Spring Core Notes **Hariprasanth A H**

What is Spring

Spring is a lightweight Java framework used to build powerful and scalable applications.
It helps manage objects and their dependencies using **Inversion of Control (IoC)**.
It supports features like **web development**, **database access**, and **security**.
Spring makes applications easier to test, maintain, and scale.
It is called a **"framework of frameworks"** because it integrates many technologies in one place.

Spring is preferred over EJB because it is **lightweight**, **easier to configure**, **more testable**, and gives developers **greater flexibility and control**. It promotes cleaner code using POJOs and integrates seamlessly with modern technologies.

Advantages of Spring Framework

- **Lightweight**
  - Requires minimal memory and CPU; no need for heavy Java EE containers.
- **Open Source and Free**
  - No licensing cost; backed by a strong community.
- **Loose Coupling**
  - Promotes separation of concerns using Dependency Injection (DI).
  - **Loose Coupling** means that classes are **independent** and only know about each other through **interfaces or abstractions**, not through concrete implementations. This makes your code **flexible**, **easier to test**, and **easier to maintain**.
  - **Example**

```
interface Engine {

    void start();

}
```

```
class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol Engine Started");
    }
}

class ElectricEngine implements Engine {
    public void start() {
        System.out.println("Electric Engine Started");
    }
}

class Car {
    private Engine engine;

        public Car(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Engine engine = new ElectricEngine();
        Car car = new Car(engine);
        car.drive();
    }
}
```

- Car doesn't care what kind of engine it has — it just depends on the Engine interface.
- You can switch engines easily without changing the Car class.
- Supports **flexibility**, **testability**, and **maintainability**.

- **Inversion of Control (IoC)**
  - Spring container manages object creation and dependencies.
- **Dependency Injection (DI)**
  - Automatically injects required objects, reducing boilerplate code.
- **Cross-platform Support**
  - Works with Java SE, Java EE, and cloud environments.

What is a Design Principle in Java?

Design principles in Java are **general guidelines or best practices** that help developers write **clean, maintainable, reusable, and scalable** code.

They are **not rules** or **code syntax**, but rather **concepts** you follow while designing software systems.

Why are Design Principles Important?

- Help reduce **code duplication**

- Make your code more **flexible** and **easy to extend**
- Improve **readability**, **testability**, and **maintainability**
- Encourage **loose coupling** and **high cohesion**

Common Design Principles in Java

Single Responsibility Principle (SRP)

A class should have only one reason to change

```
package com;

class Report {
    public String getText() {
        return "Report Data";
    }
}

class ReportPrinter {
    public void print(Report report) {
        System.out.println(report.getText());
    }
}
```

Now, **each class has a single job**

2. **Open/Closed Principle (OCP)**
Classes should be open for extension but closed for modification.

```
package com;

interface PaymentMethod {
    void pay();
}

class CardPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Paid by card");
    }
}

class UpiPayment implements PaymentMethod {
```

```
    public void pay() {
        System.out.println("Paid by UPI");
    }
}

class PaymentService {
    public void makePayment(PaymentMethod method) {
        method.pay();
    }
}
```

Easily **add new payment types** without modifying PaymentService.

3. **Liskov Substitution Principle (LSP)**
Subclass should be replaceable for its superclass without altering program behavior.

```
interface Bird { }

interface FlyingBird extends Bird {
    void fly();
}

class Sparrow implements FlyingBird {
    public void fly() {
        System.out.println("Flying...");
    }
}

class Ostrich implements Bird {
    // Ostrich doesn't fly – no fly() method needed
}
```

No incorrect behavior when replacing subclass with superclass

4. **Interface Segregation Principle (ISP)**

```
interface Printer {

    void print();

}



interface Scanner {

    void scan();

}
```

```
class BasicPrinter implements Printer {
    public void print() {
        System.out.println("Printing...");
    }
}
```

5. Dependency Inversion Principle (DIP)

Depend on abstractions, not concrete implementations.

```
interface Database {
    void connect();
}

class MySQLDatabase implements Database {
    public void connect() {
        System.out.println("Connected to MySQL");
    }
}

class App {
    private Database db;

    public App(Database db) {
        this.db = db;
    }
```

```
    public void start() {
        db.connect();
    }
}
```

You can **easily switch** from MySQL to Oracle, etc.

Inversion of Control (IoC) in Spring

Inversion of Control (IoC) in Spring is a design principle in which the control of creating and managing objects (beans) is given to the **Spring IoC container**, rather than the programmer manually creating objects using new

- **Inversion of Control (IoC)** is a core concept in the Spring Framework.
- It means **giving control of object creation and dependency management to the framework**.
- Instead of using new to create objects, Spring creates and manages them for you.
- The control of creating, configuring, and managing objects is **inverted from the developer to the Spring container**.
- This reduces tight coupling between classes and improves flexibility.
- Spring uses **Dependency Injection (DI)** to implement IoC.

IoC Container in Spring

The **IoC (Inversion of Control) Container** in Spring is the **core part of the Spring Framework** responsible for

Creating objects (beans)
Managing their lifecycle
Injecting dependencies automatically
Configuring beans using XML, annotations, or Java code

The **Spring IoC Container** is a component of the Spring Framework that **instantiates, configures, wires, and manages the lifecycle** of Spring beans using the principles of **Inversion of Control** and **Dependency Injection**.

Dependency Injection

Dependency Injection (DI) is a design pattern used in object-oriented programming to **provide the dependencies of a class from outside**, rather than creating them inside the class.

In DI, one object gets the objects it depends on (called **dependencies**) from an external source like a **constructor**, **setter method**, or **framework container** (e.g., Spring).

This helps to **achieve loose coupling**, making the code **more flexible, easier to test, and maintainable**.

Dependency Injection is a key concept of **Inversion of Control (IoC)**, where the control of creating and managing objects is shifted from the class to a container.

It is widely used in frameworks like **Spring**, where objects are automatically injected

Spring Framework provides two of the most fundamental and important packages, they are
the **org.springframework.beans** and **org.springframework.context** packages. Code in these packages provides the basis for Spring's **Inversion of Control/Dependency Injection** features. Spring containers are responsible for creating bean objects and injecting them into the classes. The two containers are namely

1. **BeanFactory(I)** - Available in org.springframework.beans.factory package.
2. **ApplicationContext(I)** - Available in org.springframework.context package.

BeanFactory Interface

BeanFactory is the **main interface** in Spring used to **access and manage beans**.

It is the actual **container** that:

- **Creates** objects (called beans)
- **Configures** them
- **Manages** them throughout their life

These beans can **work together** (collaborate) and **depend on each other**.

Their connections (dependencies) are defined in the **configuration file** (like `beans.xml`), which BeanFactory reads.

Each bean in the configuration has a **unique name (ID)** to identify it.

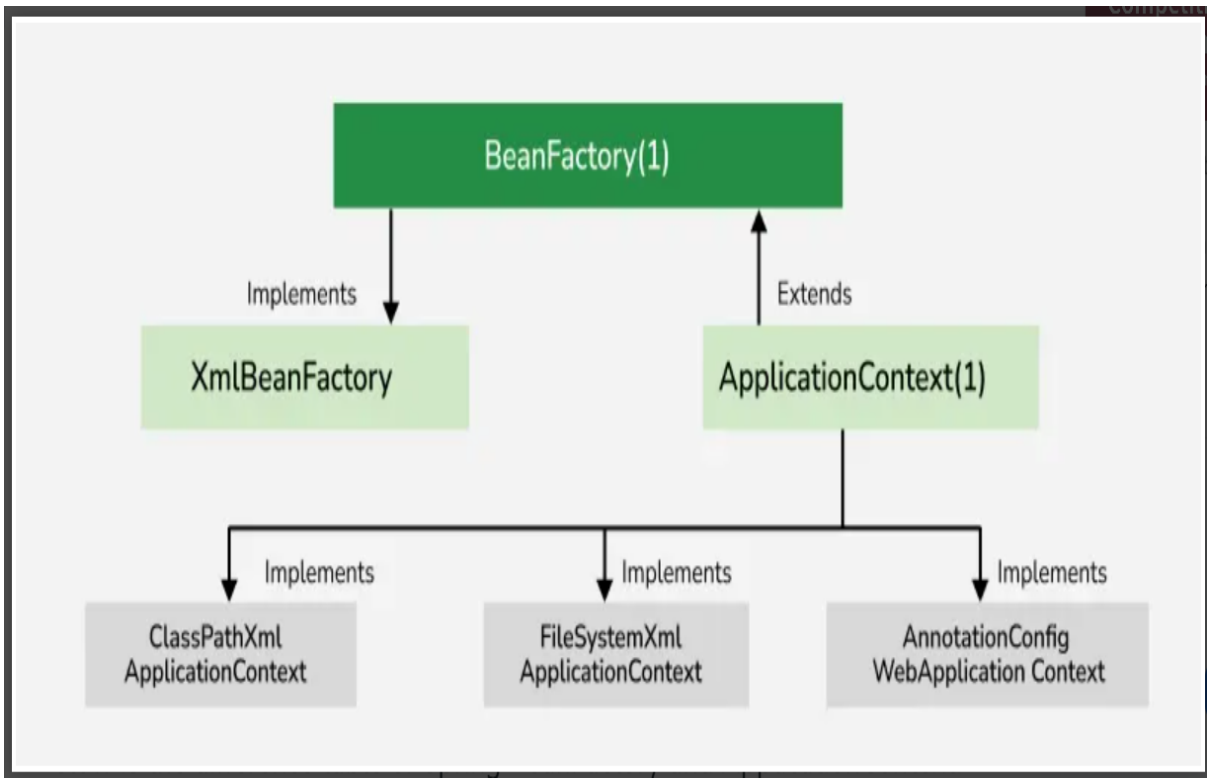
One commonly used class that implements BeanFactory is **`XmlBeanFactory`**, which is found in the package

Note:

- ***`XmlBeanFactory`*** *is deprecated in Spring 3.1 and removed in Spring 4.0. It was used for loading Spring beans from XML configuration files.*
- *`BeanFactory` is not deprecated but less commonly used directly in favor of `ApplicationContext`.*

ApplicationContext Interface

This interface is designed on top of the BeanFactory interface. The ApplicationContext interface is the advanced container that enhances BeanFactory functionality in a more framework-oriented style. While the BeanFactory provides basic functionality for managing and manipulating beans, often in a programmatic way, the ApplicationContext provides extra functionality. There are so many implementation classes that can be used such as **ClassPathXmlApplicationContext**, **FileSystemXmlApplicationContext**, **AnnotationConfigApplicationContext** etc.



Steps to Create an IoC Container Using XML in Java using Spring

Step 1: Add Spring Core Dependency

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.34</version>
</dependency>
```

Step 2: Create Your Bean Classes (POJOs)

```
public class MessageService {
    public void sendMessage() {
```

```
        System.out.println("Message sent successfully!");
    }
}
```

Step 3: Create XML Configuration File (`beans.xml`)

```
<?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="msgService" class="MessageService" />


</beans>
```

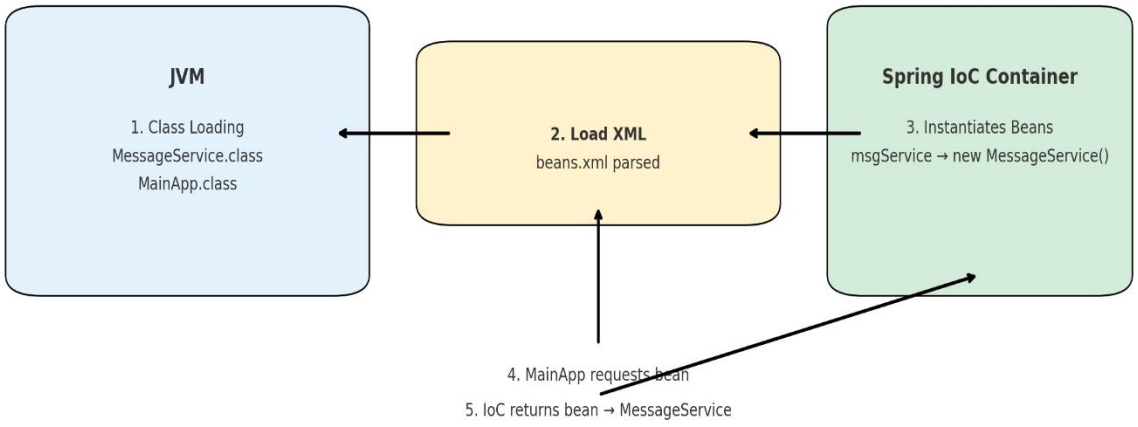
Step 4: Load IoC Container in Main Class

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

public class MainApp {
    public static void main(String[] args) {
        // Load IoC container from XML
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

```
        // Get the bean from container
        MessageService service = context.getBean("msgService",MessageService.class);

        // Use the bean
        service.sendMessage();
    }
}
```



Step 1: Class Loading

- JVM loads the `MessageService` and `MainApp` classes into memory.

Step 2: XML Configuration Loaded

- `ClassPathXmlApplicationContext("beans.xml")` loads the `beans.xml` file.
- Spring parses the XML and identifies the `<bean>` definition for `msgService`.

Step 3: Bean Instantiation by IoC Container

- Spring creates an object of `MessageService` (`new MessageService()`).
- This instance is stored in the **IoC container** and managed for future use.

Step 4: Bean Retrieval

- In `MainApp`, `context.getBean("msgService", MessageService.class)` is called.
- The IoC container returns the already-created `MessageService` bean.

Step 5: Bean Usage

- The `sendMessage()` method is called on the `MessageService` object.

@Configuration

- This annotation is used to **tell Spring** that the class contains **bean definitions**.
- It **replaces the XML configuration files** used in older Spring versions.
- When Spring sees a class annotated with @Configuration, it knows to **look inside** the class for methods that define beans using @Bean.
- It's part of the **Java-based configuration** approach in Spring.
- "This class contains instructions for how to set up and configure objects (beans) that Spring should manage."

@ComponentScan

- This tells Spring **where to look** in the project for **components to register as beans**.

- It **scans the specified package(s)** and automatically finds classes annotated with @Component, @Service, @Repository, or @Controller.
- This scanning helps Spring **automatically discover and create objects** without manual registration.
- Look in these folders and automatically find any components you need to manage

@Component

- Marks a class as a **Spring-managed bean**.
- Any class annotated with @Component will be **automatically detected and instantiated** by Spring during the component scan.
- It is a **generic stereotype annotation** used for any Spring-managed class (logic, helper, etc.).

@Autowired

- This tells Spring to **automatically inject (provide) dependencies** into a class.
- You don't need to manually write code to create objects; Spring will **figure out what to inject and where**.
- It works on fields, constructors, and setter methods.
- Spring will search for the correct bean and **inject it where needed**.

@Value

- Used to **inject values into variables**, especially values from application.properties or environment variables.
- You can use it to inject **simple values like strings, numbers, or expressions**.
- Commonly used to **configure values** like app name, URLs, or limits that may vary between environments.

@Bean

- Used inside a class marked with @Configuration.
- It **manually defines a bean**, which means you're telling Spring how to create a particular object.
- You use @Bean when you need **more control** over object creation, or when the object is from a **third-party library** that doesn't use @Component

@Repository

- A **specialized type of @Component**, meant to indicate that the class is responsible for **database access or operations**.
- It is used in the **data access layer** of your application.
- Also provides **exception translation**, converting database-specific exceptions into **Spring's unified exception hierarchy**.

@Qualifier

- When multiple beans of the same type exist, Spring gets **confused** about which one to inject.
- @Qualifier helps Spring choose the **exact bean** by **name**.
- It works **along with @Autowired** to clarify which bean you want.

@Primary

- This annotation is also used when **multiple beans of the same type** are present.
- It tells Spring:

  "If no @Qualifier is given, use **this one** by default."

- It's a way to **set a default preference** when multiple options are available.

@Scope

- By default, Spring beans are **singleton**, meaning only **one instance** is created and reused.
- @Scope allows you to change the **lifecycle and visibility** of a bean.

Scope	Meaning
singleton	One shared instance (default)
prototype	A new instance every time it's needed
request	One per HTTP request (web apps)
session	One per HTTP session (web apps)

@PostConstruct

- This marks a **method that should run automatically after the bean is created and dependencies are injected**.
- It's often used for **initialization logic**, like opening a file, database connection, or preloading data.

@PreDestroy

- This marks a method that runs **before the bean is destroyed**.
- It's used to **release resources** like closing connections, saving data, etc.
- Works for **singleton beans**, since Spring manages their full lifecycle.