Scope("prototype")
    @Bean("productTwoDetails")
    ProductDetails getProductTwoDetails() {
        return new ProductDetails();
    }
}

```

- Now Test Bean **ProductDetails** Object is **prototype** or not. Request multiple times **ProductDetails** Object from Spring Container by passing bean id `productTwoDetails`.

```

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.amazon.config.BeansConfigurationThree;
import com.amazon.products.ProductDetails;

```

```

public class SpringBeanScopeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfigurationThree.class);
        context.refresh();
        //Prototype Beans:
        ProductDetails productThree =
            (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productThree);
        ProductDetails productFour =
            (ProductDetails) context.getBean("productTwoDetails");
        System.out.println(productFour);
        context.close();
    }
}

```

Output:

```

com.amazon.products.ProductDetails@12591ac8
com.amazon.products.ProductDetails@5a7fe64f

```

From above output, we can see different hash codes printed for both `getBean()` calls on Spring Container. Means, Container created new instance every time when we requested for instance of bean id "`productTwoDetails`".

Scope of @Component classes:

If we want to define scope of with component classes and Objects externally, then we will use same **@Scope** at class level of component class similar to **@Bean** method level in previous examples.

If we are not passed any scope value via **@Scope** annotation to a component class, then Component Bean Object will be created as singleton as usually.

Now for **UserDetails** class, added scope as **prototype**.

```

@Scope("prototype")
@Component
public class UserDetails {
    //Properties
    //Setter & Getters
    // Methods
}

```

Now test from Main application class, whether we are getting new Instance or not for every request of Bena Object **UserDetails** from Spring Container.

```

package com.tek.teacher.products;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringComponentDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.scan("com.tek.*");
        context.refresh();

        // UserDetails Component
        UserDetails userDetails = context.getBean(UserDetails.class);
        System.out.println(userDetails);

        UserDetails userTwo = context.getBean(UserDetails.class);
        System.out.println(userTwo);
    }
}

```

Output:

```

com.tek.teacher.UserDetails@74f6c5d8
com.tek.teacher.UserDetails@27912e3

```

NOTE: Below four are available only if you use a web-aware **ApplicationContext** i.e. inside Web applications.

- request
- session
- application
- globalsession

Auto Wiring In Spring

- Autowiring feature of spring framework enables you to inject the object dependency implicitly.
- Autowiring can't be used to inject primitive and string values. It works with reference only.
- It requires the less code because we don't need to write the code to inject the dependency explicitly.
- Autowired is allows spring to resolve the collaborative beans in our beans. Spring boot framework will enable the automatic injection dependency by using declaring all the dependencies in the configurations.
- We will achieve auto wiring with an Annotation **@Autowired**

Auto wiring will be achieved in multiple ways/modes.

Auto Wiring Modes:

- no
- byName
- byType
- constructor

- ✚ **no:** It is the default autowiring mode. It means no autowiring by default.
- ✚ **byName:** The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
- ✚ **byType:** The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
- ✚ **constructor:** The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

In XML configuration, we will enable auto wiring between Beans as shown below.

```
<bean id="a" class="org.sssit.A" autowire="byName">
    .....
</bean>
```

In Annotation Configuration, we will use **@Autowired** annotation.

We can use **@Autowired** in following methods.

1. On properties
2. On setter
3. On constructor

@Autowired on Properties

Let's see how we can annotate a property using *@Autowired*. This eliminates the need for getters and setters.

First, Let's Define a bean : Address.

```
package com.dilip.account;

import org.springframework.stereotype.Component;
```

```

@Component
public class Address {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }
    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    @Override
    public String toString() {
        return "Address [streetName=" + streetName + ", pincode=" + pincode + "]";
    }
}

```

Now Define, Another component class **Account** and define Address type property inside as a Dependency property.

```

package com.dilip.account;

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class Account {

    private String name;
    private long accNumber;

    // Field/Property Level
    @Autowired
    private Address addr;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

```

    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
    public Address getAddr() {
        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
}

```

- Create a configuration class, and define Component Scan packages to scan all packages.

```

package com.dilip.account;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.dilip.*")
public class BeansConfiguration {

}

```

- Now Define, Main class and try to get Account Bean object and check really Address Bean Object Injected or Not.

```

package com.dilip.account;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringAutowiringDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        context.register(BeansConfiguration.class);
        context.refresh();

        // UserDetails Component
        Account account = (Account) context.getBean(Account.class);
        //Getting Injected Object of Address
        Address address = account.getAddr();
        address.setPincode(500072);
    }
}

```

```
        System.out.println(address);
    }
}
```

Output: Address [streetName=null, pincode=500072]

So, Dependency Object **Address** injected in **Account** Bean Object implicitly, with **@Autowired** on property level.

Autowiring with Multiple Bean ID Configurations with Single Bean/Component Class:

Let's Create Bean class: Below class Bean Id is : **home**

```
package com.hello.spring.boot.employees;

import org.springframework.stereotype.Component;

@Component("home")
public class Addresss {

    private String streetName;
    private int pincode;

    public String getStreetName() {
        return streetName;
    }
    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
    public void printAddressDetails() {
        System.out.println("Street Name is : " + this.streetName);
        System.out.println("Pincode is : " + this.pincode);
    }
}
```

➤ For above Address class create a Bean configuration in Side Configuration class.

```

package com.hello.spring.boot.employees;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.hello.spring.boot.*")
public class BeansConfig {

    @Bean("hyd")
    Addresss createAddress() {
        Addresss a = new Addresss();
        a.setPincode(500067);
        a.setStreetName("Gachibowli");
        return a;
    }
}

```

➤ **Now Autowire Address in Employee class.**

```

package com.hello.spring.boot.employees;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public long getMobile() {
        return mobile;
    }

    public void setMobile(long mobile) {

```



```

        this.mobile = mobile;
    }
    public Addresss getAdd() {
        return add;
    }
    public void setAdd(Addresss add) {
        this.add = add;
    }
}

```

- Now Test which Address Object Injected by Container i.e. either home or hyd bean object.

```

package com.hello.spring.boot.employees;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AutowiringTestMainApp {

    public static void main(String[] ar) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        context.register(BeansConfig.class);
        context.refresh();
        Employee emp = (Employee) context.getBean("emp");
        Addresss empAdd = emp.getAdd();
        System.out.println(empAdd);
    }
}

```

We got an exception now as follows,

```

Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'emp': Unsatisfied
dependency expressed through field 'add': No qualifying bean
of type 'com.hello.spring.boot.employees.Addresss' available:
expected single matching bean but found 2: home, hyd

```

i.e. Spring Container unable to inject Address Bean Object into Employee Object because of Ambiguity/Confusion like in between **home** or **hyd** bean Objects of Address type.

To resolve this Spring provided one more annotation called as **@Qualifier**

@Qualifier:

By using the **@Qualifier** annotation, we can eliminate the issue of which bean needs to be injected. There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

We need to take into consideration that the qualifier name to be used is the one declared in the **@Component** or **@Bean** annotation.

Now add **@Qualifier** on **Address** field, inside **Employee** class.

```
package com.hello.spring.boot.employees;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    @Qualifier("hyd")
    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
    public Addresss getAdd() {
        return add;
    }
    public void setAdd(Addresss add) {
        this.add = add;
    }
}
```

```
}  
}
```

- Now Test which Address Bean Object with bean Id “**hyd**” Injected by Container.

```
package com.hello.spring.boot.employees;  
  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class AutowiringTestMainApp {  
    public static void main(String[] ar) {  
  
        AnnotationConfigApplicationContext context =  
            new AnnotationConfigApplicationContext();  
        context.register(BeansConfig.class);  
        context.refresh();  
        Employee employee = (Employee) context.getBean("emp");  
        Address empAdd = employee.getAdd();  
        System.out.println(empAdd.getPincode());  
        System.out.println(empAdd.getStreetName());  
    }  
}
```

Output: 500067
Gachibowli

i.e. **Address** Bean Object Injected with Bean Id called as **hyd** into **Employee** Bean Object.

@Primary:

There's another annotation called **@Primary** that we can use to decide which bean to inject when ambiguity is present regarding dependency injection. This annotation **defines a preference when multiple beans of the same type are present**. The bean associated with the **@Primary** annotation will be used unless otherwise indicated.

Now add One more **@Bean** config for **Address** class inside Configuration class.

```
package com.hello.spring.boot.employees;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
  
@Configuration  
@ComponentScan("com.hello.spring.boot.*")  
public class BeansConfig {  
    @Bean("hyd")
```

```

Addresss createAddress() {
    Addresss a = new Addresss();
    a.setPincode(500067);
    a.setStreetName("Gachibowli");
    return a;
}
@Bean("banglore")
@Primary
Addresss bangloreAddress() {
    Addresss a = new Addresss();
    a.setPincode(560043);
    a.setStreetName("Banglore");
    return a;
}
}

```

In above, we made **@Bean("banglore")** as Primary i.e. by Default bean object with ID **"banglore"** should be injected out of multiple Bean definitions of Address class when **@Qualifier** is not defined with **@Autowired**.

```

package com.hello.spring.boot.employees;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("emp")
public class Employee {

    private String lastName;
    private long mobile;

    //No @Qualifier Defined
    @Autowired
    private Addresss add;

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public long getMobile() {
        return mobile;
    }
    public void setMobile(long mobile) {
        this.mobile = mobile;
    }
}

```

```

    public Addresss getAdd() {
        return add;
    }
    public void setAdd(Addresss add) {
        this.add = add;
    }
}

```

Output:

560043
Banglore

I.e. **Address** Bean Object with "**banglore**" injected in **Employee** object level.

NOTE: if both the **@Qualifier** and **@Primary** annotations are present, then the **@Qualifier** annotation will have precedence/priority. Basically, **@Primary** defines a default, while **@Qualifier** is very specific to Bean ID.

Autowiring With Interface and Implemented Classes:

In Java, Interface reference can hold Implemented class Object. With this rule, We can Autowire Interface references to inject implemented component classes.

➤ Now Define an Interface: **Animal**

```

package com.dilip.auto.wiring;

public interface Animal {
    void printNameOfAnimal();
}

```

➤ Now Define A class from interface : **Tiger**

```

package com.dilip.auto.wiring;

import org.springframework.stereotype.Component;

@Component
public class Tiger implements Animal {
    @Override
    public void printNameOfAnimal() {
        System.out.println("I am a Tiger ");
    }
}

```

- **Now Define a Configuration class for component scanning.**

```
package com.dilip.auto.wiring;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.dilip.*")
public class BeansConfig {

}
```

- Now Autowire **Animal** type property in any other Component class i.e. Dependency of **Animal** Interface implemented class Object **Tiger** should be injected.

```
package com.dilip.auto.wiring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class AnimalManagement {

    @Autowired
    //Interface Type Property
    Animal animal;

}
```

- Now Test, Animal type property injected with what type of Object.

```
package com.dilip.auto.wiring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

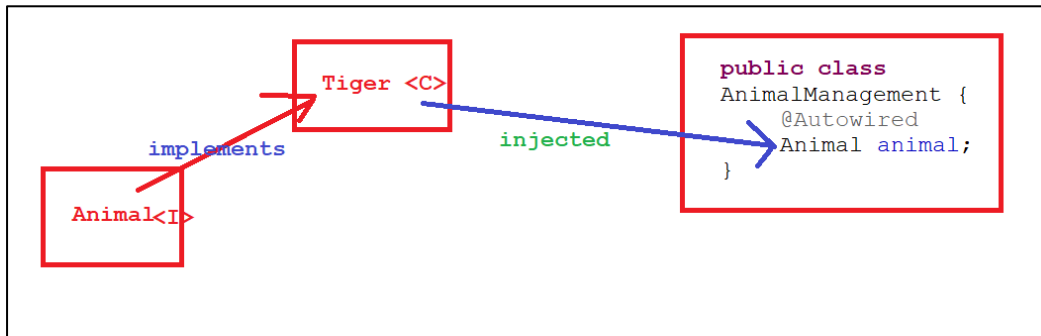
public class AutoWringDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfig.class);
        context.refresh();

        // Getting AnimalManagement Bean Object
        AnimalManagement animalMgmt = context.getBean(AnimalManagement.class);
        animalMgmt.animal.printNameOfAnimal();
    }
}
```

```
}
```

Output: I am a Tiger

So, implicitly Spring Container Injected one and only implanted class Tiger of Animal Interface inside Animal Type reference property of AnimalManagement Object.



If we have multiple Implemented classes for same Interface i.e. Animal interface, How Spring Container deciding which implanted Bean object should Injected?

- Define one more Implementation class of Animal Interface : **Lion**

```
package com.dilip.auto.wiring;

import org.springframework.stereotype.Component;

@Component("lion")
public class Lion implements Animal {
    @Override
    public void printNameOfAnimal() {
        System.out.println("I am a Lion ");
    }
}
```

- Now Test, **Animal** type property injected with what type of Object either **Tiger** or **Lion**.

```
package com.dilip.auto.wiring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

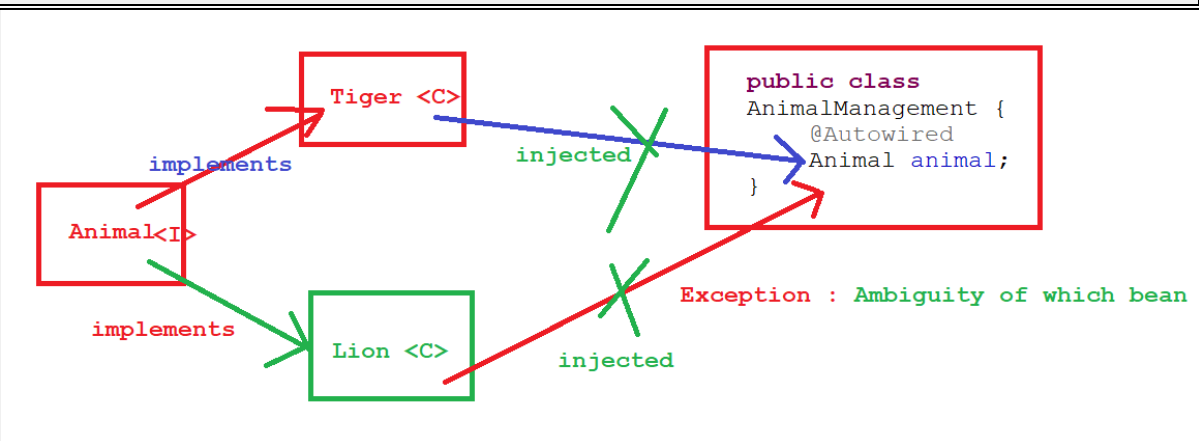
public class AutoWringDemo {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfig.class);
        context.refresh();
    }
}
```

```
// Getting AnimalManagement Bean Object
AnimalManagement animalMgmt = context.getBean(AnimalManagement.class);
animalMgmt.animal.printNameOfAnimal();
}
}
```

We got an Exception as,

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyExcept
ion: Error creating bean with name 'animalManagement':
Unsatisfied dependency expressed through field 'animal': No
qualifying bean of type 'com.dilip.auto.wiring.Animal'
available: expected single matching bean but found 2:
lion,tiger
```



So to avoid this ambiguity again between multiple implementation of single interface, again we can use **@Qualifier** with Bean Id or Component Id.

```
package com.dilip.auto.wiring;

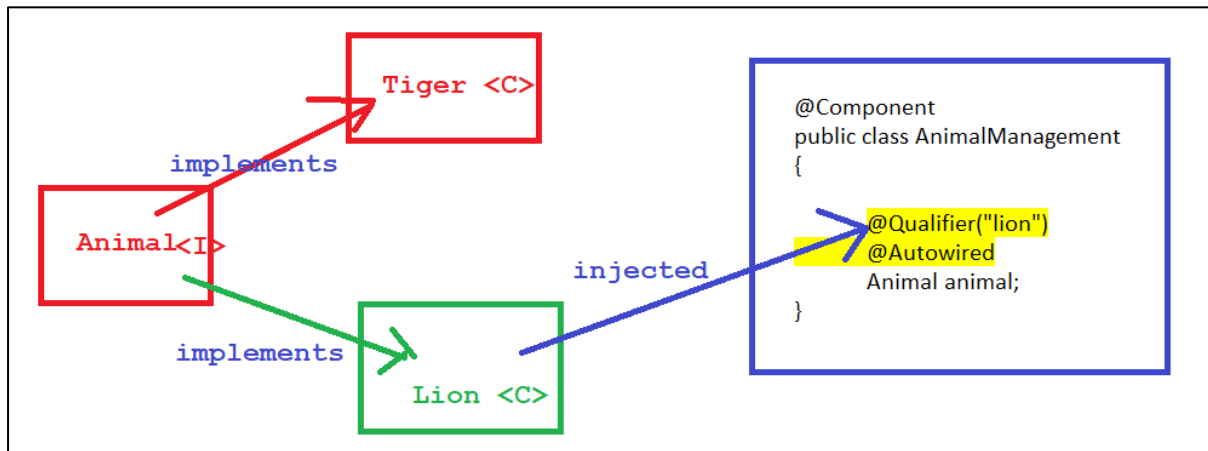
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class AnimalManagement {
    @Qualifier("lion")
    @Autowired
    Animal animal;
}
```

Now it will inject only **Lion** Object inside **AnimalManagement** Object as per **@Qualifier** annotation value out of **lion** and **tiger** bean objects.

Run again now **AutoWringDemo.java**

Output : **I am a Lion.**



Can we inject Default implemented class Object out of multiple implementation classes into Animal reference if not provided any Qualifier value?

Yes, we can inject default Implementation bean Object of Interface. We should mark one class as **@Primary**. Now I marked Tiger class as **@Primary** and removed **@Qualifier** from **AnimalManagement**.

```
package com.dilip.auto.wiring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class AnimalManagement {
    @Autowired
    Animal animal;
}
```

Run again now **AutoWringDemo.java**

Output : **I am a Tiger**

Types of Dependency Injection with Annotations :

The process where objects use their dependent objects without a need to define or create them is called dependency injection. It's one of the core functionalities of the Spring framework.

We can inject dependent objects in three ways, using:

Spring Framework supporting 3 types if Dependency Injection .

1. Filed/Property level Injection (Only supported Via Annotations)
2. Setter Injection
3. Constructor Injection

Filed Injection:

As the name says, the dependency is injected directly in the field, with no constructor or setter needed. This is done by annotating the class member with the **@Autowired** annotation. If we define **@Autowired** on property/field name level, then Spring Injects Dependency Object directly into filed.

Requirement : Address is Dependency of Employee class.

Address.java : Create as Component class

```
package com.dilip.spring;

import org.springframework.stereotype.Component;

@Component
public class Address {

    private String city;
    private int pincode;

    public Address() {
        System.out.println("Address Object Created.");
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getPincode() {
        return pincode;
    }
    public void setPincode(int pincode) {
        this.pincode = pincode;
    }
}
```

➤ **Employee.java** : Component class with Dependency Injection.

```
package com.dilip.spring;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Employee {

    private String empName;
    private double salary;

    //Field Injection
    @Autowired
    private Address address;

    public Employee(Address address ) {
        System.out.println("Employee Object Created");
    }
    public String getEmpName() {
        return empName;
    }
    public void setEmpName(String empName) {
        this.empName = empName;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        System.out.println("This is etter method of Emp of Address");
        this.address = address;
    }
}

```

We are Defined **@Autowired** on **Address** type field in side **Employee** class, So Spring IOC will inject **Address** Bean Object inside **Employee** Bean Object via field directly.

Testing DI:

```

package com.dilip.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class DiMainAppDemo {

```

```

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        context.scan("com.*");
        context.refresh();
        Employee emp = context.getBean(Employee.class);
        System.out.println(emp);
        System.out.println(emp.getAddress());
    }
}

```

Output: Address Object Created.
Employee Object Created
Address Object Created.
com.dilip.spring.Employee@791f145a
com.dilip.spring.Address@38cee291

Setter Injection Overview:

Setter injection uses the setter method to inject dependency on any Spring-managed bean. Well, the Spring IOC container uses a setter method to inject dependency on any Spring-managed bean. We have to annotate the setter method with the **@Autowired** annotation.

Let's create an interface and Impl. Classes in our project.

Interface : **MessageService.java**

```

package com.dilip.setter.injection;

public interface MessageService {
    void sendMessage(String message);
}

```

Impl. Class : **EmailService.java**

```

package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("emailService")
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}

```

We have annotated **EmailService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

Impl. Class : **SMSService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("smsService")
public class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **SMSService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

MessageSender.java: In setter injection, Spring will find the **@Autowired** annotation and call the setter method to inject the dependency.

```
package com.dilip.setter.injection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class MessageSender {

    private MessageService messageService;

    //At setter method level.
    @Autowired
    public void setMessageService(@Qualifier("emailService") MessageService messageService) {
        this.messageService = messageService;
        System.out.println("setter based dependency injection");
    }

    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
    }
}
```

@Qualifier annotation is used in conjunction with **@Autowired** to avoid confusion when we have two or more beans configured for the same type.

➤ **Now create a Test class to validate, dependency injection with setter Injection.**

```
package com.dilip.setter.injection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Client {
    public static void main(String[] args) {

        String message = "Hi, good morning have a nice day!.";
        ApplicationContext context = new AnnotationConfigApplicationContext();
        context.scan("com.dilip.*");

        MessageSender messageSender = context.getBean(MessageSender.class);
        messageSender.sendMessage(message);
    }
}
```

Output:

```
setter based dependency injection
Hi, good morning have a nice day!.
```

Injecting Multiple Dependencies using Setter Injection:

Let's see how to inject multiple dependencies using Setter injection. To inject multiple dependencies, we have to create multiple fields and their respective setter methods. In the below example, the **MessageSender** class has multiple setter methods to inject multiple dependencies using setter injection:

```
package com.dilip.setter.injection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class MessageSender {

    private MessageService messageService;
    private MessageService smsService;
```

@Autowired

```
public void setMessageService(@Qualifier("emailService") MessageService messageService) {  
  
    this.messageService = messageService;  
    System.out.println("setter based dependency injection");  
}
```

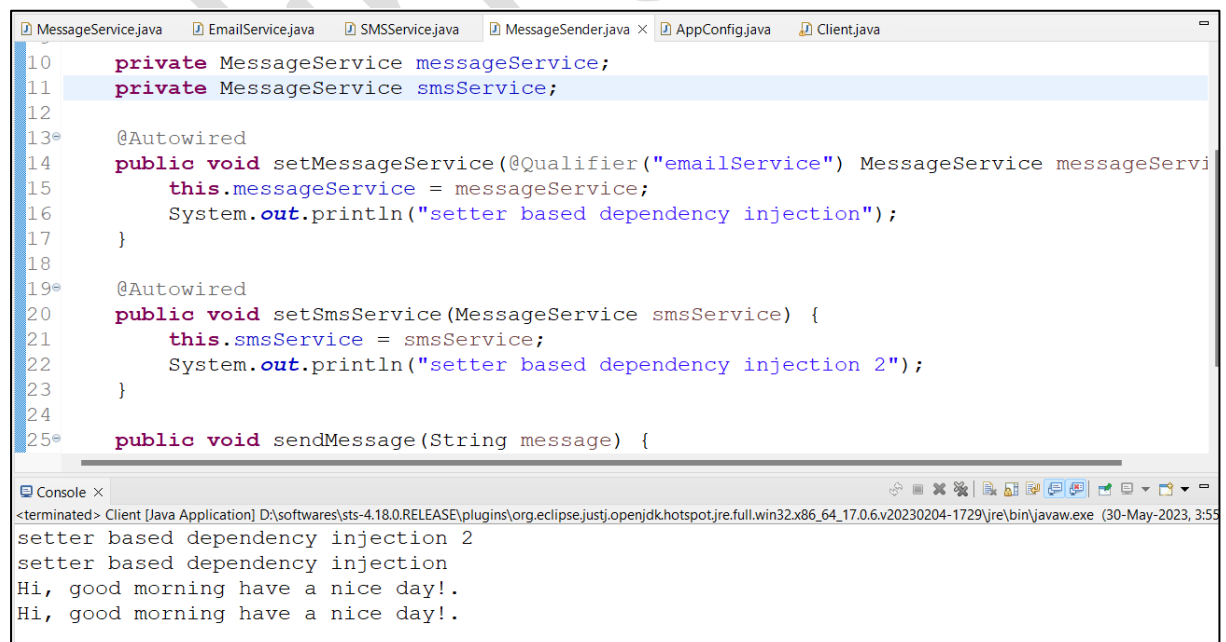
@Autowired

```
public void setSmsService(MessageService smsService) {  
    this.smsService = smsService;  
    System.out.println("setter based dependency injection 2");  
}  
  
public void sendMessage(String message) {  
    this.messageService.sendMessage(message);  
    this.smsService.sendMessage(message);  
}  
}
```

➤ **Now Run Client.java**, One more time to see both Bean Objects injected or not.

Output:

setter based dependency injection 2
setter based dependency injection
Hi, good morning have a nice day!.
Hi, good morning have a nice day!.



The screenshot shows an IDE with several tabs: MessageService.java, EmailService.java, SMSService.java, MessageSender.java, AppConfig.java, and Client.java. The Client.java file is open, showing the following code:

```
10 private MessageService messageService;  
11 private MessageService smsService;  
12  
13 @Autowired  
14 public void setMessageService(@Qualifier("emailService") MessageService messageService) {  
15     this.messageService = messageService;  
16     System.out.println("setter based dependency injection");  
17 }  
18  
19 @Autowired  
20 public void setSmsService(MessageService smsService) {  
21     this.smsService = smsService;  
22     System.out.println("setter based dependency injection 2");  
23 }  
24  
25 public void sendMessage(String message) {
```

The console output at the bottom shows the following lines:

```
<terminated> Client [Java Application] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe (30-May-2023, 3:55)  
setter based dependency injection 2  
setter based dependency injection  
Hi, good morning have a nice day!.  
Hi, good morning have a nice day!.
```

Constructor Injection:

Constructor injection uses the constructor to inject dependency on any Spring-managed bean. Well, the Spring IOC container uses a constructor to inject dependency on any Spring-managed bean. In order to demonstrate the usage of constructor injection, let's create a few interfaces and classes.

➤ **MessageService.java**

```
package com.dilip.setter.injection;

public interface MessageService {
    void sendMessage(String message);
}
```

➤ **EmailService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **EmailService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

➤ **SMSService.java**

```
package com.dilip.setter.injection;

import org.springframework.stereotype.Component;

@Component("smsService")
public class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println(message);
    }
}
```

We have annotated **SMSService** class with **@Component** annotation so the Spring container automatically creates a Spring bean and manages its life cycle.

➤ **MessageSender.java**

```
package com.dilip.setter.injection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class MessageSender {

    private MessageService messageService;

    //Constructor level Auto wiring
    @Autowired
    public MessageSender(@Qualifier("emailService") MessageService messageService) {

        this.messageService = messageService;
        System.out.println("constructor based dependency injection");
    }
    public void sendMessage(String message) {
        this.messageService.sendMessage(message);
    }
}
```

➤ **Now create a Configuration class: AppConfig.java**

```
package com.dilip.setter.injection;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.dilip.*")
public class AppConfig {

}
```

➤ **Now create a Test class to validate, dependency injection with setter Injection.**

```
package com.dilip.setter.injection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Client {
```

```

public static void main(String[] args) {

    String message = "Hi, good morning have a nice day!.";

    ApplicationContext applicationContext =
        new AnnotationConfigApplicationContext(AppConfig.class);
    MessageSender messageSender =
        applicationContext.getBean(MessageSender.class);
    messageSender.sendMessage(message);
}
}

```

Output:

constructor based dependency injection
Hi, good morning have a nice day!.

How to Declare Types of Autowiring with Annotations?

When we discussed of autowiring with beans XML configurations, Spring Provided 4 types autowiring configuration values for **autowire** attribute of **bean** tag.

1. no
2. byName
3. byType
4. constructor

But with annotation Bean configurations, we are not using these values directly because we are achieving same functionality with **@Autowired** and **@Qualifier** annotations directly or indirectly.

Let's compare functionalities with annotations and XML attribute values.

no: If we are not defined **@Autowired** on field/setter/constructor level, then Spring not injecting Dependency Object in side composite Object.

byType: If we define only **@Autowired** on field/setter/constructor level then, Spring injecting Dependency Object in side composite Object specific to Datatype of Bean. This works when we have only one Bean Configuration of Dependent Object.

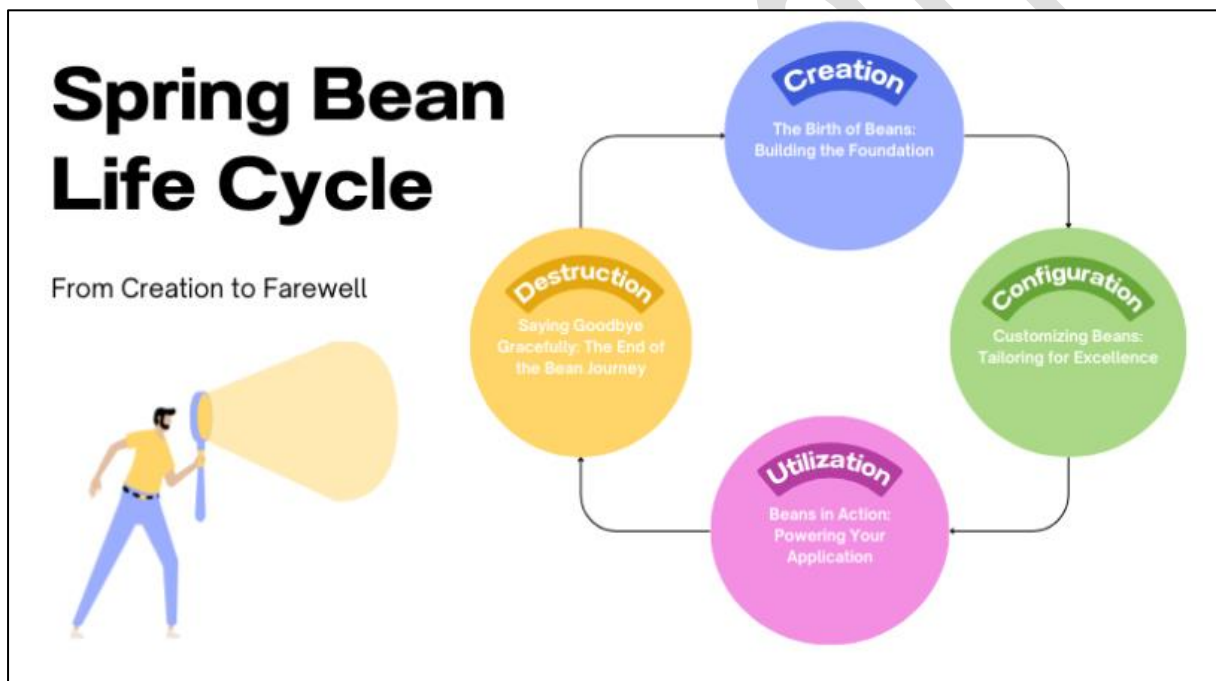
byName: If we define **@Autowired** on field/setter/constructor level along with **@Qualifier** then, Spring injecting Dependency Object in side composite Object specific to Bean ID.

constructor : when we are using **@Autowired** and **@Qualifier** along with constructor, then Spring IOC container will inject Dependency Object via constructor.

So explicitly we no need to define any autowiring type with annotation based Configurations like in XML configuration.

Bean life cycle in Java Spring:

The Spring Bean life cycle is the heartbeat of any Spring application, dictating how beans are created, initialized, and eventually destroyed. The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.



Spring bean is a Java object managed by the Spring IoC container. These objects can be anything, from simple data holders to complex business logic components. The magic lies in Spring's ability to manage the creation, configuration, and lifecycle of these beans.

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 configuration, and then dependencies are injected. After utilization of Bean Object and then 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()** methods.

Benefits of Exploring the Spring Bean Life Cycle:

- **Resource Management:** As you traverse the life cycle stages of Bean Object, you're in control of resources. This translates to efficient memory utilization and prevents resource leaks, ensuring your application runs like a well configured machine.
- **Customization:** By walking through the life cycle stages, you can inject custom logic at strategic points. This customization allows your beans to adapt to specific requirements, setting the stage for a flexible and responsive application.
- **Dependency Injection:** Understanding the stages of bean initialization also resolves the magic of dependency injection. You'll learn how beans communicate, collaborate, and share information, building a cohesive application architecture.
- **Debugging:** With a firm grasp of the life cycle, troubleshooting becomes very easy. By tracing a bean's journey through each stage, you can pinpoint issues and enhance the overall stability of your application.

Defining Bean Life Cycle Methods:

Spring allows us to attach custom actions to bean creation and destruction. We can do it by implementing the **InitializingBean** and **DisposableBean** interfaces.

InitializingBean:

InitializingBean is an interface in the Spring Framework that allows a bean to perform initialization tasks after its properties have been set. It defines a single method, **afterPropertiesSet()**, which a bean class must implement to carry out any initialization logic.

When the Spring container initializes the Bean instance, it will first set any properties configured, and then it will call the **afterPropertiesSet()** method automatically. This allows you to perform any custom initialization tasks within that method.

DisposableBean:

DisposableBean is another interface in the Spring Framework that complements the **InitializingBean**. While **InitializingBean** is used for performing initialization tasks, **DisposableBean** is used for performing cleanup or disposal tasks when a bean is being removed from the Spring container.

The **DisposableBean** interface defines a single method, **destroy()**, which a bean class must implement to carry out any cleanup logic.

When the Spring container is shutting down or removing the bean, it will call the **destroy()** method automatically, allowing you to perform any necessary cleanup tasks.

Defining Bean Life Cycle Methods with Beans:

- Create a Bean class by implementing both **InitializingBean** and **DisposableBean**.

```
package com.dilip;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;

@Component
public class Customer implements InitializingBean, DisposableBean {

    private int id;
    private String name;

    public Customer() {
        System.out.println("Customer Object is Created");
    }

    // This will be executed once instance created
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("This is Init logic from afterPropertiesSet()");
        System.out.println("Initialization logic goes here");
    }

    // This will be executed before container instance closing
    @Override
    public void destroy() throws Exception {
        System.out.println("This is destroying logic from destroy()");
        System.out.println("Cleanup logic goes here");
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        System.out.println("Setter for injecting ID value");
        this.id = id;
    }

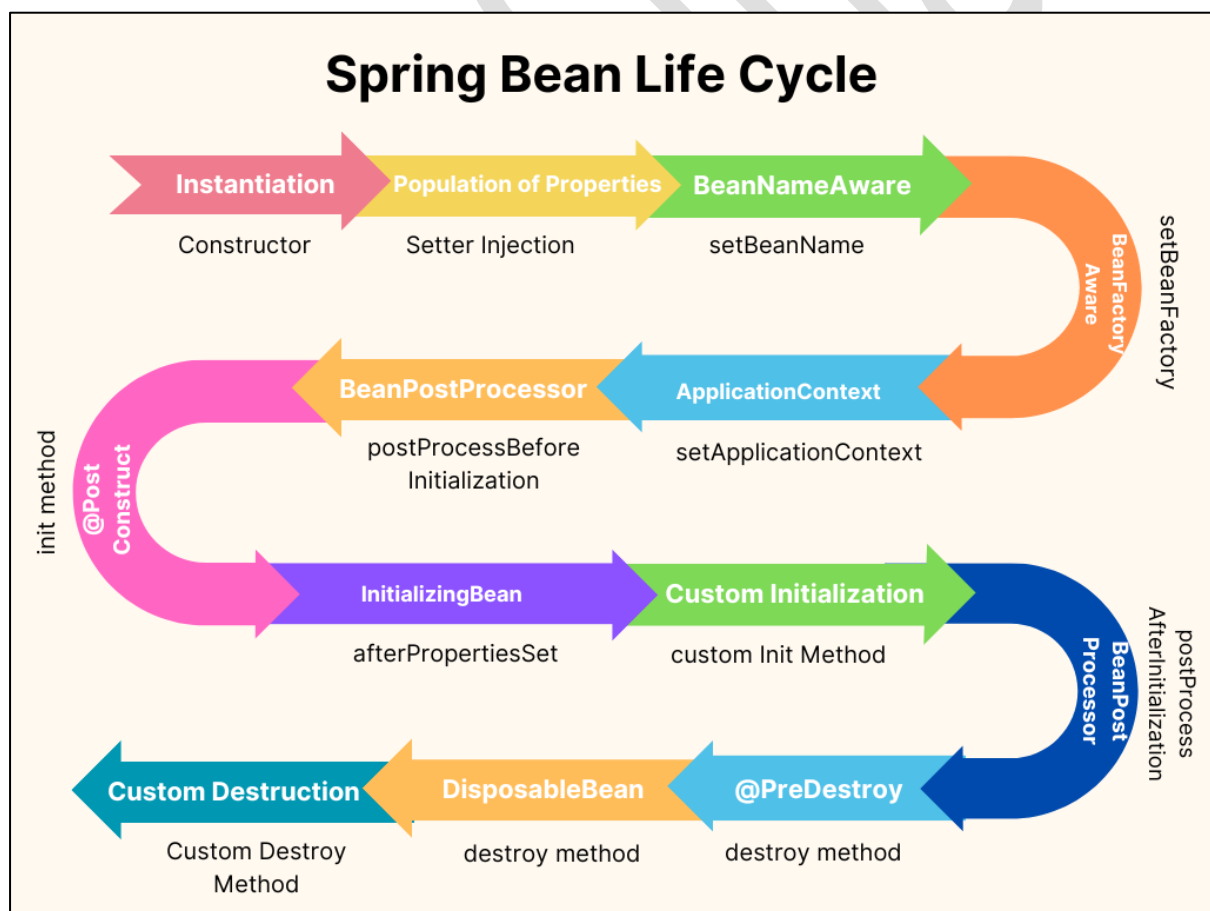
    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("Setter for injecting name value");
        this.name = name;
    }
}
```

Spring Bean Object Life Cycle followed below steps in an order.

- Now get this Bean Object from IOC container.
- Once we created or initialized Spring Container, Container starts the process of Instantiating Bean Objects.
- Bean Object will be created/instantiated
- After Bean Object creation , Properties Values will be injected if any available.
- All Dependencies will be identified and injected
- Now Bean Initialization method i.e. **afterPropertiesSet()** method logic will be executed by IOC container one time i.e. this method will be executed once anew Object is created always by container.
- Now Bean Object is ready with all configuration values of properties and initialization values.
- We will always get current object always when we get it from container always.
- After utilization of Bean Object, when the Spring container is shutting down or removing the bean, it will call the **destroy()** method automatically for every Bean Object, allowing you to perform any necessary cleanup tasks written as part of the method.
- After executing all bean Objects **destroy()** methods then finally container got closed.

The following image shows the process flow of the Bean Object life cycle.



Spring Bean Life Cycle Flow

- Now create Spring IOC container instance and try to get Bean Object and then close the container Instance.
- Creating **Beans Configuration** class.

```
package com.dilip;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {

}
```

- Now Pass above Configuration class to container.

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {

        // Created the Container
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        // Providing Bean Classes information to Container
        context.register(BeansConfiguration.class);

        // Create/Instantiate Bean Objects
        context.refresh();

        //Get the Bean Object from Container and Utilize it
        Customer customer = context.getBean(Customer.class);
        System.out.println(customer);

        // Closing the Container
        context.close();
    }
}
```

Output:

```
Problems @ Javadoc Declaration Console ×
<terminated> SpringBeanLifeCycleDemo [Java Application] D:\softwares\eclipse\plugins\org.eclipse.justj.o
Customer Object is Created
This is Init logic from afterPropertiesSet()
Initialization logic goes here
com.dilip.Customer@1a942c18
This is destroying logic from destroy()
Cleanup logic goes here
```

- Same Process will follow by container internally for every Bean Object of class.
- Adding a Bean Method inside Configuration class for another Customer Object and then we will see same process followed for new Bean Object as usually.

BeansConfiguration.java

```
package com.dilip;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {

    @Bean(name="customer2")
    Customer getCustomer() {
        return new Customer();
    }

}
```

- Now Execute container creation and closing Lofigc again and observe initialization and destroy methods executed 2 times for 2 Customer Bean Objects creation.

```
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {
        // Created the Container
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();

        // Providing Bean Classes information to Container
        context.register(BeansConfiguration.class);
    }
}
```



```

        // Create/Instantiate Bean Objects
        context.refresh();
        // Closing the Container
        context.close();
    }
}

```

Output : For Every bean Object, executed both actions of initialization and destroy.

```

<terminated> SpringBeanLifecycleDemo [Java Application] D:\softwares\ eclipse\plugins\org.eclipse.justj.open
Customer Object is Created ✓
This is Init logic from afterPropertiesSet() ✓
Initialization logic goes here
Customer Object is Created ✓
This is Init logic from afterPropertiesSet() ✓
Initialization logic goes here
This is destroying logic from destroy() ✓
Cleanup logic goes here
This is destroying logic from destroy() ✓
Cleanup logic goes here

```

This is how we can define lifecycle methods explicitly to provide instantiation logic and destruction logic for a bean Object.

Note: Same above approach of writing Bean class with **InitializingBean** and **DisposableBean**, can be followed in Spring Beans XML configuration for a Bean class and Objects.

Question: **Do we have any other ways to define life cycle methods apart from InitializingBean and DisposableBean?**

Yes, we have second possibility, the **@PostConstruct** and **@PreDestroy** annotations from Java EE.

Note: Both **@PostConstruct** and **@PreDestroy** annotations are part of Java EE. Since Java EE was deprecated in Java 9, and removed in Java 11, we have to add an additional dependency in pom.xml to use these annotations.

```

<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>

```

Define these annotations on custom methods of Bean instantiation and destroying logic without using any Predefined Interfaces from Spring FW.

@PostConstruct:

Spring calls the methods annotated with **@PostConstruct** only once, just after the initialization of bean properties i.e. this is a replacement of **InitializingBean** and its associated abstract method implementation.

@PreDestroy:

Spring calls the methods annotated with **@PreDestroy** runs only once, just before Spring removes our bean from the application context.

Note: **@PostConstruct** and **@PreDestroy** annotated methods can have any access level, but can't be static.

Example: Bean Class: Customer.java

```
package com.dilip;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.stereotype.Component;

@Component
public class Customer {

    private int id;
    private String name;

    public Customer() {
        System.out.println("Customer Object is Created");
    }

    @PostConstruct
    public void init() {
        System.out.println("This is Init logic from init()");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("This is destroying logic from destroy()");
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

```

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

- **Beans Configuration Class: BeansConfiguration.java** : Created another Bean Instance

```

package com.dilip;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan("com.dilip")
@Configuration
public class BeansConfiguration {
    @Bean(name="customer2")
    Customer getCustomer() {
        return new Customer();
    }
}

```

- Now Execute container creation and closing Lofgc again and observe initialization and destroy methods executed 2 times for 2 Customer Bean Objects creation.

```

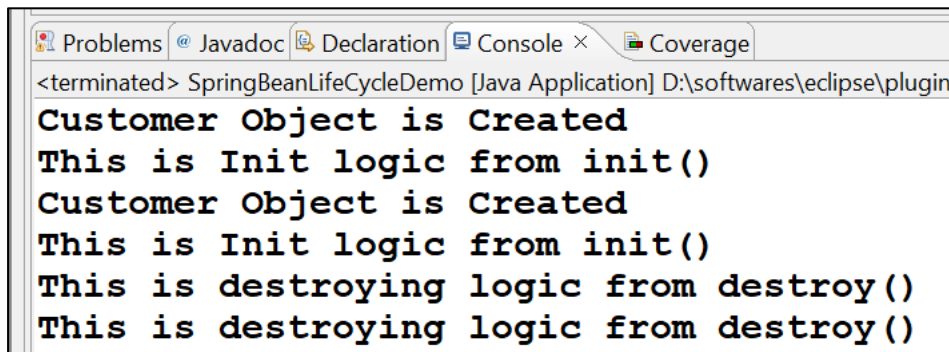
package com.dilip;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringBeanLifeCycleDemo {
    public static void main(String[] args) {
        // Created the Container
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
        context.register(BeansConfiguration.class);
        context.refresh();
        // Closing the Container
        context.close();
    }
}

```

Output : For Every bean Object, executed both actions of initialization and destroy.



```
<terminated> SpringBeanLifeCycleDemo [Java Application] D:\softwares\eclipse\plugin
Customer Object is Created
This is Init logic from init()
Customer Object is Created
This is Init logic from init()
This is destroying logic from destroy()
This is destroying logic from destroy()
```

Question: Can we define custom methods in class for initialization and destruction of a bean object i.e. without using Pre-Defined Interfaces and Annotations ?

Yes, We can Define custom methods with user defined names of methods of both initialization and destroying actions.

Bean class: Student.java

```
package com.dilip;

public class Student {

    private int sid;

    public Student() {
        System.out.println("Student Constructor : Object Created");
    }

    public int getSid() {
        return sid;
    }

    public void setSid(int sid) {
        this.sid = sid;
    }

    // For Initialization
    public void beanInitialization() {
        System.out.println("Bean Initialization Started... ");
    }

    // For Destruction
    public void beanDestruction() {
        System.out.println("Bean Destruction Started..... ");
    }

}
```

In XML Configuration :

- Now inside Beans XML file configuration, define which method is Responsible for Bean life cycle method of initialization and destruction. Spring framework provide 2 pre-defined attributes **init-method** and **destroy-method** as part of **<bean>** tag .
- **Configure custom life cycle methods in Beans.xml by using both attributes:**

```
<?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 id="student" class="com.dilip.Student"
        init-method="beanInitialization"
        destroy-method="beanDestruction">

    </bean>
</beans>
```

- Now Instantiate and close Spring Container and then container will execute life cycle methods as per our configuration of a bean Object.

In Annotation based Configuration :

- Spring Framework provided two attributes **initMethod** and **destroyMethod** as part of **@Bean** annotation. By using these attributes, we will define the method names as followed.

```
@Bean(initMethod = "beanInitialization", destroyMethod = "beanDestruction")
Student getStudent() {
    return new Student();
}
```

Question: Why Understand the Spring Bean Life Cycle?

Understanding the life cycle of Spring beans is like having a backstage pass to the inner workings of your Spring application. A solid grasp of the bean life cycle empowers you to effectively manage resources, configure beans, and ensure proper initialization and cleanup. With this knowledge, you can optimize your application's performance, prevent memory leaks, and implement custom logic at various stages of a bean's existence.