**Spring Framework🌿**

* **Spring** is an **open-source, lightweight framework** for building Java applications.
* It provides comprehensive infrastructure support for developing **Java enterprise applications**.
* It helps developers create applications in a **loosely coupled, easily testable, and maintainable** way.

**Spring Core**

* Spring Core is the **foundational module of the Spring Framework** that provides the essential features for building Java-based applications.
* It mainly offers an **Inversion of Control (IoC)** container that manages the **lifecycle** and **configuration of application objects (beans)** through **Dependency Injection (DI).**
* This design pattern helps to promote loose coupling between components, making the code more modular, maintainable, and testable.

**Key Features of Spring Core**

* **Inversion of Control (IoC):** The container controls the creation, wiring, and lifecycle of objects.
* **Dependency Injection (DI):** Dependencies are injected into objects by the container rather than the objects creating dependencies themselves.
* **Modular and Flexible:** Developers can choose and configure only the components they need.
* **Supports Loose Coupling:** This design makes applications easier to maintain, test, and scale by ensuring components are not tightly dependent on each other. ***Changes in one component typically do not drastically affect related components in different layers***
* **Bean Lifecycle Management:** Manages the lifecycle and configuration of application beans.**Bean** is a lightweight object used in the Spring based applications.

**Role in Spring Framework**

* **Spring Core acts as the backbone of the entire Spring Framework**, upon which other modules like **Spring MVC, Spring Security, and Spring Data are built**.
* It simplifies Java application development by handling complexity around object creation and dependency management through configuration and annotations.

**IOC**

* **IOC stands for Inversion of Control.** IOC feature of spring framework is responsible for the management of **beans (objects)** in the Spring based applications.
* Before Spring framework developers were responsible to manage the **life cycle of a bean (object)** within the application like creating a new object, initializing it, utilizing it whenever required in the application and destroying it if the object is no more required in the application.
* But once after the Spring Framework was introduced, the responsibility of managing the ***bean (object) life cycle has been given to the spring framework itself*** where ***the control of managing the beans has been inverted from the developer to the spring framework, that's why it is called as IOC (Inversion Of Control).***

**Why was Spring Framework introduced?**

* Before Spring, Java developers often used **Enterprise JavaBeans (EJBs)** for building enterprise applications.
* But EJBs had several limitations like **:**
* **Complexity:** Configuration and deployment were complex.
* **Heavyweight:** Large overhead in terms of runtime and setup.
* **Difficult Testing:** Tight coupling made unit testing almost impossible
* **Rigid Architecture:** Changes in one class often required changes across many other classes.

**So, Spring was introduced to solve these problems**

**Spring Containers**

* Spring containers are the components of spring framework which are responsible for bean management i.e ***controlling the entire life cycle of pojo classes objects*** in Spring based applications.
* **POJO classes (plain old java objects)** are simple Java classes that do not have restrictions of any frameworks.
* They are :
* independent from any specific technology or framework,
* not inheriting any classes or interfaces.
* usually define domain models(**entity),** with ***private fields, all and no args contructors, getters , setters, overrriden methods like toString() and sometimes also the object references of another POJO/entity classes***.
* **The Spring container** is responsible for instantiating, configuring, and assembling these objects,**acts as an execution environment,** allowing developers to focus on business logic rather than boilerplate code related to object creation and management.
* It acts **as an interface** between ***low level platform functionality layer(JRE and all java operations)*** and ***component layer(DAO/Repository/Service layers)***
* It has following containers:

1. **IOC:** 
   1. Core
   2. J2EE
2. **Web:**

**(*built on the top of IOC, used in spring mvc while creating web applications)***

**IOC Container**

**A diagram of a factory

AI-generated content may be incorrect.**

* For an IOC container to perform any operation, POJO classes are required.
* We must implement **has-a** instead of **is-a** to establish relationship with multiple **entities, *(injecting an object/reference of another entity)***
* **IOC** container will read the instructions from **configuration xml file,** and perform some actions on **POJO class** like ***Dependency Injection(DI)*.**

**Dependency Injection in Spring based applications**

* Dependency Injection is the process where **spring containers** will provide the required objects in the application whenever there is a demand for the object.
* Its is process of taking/reading values/objects from **configuration file** and ***injecting*** *them into the* ***POJO classes.***
* ***E.g. Person*** *has a* ***mobile object.***

***(mobile******object*** *can be* ***injected*** *into* ***Person class)***

**Steps to create a Spring Maven Project**

**Step 1:** Create a simple maven project

**Step 2:** Go to [mvnrepository.com](https://mvnrepository.com/artifact/org.springframework/spring-context/6.2.7) and search for **Spring Context**, add 6.2.7(if 0 vulnerabilities)

**Step 3:** Create one **pojo class(Person)** in src/main/java/pojo\_classes

**Step 4:** Create one xml file **config.xml** in src/main/java and paste the code available in

[**github.com/ArpitKharche123/Configuration-Files /config.xml**](https://github.com/ArpitKharche123/Configuration-Files/blob/main/config.xml)

**Step 5:** Create a **Driver class(with main method)** in src/main/java/driver\_class

**Using Spring Containers**

**1.** Using **BeanFactory Container** is not recommended as **implementation class of BeanFactory interface, XmlBeanFactory is deprecated in Spring 3.1 and removed in Spring 5.**

**2.Using ApplicationContext Container**

* **Step 1:** In **config.xml,** add <**bean** id=*"demo"* class=*"pojo\_classes.Demo"*></**bean**> **in main beans tag.**
* Class attribute **must have fully qualified name of the POJO class.**
* **Step 2:** Use ApplicationContext or ConfigurableApplicationContext to load the info in config.xml file.
* **Step 3:** Fetch bean/object of POJO class created by Container in Driver class.

**[refer driver\_class.TestApplicationContextContainer , *pojo\_classes.Demo*]**

* **Note: *We can configure multiple beans of POJO classes in xml file using bean tag, and also load them in driver classes.***
* We can also have multiple configuration files in one project
* Create one folder for Configuration files and store all configuration files here.

**A screenshot of a computer

AI-generated content may be incorrect.**

* To access a configuration file:

e.g. ConfigurableApplicationContext context=new ClassPathXmlApplicationContext("**Configuration\_Files/config.xml**");

**[refer driver\_class.TestMultipleBeans]**

**Injecting Values**

* + 1. **Using setter**
* **Step1:** Create a POJO,Driver and config file
* **Step 2:** Create a separate config file, **setter\_injection\_config.xml**

.In config file, add:

* <!-- Employee bean info -->
* <**bean** id=*"employee"* class=*"pojo\_classes.Employee"*>
* <!-- Injecting values for properties of Employee -->
* <**property** name=*"name"* value=*"Michael"*></**property**>
* <**property** name=*"emp\_id"* value=*"ASD123"*></**property**>
* <**property** name=*"blood\_group"* value=*"A+"*></**property**>
* <**property** name=*"age"* value=*"23"*></**property**>
* </**bean**>
* ***Note: Property name must be same as the variable name in pojo class!!!***
* **Step 3:** In **POJO class,** add properties, **setters and getters for all properties.**
* **Step 4:** Provide implementation in **EmployeeDriver** class.

**[refer config.xml, pojo\_classes.Employee, driver\_class.EmployeeDriver]**

* ***Here, values provided in property tags are injected into bean by setters!***
  + 1. **Using Constructor**
* **Step 1:** Take POJO same as above(**add one all-arg and no-arg constructor)**
* **Step 2:** Create a separate config file, **injection\_by\_constructor\_config.xml.**

In config file, add:

***<!-- Employee bean info -->***

<**bean** id=*"employee"* class=*"pojo\_classes.Employee"*>

<**constructor-arg** value=*"Nitesh"*></**constructor-arg**>

<**constructor-arg** value=*"SWE546"*></**constructor-arg**>

<**constructor-arg** value=*"B+"*></**constructor-arg**>

<**constructor-arg** value=*"24"*></**constructor-arg**>

</**bean**>

* **Step 3:** Provide implementation in **EmployeeDriver**class.

**Dependency Injection(Injecting Object)**

1. **Using setter**

* **Step 1:** Create two pojo classes **Person(with setter) and Mobile,**
* **Step 2:** Create a reference of Mobile type into Person class.
* **Step 3:** In config file,add:

**<!-- Person and Mobile bean info -->**

<**bean** id=*"mobile\_id"* class=*"pojo\_classes.Mobile"*></**bean**>

<**bean** id=*"person"* class=*"pojo\_classes.Person"*>

**<!-- Injecting the mobile object -->**

<**property** name=*"mobile"* ref=*"mobile\_id"*></**property**>

</**bean**>

* ***Note:***
* ***Property name must be same as the variable name in pojo class!!!***
* ***Id of Mobile must be same as the ref value***
* **Step 4:** Provide Implementation in PersonMobileDriver class

1. **Using Constructor**

* **Step 1:** Take POJO same as above(**add one all-arg and no-arg constructor)**
* **Step 2:** Create a separate config file, **injection\_by\_constructor\_config.xml.**

In config file, add:

***<!-- Person and Mobile bean info Object Injection -->***

<**bean** id=*"mobile\_id"* class=*"pojo\_classes.Mobile"*></**bean**>

<**bean** id=*"person"* class=*"pojo\_classes.Person"*>

<!-- Injecting the mobile object -->

<**constructor-arg** ref=*"mobile\_id"*></**constructor-arg**>

</**bean**>

* **Step 3:** Provide implementation in **PersonMobileDriver** class.

**DI (Injecting List and Map)**

1. **Using setter**

* **Step 1:** Create one POJO class **Car(with setter)**
* **Step 2:** In config file,add:

***<!-- Car bean info -->***

<**bean** id=*"car"* class=*"pojo\_classes.Car"*>

<**property** name=*"model"* value=*"maybach"*></**property**>

<**property** name=*"brand"*>

<**value**>Mercedes</**value**>

</**property**>

<**property** name=*"tyres"*>

***<!-- Injecting List -->***

<**list**>

<**value**>MRF</**value**>

<**value**>Yokohama</**value**>

<**value**>CEAT</**value**>

<**value**>Hero</**value**>

</**list**>

</**property**>

<**property** name=*"price"*>

***<!-- Injecting Map -->***

<**map**>

<**entry** key=*"Base Manual"* value=*"1200000.0"*></**entry**>

<**entry** key=*"Base Auto"* value=*"1800000.0"*></**entry**>

<**entry** key=*"Top Manual"* value=*"2300000.0"*></**entry**>

<**entry** key=*"Top Auto"* value=*"3200000.0"*></**entry**>

</**map**>

</**property**>

</**bean**>

* **Step 3:** Provide implementation in **CarDriver** class.

1. **Using Constructor**

* **Step 1:** Take POJO same as above(**add one all-arg and no-arg constructor)**
* **Step 2:** Create a separate config file, **injection\_by\_constructor\_config.xml.**

In config file, add:

<!-- Car bean info -->

<**bean** id=*"car"* class=*"pojo\_classes.Car"*>

<**constructor-arg** value=*"ADV4X4"*></**constructor-arg**>

<**constructor-arg** value=*"TOYOTA"*></**constructor-arg**>

<!-- Injecting List -->

<**constructor-arg**>

<**list**>

<**value**>MRF</**value**>

<**value**>Yokohama</**value**>

<**value**>CEAT</**value**>

<**value**>Hero</**value**>

</**list**>

</**constructor-arg**>

<!-- Injecting Map -->

<**constructor-arg**>

<**map**>

<**entry** key=*"Base Manual"* value=*"1200000.0"*></**entry**>

<**entry** key=*"Base Auto"* value=*"1800000.0"*></**entry**>

<**entry** key=*"Top Manual"* value=*"2300000.0"*></**entry**>

<**entry** key=*"Top Auto"* value=*"3200000.0"*></**entry**>

</**map**>

</**constructor-arg**>

</**bean**>

* **Step 3:** Provide implementation in **CarDriver** class.

**Reading values from properties file into xml file**

* **Step 1:** Create a file named **prod\_info.properties**  in **src/main/java** and insert some data.
* **Step 2:** Create a POJO class named **Product** with some properties, **getters and setters**
* **Step 3:** In **config.xml,** add:

**<!-- Loading values from prod\_info.properties file-->**

**<context:property-placeholder**

location="classpath:prod\_info.properties"**/>**

**<bean** id="product" class="pojo\_classes.Product"**>**

**<property** name="name" value="${name}"**></property>**

**<property** name="cost" value="${cost}"**></property>**

**<property** name="brand" value="${brand}"**></property>**

**</bean>**

* **Note:**
* We have to give properties file name in location
* Name inside **${}** must match the name given in **properties file**
* **Step 4:** Provide implementation in **ProductDriver** class.

**Bean LifeCycle Methods**

* Whenever Spring manages a bean, it takes care of creating it, injecting dependencies, initializing it, and finally destroying it when the container shuts down.
* A Spring bean generally goes through these **phases**:

1. **Instantiation** → Object of bean class is created.
2. **Dependency Injection** → Spring injects required dependencies (via constructor/setter).
3. **Initialization (post-construct phase)** → After dependencies are set, Spring allows us to run **custom initialization logic** (this is like *post construct*).
4. **Ready for Use** → Bean can now be used by the application.
5. **Destruction (Pre-Destroy phase)** → Before container shuts down, Spring calls **custom cleanup logic** (this is like *pre destroy*).

* **init-method and destroy-method**

These are **XML-based configuration attributes** :

* which method to call **after bean creation** → *init-method*
* which method to call **before bean destruction** → *destroy-method*

**🔹 init-method (Post-Construct phase)**

* Called after Spring finishes dependency injection.
* Used to **initialize resources**, **validate data**, or **start connections**.
* Equivalent to the “post construct” phase.

**🔹 destroy-method (Pre-Destroy phase)**

* Called before the bean is destroyed (when the container closes).
* Used to **release resources**, **close connections**, or **cleanup temporary data**.
* Equivalent to the “pre destroy” phase.

**Implementation:**

* **Step 1:** Create **BeanLifecycle** with one **no-arg constructor,** methods with name: **postConstruct() and preDestruct()**
* **Step 2:** In **config.xml** , add:

***<!-- Bean LifeCycle Method attributes -->***

<**bean** id=*"bean\_life\_cycle"* class=*"pojo\_classes.BeanLifecycle"*

init-method=*"postConstruct"* destroy-method=*"preDestruct"*></**bean**>

* **Step 3:** Provide implementation in **BeanLifecycleDriver** class.

**Annotations used for IOC**

* Instead of using multiple tags in **configuration file,** we can make use of **some annotations in** out **POJO classes**, to make the task more **easy, readable and more understandable.**
* **@Component:**
* This annotation is a **class level annotation** , which is used with the class for which we want **Spring Container to manage its beans!**
* It can be used as a replacement of:

**<bean** id=*"xyz "* class=*"fully.qualified.name"*></**bean>**

* In config file, we have to add:
* <!-- @Component annotation -->
* <**context:component-scan** base-package=*"pojos"*></**context:component-scan**>

**[refer Configuration\_Files/minimal\_config.xml]**

**e.g. [refer pojos.Car]**

***@Component***

public class Car

🡪here, bean name will be automatically generated as **car**

**🡪automatic bean name generation is like:**

**Classname 🡪 bean name**

* Spring 🡪 spring
* SPRINGCORE 🡪 SPRINGCORE
* SpringCore 🡪 springCore

🡪We can also give cutom bean name like:

***@Component*("car")**

**[Refer Drivers.CarDriver for implementation]**

* **@Value**
* We can inject a value into a variable using **@Value annotation,** as we did using **setters and constructors**
* **E.g @Value(“xyz”)**
* **[refer pojos.Car and Drivers.CarDriver]**

**Eliminating use on xml configuration file, using Config class**

* We can implement **IOC** without using **xml configuration file,** with the help of **configuration class.**
* **Steps to make a class as a configuration class:**

**🡪Step 1:** Create a class and name it as **ConfigClass**

**🡪Step 2:** Use :

**@Configuration and @ComponantScan(“base\_package\_name”)** as a class level annotation with **ConfigClass**

**@Configuration:**

It is a **class-level annotations** which indicates that a class **declares one or more** [**@Bean**](eclipse-javadoc:%E2%98%82=spring_core/C:%5C/Users%5C/arpit%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/6.2.7%5C/spring-context-6.2.7.jar=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.2.7=/=/maven.scope=/compile=/=/maven.pomderived=/true=/%3Corg.springframework.context.annotation(Configuration.class%E2%98%83Configuration%E2%98%82Bean) **methods** and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

**@ComponentScan:** It is a class level annotation which is used with the configuration class.

This annotation is used to help the spring container to look for the classes annotated with **@Component , in a specified package** in order to manage the beans for the Component classes.

**[refer config\_classes.ConfigClass , pojos.Pen and Drivers.PenDriver]**

* **Note: *Here, configuration xml file is not needed at all!!!,***

**It is completely replaced by Configuration class**

**Injection using @Autowired**

* **@Autowired** is an annotation provided by Spring that tells the Spring container ***to automatically inject dependencies into a bean at runtime.***
* The sole purpose of **@Autowired** is to automatically inject (supply) the required **dependent objects (beans)** into another bean managed by the Spring container.
* **Example:**

**[refer pojos.Bag, pojos.Book and Drivers.BagAndBookDriver]**

**Using @Autowired with Interface type on reference variable**

* Consider the following **scenario 1:**

**A diagram of a ruler

AI-generated content may be incorrect.**

* **If we do:**
* public class Compass {
* *@Autowired*
* private Ruler ruler;
* As we know, **we cannot create an object for interface,** then the question is how the bean will be created?
* **Answer:**
* **Spring Container** will check is there any **implementing class** with ***@Component annotation***present or not.
* If present, ***Bean of that subclass type will be created and injected into Compass bean!!***
* ***[Refer pojos.Compass, interfaces.Ruler, Drivers.CompassDriver and pojos.SteelRuler]***
* Consider the following **scenario 2:**

**A diagram of a compass

AI-generated content may be incorrect.**

* **Now,** in this scenario, **both the subclass are qualified for bean creation, so Spring Container will get *confused* about which bean to inject in Compass object 😵**
* **So,** if we try to fetch bean in this scenario, we will get Exception like this:

**UnsatisfiedDependencyException: Error creating bean with name 'compass': Unsatisfied dependency expressed through field 'ruler': No qualifying bean of type 'interfaces.Ruler' available: expected single matching bean but found 2: plasticRuler,steelRuler**

* **So ,** to fix this, we can make use of two annotations:
* **@Qualifier(“bean\_id”):** 
  + - This annotation can be used with **reference variable, method or constructor.**
    - It is used to tell the **Spring Container** about the **one class among multiple qualified classes,** for which the **the bean is to be created**
* E.g. ***@Qualifier*("plasticRuler")**
* ***[Refer pojos.Compass, interfaces.Ruler, Drivers.CompassDriver and pojos.SteelRuler, pojos.PlasticRuler]***
* **@Primary**
* It is an alternative of **@Qualifier**
* It is a **class level annotation,** which can be used with **Either one of the qualified component class for bean creation.**
* **Spring Container** will create the bean of component class with **@Primary annotation (***in case of multiple qualified component classes***) and inject it** in the **Dependent class(Compass class)**
* It indicates that **a bean should be given preference** when multiple candidates are qualified .
* ***[Refer pojos.Compass, interfaces.Ruler, Drivers.CompassDriver and pojos.SteelRuler, pojos.PlasticRuler]***

**Bean creation using @Bean**

* **@Bean:** It is a **method level annotation** which is used to indicate that the method produces a bean, to be managed by spring container.
* We can make use of **@Bean annotation**, if we want IOC Container to create a bean for a class, which is not accessible/editable by us!!
* Here, without **@Component annotation,** we can create bean for a particular **pojo class.**
* We have to create a **factory method in Configuration class, which will create an object for a particular pojo class.**

**//factory method to return POJO object**

**@Bean("duster") //bean name/id**

public Duster getDuster()

{

return new Duster();

}

* *This method will get invoked once at context startup when ApplicationContext is created* ***by default(singleton scope))***

**[refer config\_classes.ConfigClass, pojos.Duster,Drivers.DusterDriver]**

**Bean Scope**

* In Spring Framework, a bean scope defines ***how long a bean lives and how many instances of it Spring should create and manage.***
* This can implemented using **@Scope annotation**

**@Scope:**

* + This annotation can be used as **method level annotation as well as class level annotation**
  + It is used to indicate **the type of the scope** used for returning a **produced bean** by the **spring container.**

**e.g Class Level:**

***@Component***

***@Scope*(“prototype”)**

public class Emp {

}

***Type of scope is singleton by default if not provided!!!***

***Method level:***

***@Bean*("duster")**

***@Scope*("prototype") *// new object will be created for each call of getBeans()***

public Duster getDuster() {

return new Duster();

}

* **Types of Bean Scopes**

1. **Singleton (Default)**

* Only **one instance** of the bean is created for the **entire Spring container.**
* Bean production is **eager** if scope is singleton, as it will create the instance at **context startup only**.
* So, all the beans fetched using **getBean() method** will point to same **reference**  *i.e reference/address of all beans will be same*

***e.g. @Component***

***@Scope*() or *@Scope*(“singleton”)**

public class Emp {

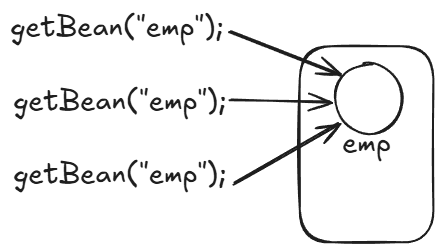
}

**Use when:**

* The bean is **stateless** (no per-request data).
* The bean performs **shared logic** or **configuration management**.
* Example: Services, Repositories, DAO, Controllers, Utility components.

**Why:**

* Memory efficient (only one instance).
* Better performance (no repeated creation).
* Suitable for shared, thread-safe operations.



1. **Prototype**

* A **new bean instance** is created each time it is requested from the container( **using getBean()**).
* Bean production is **lazy** if the scope is **prototype,** as it will create

**new bean instance** only when it is requested.

* All the **beans** fetched by **getBeans(),** will be pointed to **different objects.** *i.e reference/address of all beans will be different*

**Use when:**

* You need a fresh instance every time (stateful objects).
* The bean holds **request-specific or session-specific data**.

.

**Why:**

* Avoids unwanted data sharing between threads or requests.
* Prevents concurrency issues in beans.

**A diagram of a diagram

AI-generated content may be incorrect.**

[Refer Drivers.EmpDriver, pojos.Emp for **class level scope annotation]**

[Refer config\_classes.ConfigClass, DusterDriver **for method level scope annotation]**

**Reading Values from properties file and injecting It using annotations**

* **Step 1:** Use ***@PropertySource*("classpath:prod\_info.properties")** as a **class-level annotation** in **ConfigClass.**
* **Step 2:** Use ***@Value*(value = "${property\_name }")** with fields to which we want to inject the values.
* **Step 3:** Provide implementation to **ProdInfoDriver class**

**[Refer config\_classes.ConfigClass, pojos.ProdInfo, Drivers.ProdInfoDriver and prod\_info.properties]**