## Spring Framework

## Core Module

## By - Dilip Singh

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 etcc..
* 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 Boot helps you to create stand-alone, production-grade Spring-based applications that you can run. We take an opinionated view of the Spring platform and third-party libraries, so that you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration. It is based on the Spring Framework, favours convention over configuration, and is designed to get you up and running as quickly as possible.

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**](https://en.wikipedia.org/wiki/Aspect-oriented_programming)**:** enables implementing [cross-cutting concerns](https://en.wikipedia.org/wiki/Cross-cutting_concern).
* [**Data access**](https://en.wikipedia.org/wiki/Data_access)**:** working with [relational database management systems](https://en.wikipedia.org/wiki/RDBMS) on the Java platform using [Java Database Connectivity](https://en.wikipedia.org/wiki/Java_Database_Connectivity) (JDBC) and [object-relational mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) tools and with [NoSQL](https://en.wikipedia.org/wiki/NoSQL) databases
* [**Authentication**](https://en.wikipedia.org/wiki/Authentication)**and**[**authorization**](https://en.wikipedia.org/wiki/Authorization)**:** configurable security processes that support a range of standards, protocols, tools and practices via the [Spring Security](https://en.wikipedia.org/wiki/Spring_Security) sub-project.
* [**Model–View–Controller**](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller): an [HTTP](https://en.wikipedia.org/wiki/HTTP)- and [servlet](https://en.wikipedia.org/wiki/Java_Servlet_API)-based framework providing hooks for web applications and [RESTful](https://en.wikipedia.org/wiki/REST) (representational state transfer) Web services.
* [**Testing**](https://en.wikipedia.org/wiki/Software_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.

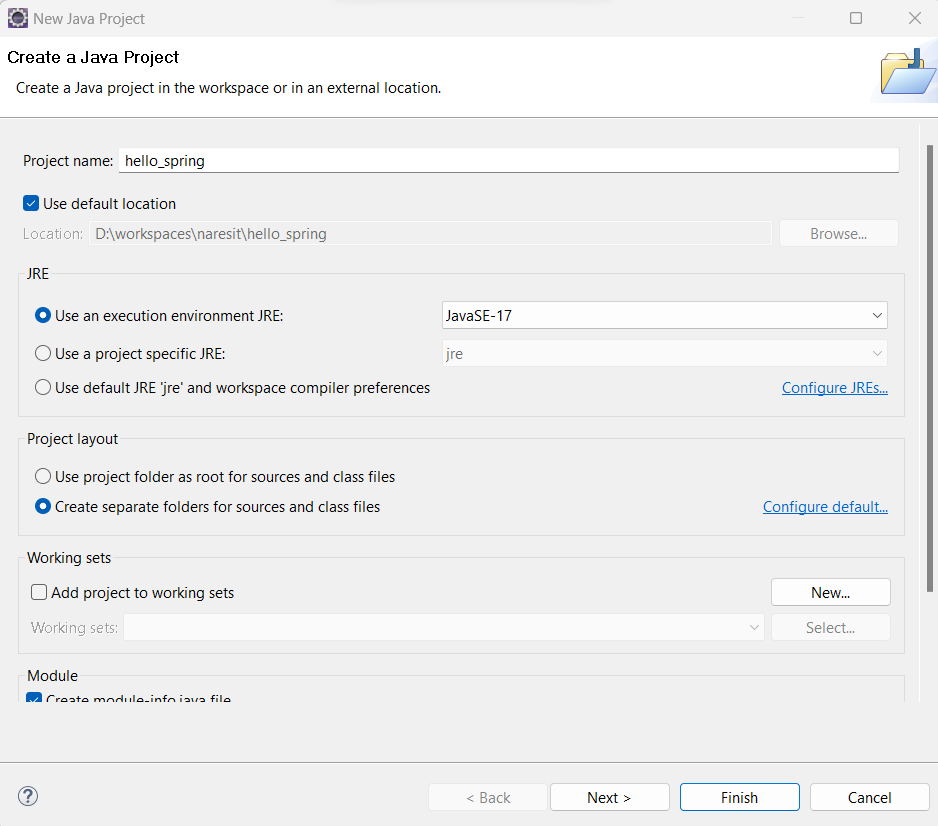
In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations. Here's one way to manually instantiate a container:

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

**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:**

1. **Open Eclipse and create new Java Project and finish.**

****

1. **Now Download Spring Framework Libraries from below link.**

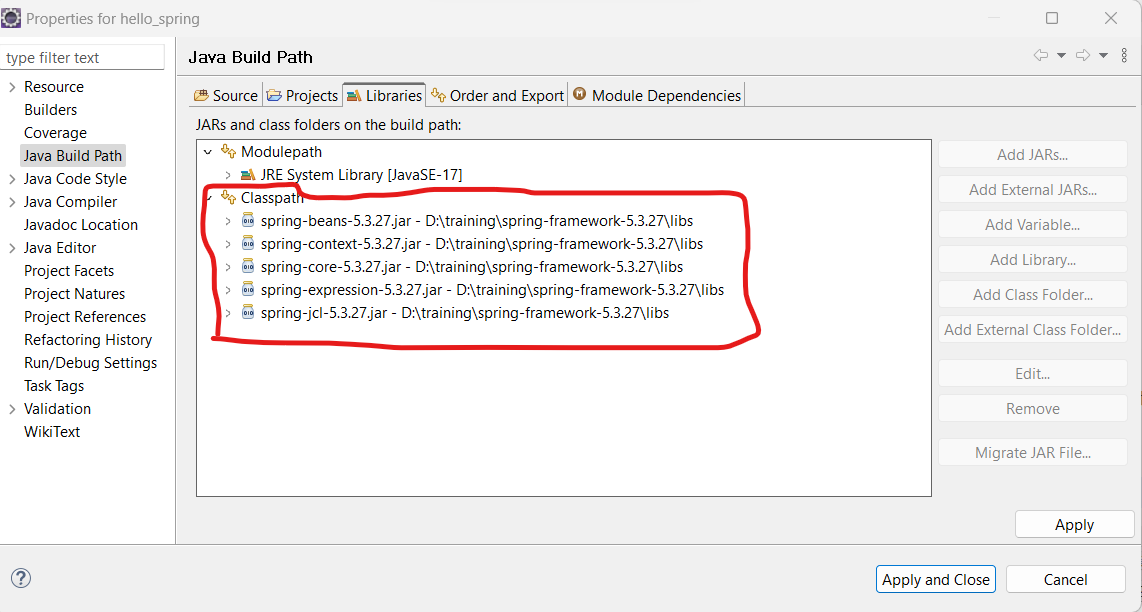
[**https://repo.spring.io/ui/native/release/org/springframework/spring/5.3.27/spring-5.3.27-dist.zip**](https://repo.spring.io/ui/native/release/org/springframework/spring/5.3.27/spring-5.3.27-dist.zip)

**or same copy uploaded in Google Drive.**

[**https://drive.google.com/file/d/1FnbtP3yqjTN5arlEGeoUHCrIJcdcBgM7/view?usp=drive\_link**](https://drive.google.com/file/d/1FnbtP3yqjTN5arlEGeoUHCrIJcdcBgM7/view?usp=drive_link)

After Download completes, Please extract .zip file.

1. Now Please set build path to project with Spring jar files from lib folder, which are shown in image.



1. **Now Create a j ava 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) {

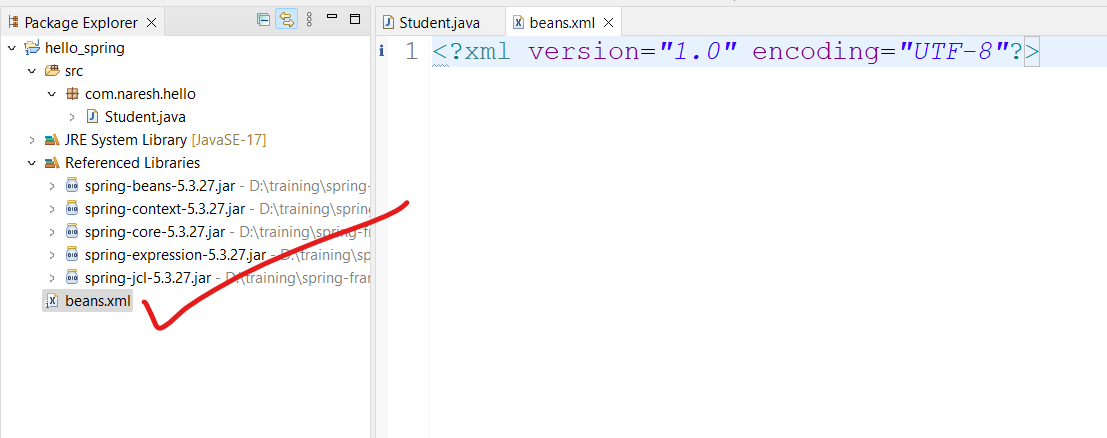
**this**.studnetName = studnetName;

}

}

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

Ex: **beans.xml**

****

1. Now open beans.xml file 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 this content from here asl well.

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

1. 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.

1. 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");

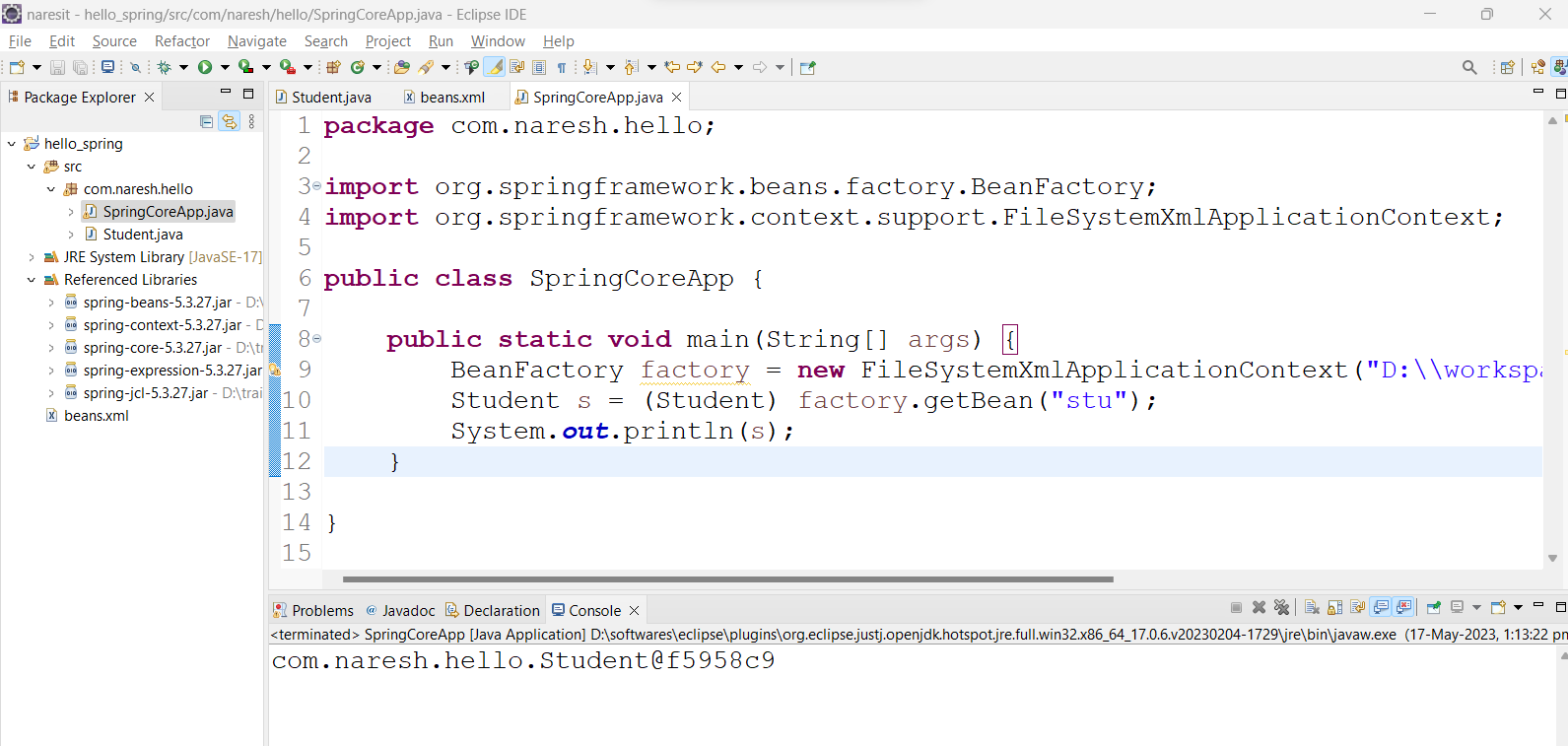
Student s = (Student) factory.getBean("stu");

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

}

}

1. **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="

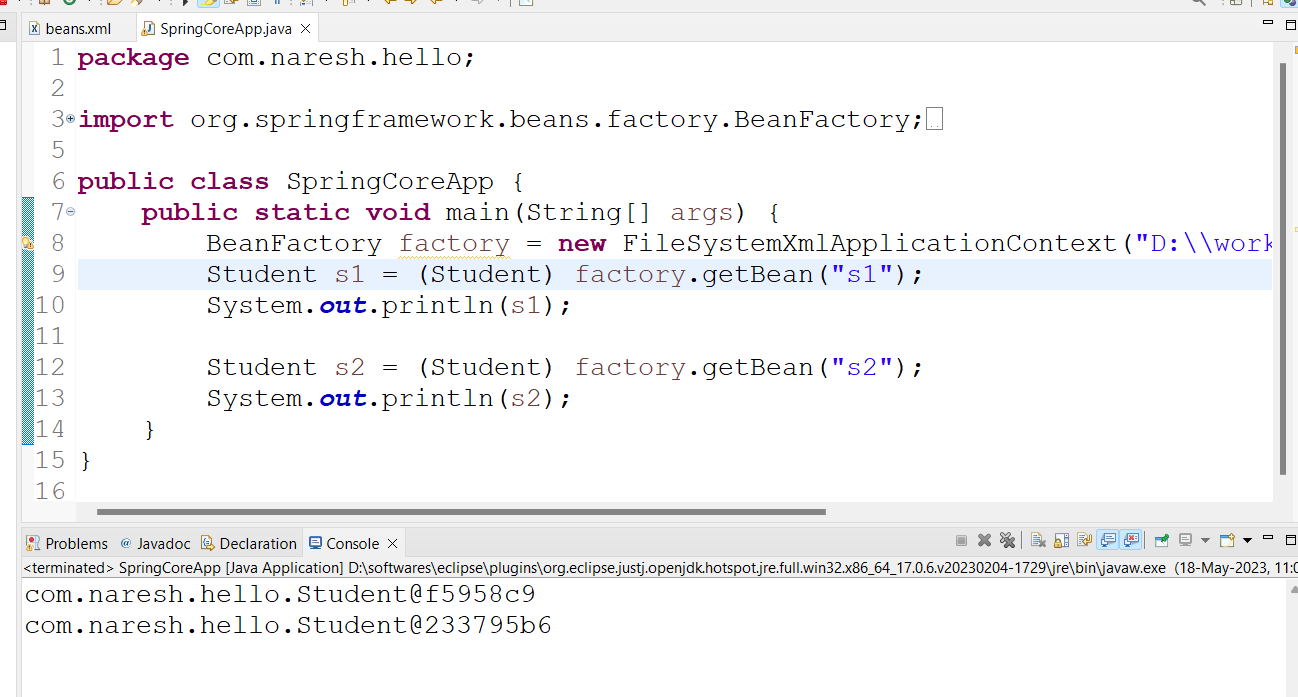
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<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.**



So we can create multiple Bean Objects for a class.

**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 provides 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. |
| BeanFactory interface provides basic features only thus requires less memory. For standalone applications where the basic features are enough and when memory consumption is critical, we can use BeanFactory. | ApplicationContext provides all the basic features and advanced features, including several that are geared towards enterprise applications thus requires more memory. |

**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;

}

}

1. 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

1. 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.getStudentId());

System.***out***.println(s1.getStudentName());

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

s1.printStudentDeatils();

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

}

}

**Output:**

100

Dilip Singh

ABC College

This is Student class

76.0

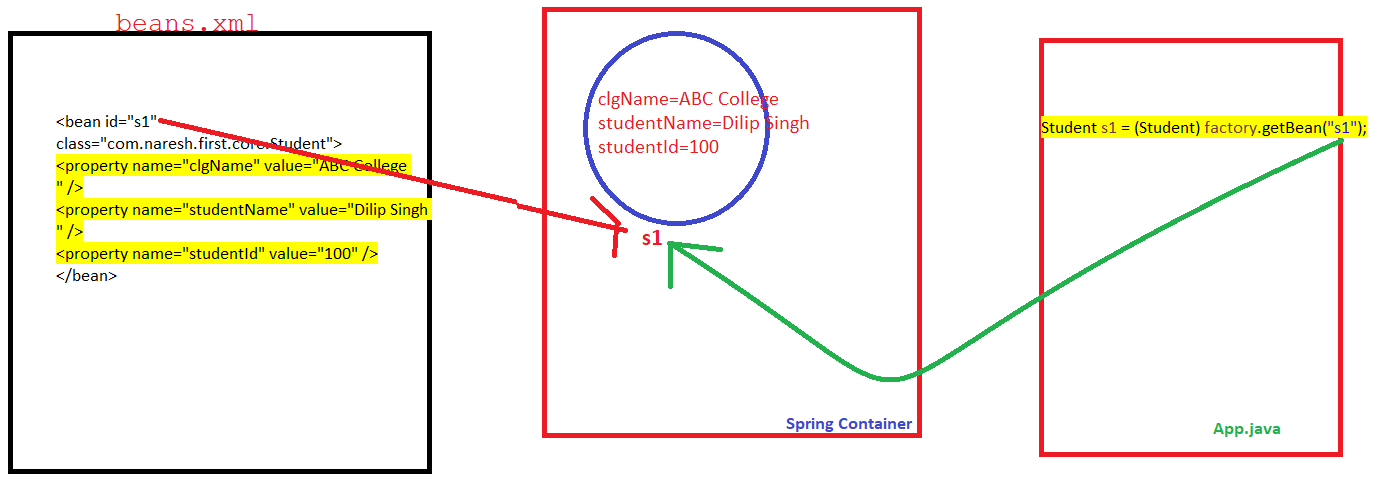
**Internal Workflow/Execution of Above Program.**

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");

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

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



1. 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}

**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.

* **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>+91-88888888</value>

<value>+232388888888</value>

</set>

</constructor-arg>

<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, +91-88888888, +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>+91-88888888</value>

<value>+232388888888</value>

</set>

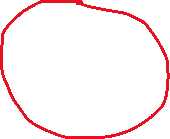
</constructor-arg>

</bean>

</beans>

*AccountDetails*

*Address*



*AreaDeatils*



**Testing of 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");

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().getArea().getPincode());

}

}

**Output:**

Dilip

500.0

[8826111377, +91-88888888, +232388888888]

333

323232