SPRING CORE BASICS

Spring:

The **Spring Framework** is a powerful, feature-rich, and widely-used **open-source framework** for building **Java applications**, especially enterprise-level applications. It provides comprehensive infrastructure support for developing Java applications and is particularly known for making development easier by promoting **loose coupling**, **dependency injection**, and **aspect-oriented programming**.

🔑 Key Features of Spring Framework

1. **Dependency Injection (DI) / Inversion of Control (IoC)**
   Helps manage object creation and their dependencies automatically. Instead of creating objects manually, Spring injects them where needed.
2. **Aspect-Oriented Programming (AOP)**
   Allows separation of cross-cutting concerns (like logging, security, or transaction management) from the business logic.
3. **Spring MVC (Model-View-Controller)**
   A web framework built on top of Spring for building web applications. It follows the MVC design pattern.
4. **Transaction Management**
   Abstracts the complexity of transaction management and allows consistent programming models.
5. **Spring Boot**
   A Spring project that simplifies the development of stand-alone, production-grade Spring applications with minimal configuration.

6. **Spring Data**
   Simplifies data access and integrates with databases like MySQL, MongoDB, etc.
7. **Spring Security**
   Provides authentication and authorization features for securing applications.
8. **Spring Integration / Spring Cloud**
   Facilitates building distributed systems and microservices.

✅ Benefits of Using Spring

- **Modular**: You can use only the parts you need.
- **Testable**: Encourages good design that leads to testable code.
- **Flexible**: Works with various other Java frameworks and technologies.
- **Widely Adopted**: Strong community support and lots of learning resources.

🌱 Simple Example of Spring IoC

```
@Component
public class HelloService {
   public String sayHello() {
      return "Hello, Spring!";
   }
}


@RestController
```

```
public class HelloController {

    @Autowired
    private HelloService helloService;

    @GetMapping("/hello")
    public String hello() {
        return helloService.sayHello();
    }
}
```

Spring automatically creates and injects the HelloService bean into HelloController.

SpringBoot:

💥 What is Spring Boot?

Spring Boot is an **extension of the Spring Framework** that makes it easier and faster to develop **standalone**, **production-ready** Spring-based applications with **minimal configuration**.

It was created to simplify the complexity of setting up Spring applications, especially for beginners or rapid development.

🚀 Key Goals of Spring Boot

- **Auto Configuration**: Automatically configures your application based on the libraries on the classpath.
- **Standalone Applications**: No need to deploy to an external server; you can run the application like any Java program (java -jar).

- **Production Ready**: Includes built-in tools like metrics, health checks, and externalized configuration.
- **Opinionated Defaults**: Offers sensible defaults so you don't have to configure everything from scratch.

✅ Advantages of Spring Boot

Feature	**Benefit**
☆ No XML Configuration	Uses annotations and auto-config
⚙ Embedded Servers	Comes with Tomcat/Jetty/Undertow
▯ Easy Testing	Built-in test support
📦 Dependency Management	Handles libraries via spring-boot-starter packages
🔒 Integration Ready	Easily integrates with databases, security, messaging, etc.

🔧 Common Spring Boot Annotations

- @SpringBootApplication: Main annotation to enable auto-configuration and component scanning.
- @RestController: Creates RESTful web services.
- @RequestMapping / @GetMapping: Maps HTTP requests.
- @Autowired: Injects beans automatically.

🛠 Example: Simple Spring Boot Application

Main Class

```
@SpringBootApplication

public class MyApp {

   public static void main(String[] args) {

      SpringApplication.run(MyApp.class, args);

   }

}
```

REST Controller

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, Spring Boot!";
    }
}
```

Output (when you visit http://localhost:8080/hello)

```
Hello, Spring Boot!
```

📦 **Dependency in pom.xml (Maven)**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Dependency injection:

✅ What is Dependency Injection (DI) in Spring?

Dependency Injection (DI) is a **design pattern** used in Spring Framework to **manage the dependencies** between different components (classes) of your application.

Instead of a class creating its own dependencies, Spring **injects them** at runtime. This makes the code more **loosely coupled**, **easier to test**, and **more maintainable**.

💡 Types of Dependency Injection in Spring

Type	Description
Constructor Injection	Dependencies are passed via the class **constructor**
Setter Injection	Dependencies are set via **setter methods**
Field Injection	Dependencies are injected **directly into the fields** (not recommended for unit testing)

Sure! Let's simplify each type of **Dependency Injection** example so it's very easy to understand and run in **Spring Tool Suite (STS)**.

✅ 1. Constructor Injection (Simple Version)

🗁 **Package: com.example.constructor**

🔹 Student.java

```
package com.example.constructor;


public class Student {
    public void show() {
        System.out.println("Constructor Injection: Hello from Student");
    }
}
```

🔹 College.java

```
package com.example.constructor;


public class College {
    private Student student;
    // Constructor Injection
    public College(Student student) {
        this.student = student;
    }

    public void show() {
        student.show();
    }
}
```

MainApp.java

```
package com.example.constructor;

import org.springframework.context.annotation.*;

@Configuration
public class MainApp {

    @Bean
    public Student student() {
        return new Student();
    }

    @Bean
    public College college() {
        return new College(student());
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainApp.class);
        College college = context.getBean(College.class);
        college.show();
    }
}
```

Explanation:

- We declare a Student object inside the College class.
- Instead of creating the Student object manually inside College, we pass it using a **constructor**.
- Spring is responsible for creating the Student object and providing it to College.

Why Use It?

- Ensures that the dependency (Student) is provided at the time the object (College) is created.
- Best for **required dependencies**.

✅ 2. Setter Injection (Simple Version)

🗁 Package: com.example.setter

◈ Student.java

```
package com.example.setter;
```

```
public class Student {
    public void show() {
        System.out.println("Setter Injection: Hello from Student");
    }
}
```

◆ **College.java**

```
package com.example.setter;

public class College {
    private Student student;

    // Setter method
    public void setStudent(Student student) {
        this.student = student;
    }

    public void show() {
        student.show();
    }
}
```

◆ **MainApp.java**

```
package com.example.setter;

import org.springframework.context.annotation.*;

@Configuration
```

```
public class MainApp {

    @Bean
    public Student student() {
        return new Student();
    }

    @Bean
    public College college() {
        College college = new College();
        college.setStudent(student());
        return college;
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MainApp.class);
        College college = context.getBean(College.class);
        college.show();
    }
}
```

Explanation:

- We use a **setter method** (setStudent()) to assign the dependency.
- Spring creates the Student object and calls the setter to inject it into College.

Why Use It?

- Useful for **optional dependencies** or when you want to change the dependency later.
- More flexible, but less safe if the setter is never called (can lead to NullPointerException if not handled properly).

✅ 3. Field Injection using @Autowired (Simple Version)

🗁 Package: com.example.field

◈ Student.java

```
package com.example.field;


import org.springframework.stereotype.Component;


@Component
public class Student {
    public void show() {
        System.out.println("Field Injection: Hello from Student");
    }
}
```

◈ College.java

```
package com.example.field;


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

```
@Component
public class College {

    @Autowired
    private Student student;

    public void show() {
        student.show();
    }
}
```

❖ **MainApp.java**

```
package com.example.field;

import org.springframework.context.annotation.*;

@Configuration
@ComponentScan("com.example.field")
public class MainApp {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(MainApp.class);
        College college = context.getBean(College.class);
        college.show();
    }
}
```

IOC Container:

Explanation:

- We simply declare the Student field and use @Autowired to tell Spring to **automatically inject** the required bean.
- No constructor or setter needed — Spring handles everything.

Why Use It?

- Very concise and easy.
- Common for small or quick projects.
- **Not recommended for unit testing** or large-scale systems because it's harder to mock dependencies.

⚖ Comparison Summary

Type	Description	Best Use Case
Constructor	Inject dependency via constructor	Mandatory dependencies
Setter	Inject dependency using setter methods	Optional or changeable deps
Field	Let Spring inject directly via @Autowired	Quick setup, but less testable

---

🔍 Output for All 3 Examples

bash

CopyEdit

```
Constructor Injection: Hello from Student

Setter Injection: Hello from Student

Field Injection: Hello from Student
```

Each method ultimately prints a message from the Student class. The difference is **how the Student object gets inside the College class.**

🔲 What is the IoC Container in Spring?

The **IoC (Inversion of Control) Container** is the **core of the Spring Framework**. It's responsible for:

- Creating objects (beans)
- Injecting dependencies
- Managing the lifecycle of those beans

💡 What is Inversion of Control (IoC)?

Inversion of Control means the control of creating and managing objects is shifted **from the developer to the Spring container**.

Instead of writing:

java

CopyEdit

```
Student s = new Student(); // Developer creates it manually
```

We let Spring handle it:

java

CopyEdit

```
ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
Student s = context.getBean(Student.class);
```

🔲 Types of IoC Containers in Spring

Container Type	**Description**
BeanFactory	Lightweight container, basic features (rarely used now)

Container Type	Description
ApplicationContext	Advanced container with full features like event handling, AOP, internationalization

💠 **ApplicationContext** is the preferred and commonly used container in Spring applications.

🔧 How It Works (with Example)

Example:

java

CopyEdit

```
@Configuration
public class AppConfig {

    @Bean
    public Student student() {
        return new Student();
    }

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        Student s = context.getBean(Student.class);
        s.show();
    }
}
```

What happens behind the scenes?

1. Spring creates an **ApplicationContext**.

2. It scans @Bean definitions or @Component classes.
3. It creates and stores those beans in its **container**.
4. When you call getBean(), it returns the ready-to-use object.

📦 Bean Lifecycle Managed by IoC Container

1. **Instantiation** – Create object.
2. **Populate properties** – Inject dependencies.
3. **Initialization** – Custom init methods (if any).
4. **Destruction** – Cleanup before app shutdown.

▯ In Simple Words

The **IoC container is like a factory**. You give it a blueprint (class), and it gives you a ready-made object (bean) with all parts assembled (dependencies injected).

XML Based Configuration:

Before **annotation** and **Java-based config**, Spring used **XML files** to define beans and their dependencies.

You write the bean definitions in an XML file (applicationContext.xml), and Spring reads this file to create and inject objects.

Sure! Here's a **simple example** using **only XML configuration** in Spring with detailed explanation.

Example: XML Configuration for Dependency Injection

1. Student.java

```
package com.example.xml;


public class Student {
    public void show() {
        System.out.println("Hello from Student (XML Config)");
    }
}
```

2. College.java

```
package com.example.xml;


public class College {
    private Student student;


    // Setter for injection
    public void setStudent(Student student) {
```

```java
        this.student = student;
    }

    public void show() {
        student.show();
    }
}
```

3. applicationContext.xml

```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">

    <!-- Define Student bean -->
    <bean id="student" class="com.example.xml.Student" />

    <!-- Define College bean and inject Student bean via setter -->
    <bean id="college" class="com.example.xml.College">
        <property name="student" ref="student" />
    </bean>

</beans>
```

4. MainApp.java

```
package com.example.xml;

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

public class MainApp {
   public static void main(String[] args) {
      // Load Spring context from XML configuration file
      ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

      // Get College bean from container
      College college = (College) context.getBean("college");

      // Call method to verify dependency injection
      college.show();
   }
}
```

Explanation

- **applicationContext.xml** defines the beans:
  - student bean of type Student.
  - college bean of type College.
- The college bean has a **property** named student which refers to the student bean.
- This tells Spring to **inject the student object into the college object via the setter** method setStudent() automatically.

- In MainApp, when we load the Spring context (ClassPathXmlApplicationContext), Spring:
    - Reads the XML,
    - Creates Student and College instances,
    - Injects Student into College,
    - Manages the lifecycle of these objects.
- When we call college.show(), it calls student.show(), and you see the message.

Output

```
Hello from Student (XML Config)
```

JAVA Based:

Absolutely! Here's the **Java-based configuration** version of the exact same example you had with XML — simple, clean, and easy to understand.

Java-Based Configuration Example for Dependency Injection

---

1. Student.java

```
package com.example.javaconfig;


public class Student {
    public void show() {
        System.out.println("Hello from Student (Java Config)");
    }
}
```

2. College.java

```
package com.example.javaconfig;

public class College {
   private Student student;

   // Setter for dependency injection
   public void setStudent(Student student) {
      this.student = student;
   }

   public void show() {
      student.show();
   }
}
```

3. AppConfig.java (Java Configuration Class)

```
package com.example.javaconfig;

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

@Configuration
public class AppConfig {

   @Bean
```

```
    public Student student() {
        return new Student();
    }

    @Bean
    public College college() {
        College college = new College();
        college.setStudent(student());  // Inject Student bean via setter
        return college;
    }
}
```

4. MainApp.java

```
package com.example.javaconfig;

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

public class MainApp {
    public static void main(String[] args) {
        // Load Spring context from Java config class
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        // Get College bean from container
        College college = context.getBean(College.class);
        // Call method to verify dependency injection
        college.show();
    }
}
```

Explanation

- AppConfig.java replaces the XML file.
- It is annotated with @Configuration — meaning **this class contains bean definitions**.
- Each method annotated with @Bean returns an object to be managed by Spring (like beans in XML).
- college() method creates a College object and **injects the Student bean using the setter** (similar to the XML <property> tag).
- In MainApp, Spring container is initialized using AnnotationConfigApplicationContext with the AppConfig class.
- Spring creates and wires beans based on the config, then you use them normally.

Output

```
Hello from Student (Java Config)
```

ANNOTATION BASED CONFIG:

Sure! Here's the **annotation-based configuration** version of the same example, which is even simpler because Spring automatically detects and wires the beans using annotations.

Annotation-Based Configuration Example for Dependency Injection

1. Student.java

```
package com.example.annotation;

import org.springframework.stereotype.Component;

@Component  // Marks this class as a Spring-managed bean
public class Student {
   public void show() {
      System.out.println("Hello from Student (Annotation Config)");
   }
}
```

2. College.java

```
package com.example.annotation;

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

@Component  // Marks this class as a Spring-managed bean
public class College {

   @Autowired  // Spring injects the Student bean here automatically
   private Student student;

   public void show() {
      student.show();
```

```
    }
}
```

3. MainApp.java

```
package com.example.annotation;

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

@Configuration
@ComponentScan("com.example.annotation")  // Scan this package for
@Component classes
public class MainApp {

    public static void main(String[] args) {
        // Load Spring context and scan for beans automatically
        ApplicationContext context = new
AnnotationConfigApplicationContext(MainApp.class);

        // Get College bean from container
        College college = context.getBean(College.class);

        // Call method to verify dependency injection
        college.show();
    }
}
```

Explanation

- @Component marks Student and College as Spring-managed beans.
- @Autowired on the student field tells Spring to automatically inject the Student bean.
- @ComponentScan tells Spring to scan the specified package to find classes with @Component.
- No XML or manual wiring is needed.
- The Spring container automatically creates and injects dependencies at runtime.

Output

```
Hello from Student (Annotation Config)
```

Summary of the three approaches for the same example:

Configuration Type	**How Beans Are Defined**	**How Dependencies Are Injected**
XML	Beans declared in XML file	<property> tags in XML
Java-based Config	Beans declared with @Bean methods	Setters or constructors in Java
Annotation-based Config	Beans detected with @Component	@Autowired annotation

---

==

LIFE CYCLE METHODS:

Sure! Let me explain the **life cycle methods of Spring beans** focusing on three ways to define them: using **XML configuration**, **implementing Spring interfaces**, and **using annotations/methods**.

🌱 Spring Bean Life Cycle Overview

When a Spring bean is created and destroyed, Spring container calls specific **life cycle callback methods** to allow you to add custom initialization and cleanup logic.

---

1. Using XML Configuration

You can specify **init-method** and **destroy-method** attributes in the <bean> tag in your XML configuration.

Example

```
<bean id="student" class="com.example.Student"
      init-method="init"
      destroy-method="cleanup" />
```

Java class with methods:

```
public class Student {
   public void init() {
      System.out.println("Student: init method called");
   }

   public void cleanup() {
      System.out.println("Student: destroy method called");
   }
}
```

}

- init-method is called **after bean creation and dependency injection**.
- destroy-method is called **before the bean is removed from the container** (when context is closed).

---

2. Implementing Spring Interfaces

Spring provides interfaces that you can implement for life cycle callbacks.

a) InitializingBean — for initialization

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


public class Student implements InitializingBean {
   @Override
   public void afterPropertiesSet() throws Exception {
      System.out.println("Student: afterPropertiesSet() called");
   }
}
```

- afterPropertiesSet() is called after Spring sets all bean properties.

b) DisposableBean — for destruction

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


public class Student implements DisposableBean {
   @Override
   public void destroy() throws Exception {
      System.out.println("Student: destroy() called");
   }
}
```

- destroy() is called when the container shuts down and bean is destroyed.

---

3. Using Annotations

Spring also supports annotations for life cycle callbacks:

- @PostConstruct — for initialization
- @PreDestroy — for destruction

Example:

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

public class Student {

    @PostConstruct
    public void init() {
        System.out.println("Student: @PostConstruct init method called");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Student: @PreDestroy destroy method called");
    }
}
```

Note: For older versions of Spring, use javax.annotation instead of jakarta.annotation.

Summary Table

Life Cycle Method	**How to Define**	**When Called**
Initialization	XML: init-method attribute	After bean creation & injection
Initialization	Implement InitializingBean interface, method afterPropertiesSet()	After bean creation & injection
Initialization	Annotate method with @PostConstruct	After bean creation & injection
Destruction	XML: destroy-method attribute	When context is closing & bean destroyed
Destruction	Implement DisposableBean interface, method destroy()	When context is closing & bean destroyed
Destruction	Annotate method with @PreDestroy	When context is closing & bean destroyed

---

Important:

- To see destruction methods called, the application context must be **closed** properly.
- For example, if you use ClassPathXmlApplicationContext, call context.close().

===

BEAN SCOPES:

Absolutely! Let me explain **Bean Scopes in Spring** with simple examples and clear explanations.

✵ What is Bean Scope?

Bean Scope defines the lifecycle and visibility of a bean in the Spring container — basically **how and when Spring creates new instances** of a bean.

Common Bean Scopes in Spring

Scope Name	Description	When a New Bean is Created
singleton	One shared instance per Spring container (default)	One instance per container, reused every time
prototype	New instance every time requested	Every request for bean creates a new object
request	One instance per HTTP request (web apps)	New instance per HTTP request
session	One instance per HTTP session (web apps)	New instance per user session
application	One instance per ServletContext (web apps)	One per ServletContext
websocket	One instance per WebSocket session	New per WebSocket session

---

1. Singleton Scope (Default)

- Spring creates **only one instance** of the bean per container.
- Every time you request the bean, you get the **same instance**.

Example

```
@Component
@Scope("singleton")  // This is default, so you can omit it
public class Student {
}
```

Test

```
Student s1 = context.getBean(Student.class);
Student s2 = context.getBean(Student.class);
System.out.println(s1 == s2);  // Output: true (same object)
```

---

2. Prototype Scope

- Spring creates **a new instance** every time the bean is requested.

Example

```
@Component
@Scope("prototype")
public class Student {
}
```

Test

```
Student s1 = context.getBean(Student.class);
Student s2 = context.getBean(Student.class);
System.out.println(s1 == s2);  // Output: false (different objects)
```

---

3. Request Scope (Web applications only)

- New bean instance for each HTTP request.

```
@Component
@Scope("request")
```

```
public class Student {
}
```

---

4. Session Scope (Web applications only)

- New bean instance for each user HTTP session.

```
@Component
@Scope("session")
public class Student {
}
```

How to Use Bean Scopes in XML Configuration

```
<bean id="student" class="com.example.Student" scope="prototype"/>
```

Summary Table

Scope	**Description**	**Lifecycle**	**Example Use Case**
singleton	One instance per Spring container	Shared globally in app	Services, repositories
prototype	New instance every request	Created on each injection/request	State-full beans, not shared
request	One per HTTP request (web only)	Lives for one HTTP request	Request-scoped beans
session	One per HTTP session (web only)	Lives for one user session	User-specific data storage

---

==

AUTOWIRING:

Great! I’ll give you **simple runnable Spring Boot projects** for **each type of autowiring** (No autowiring, byName, byType, constructor, and annotation-based) with explanations.

Each example will be minimal and easy to run in Spring Tool Suite (STS) or any IDE.

1️⃣Project: No Autowiring (Manual Wiring via XML)

Files:

---

src/main/resources/applicationContext.xml

```
<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="student" class="com.example.noautowire.Student" />

    <bean id="college" class="com.example.noautowire.College">
        <property name="student" ref="student" />
    </bean>
</beans>
```

Student.java

```
package com.example.noautowire;
```

```
public class Student {
    public void show() {
        System.out.println("Hello from Student (No Autowiring)");
    }
}
```

College.java

```
package com.example.noautowire;

public class College {
    private Student student;

    public void setStudent(Student student) {
        this.student = student;
    }

    public void show() {
        student.show();
    }
}
```

MainApp.java

```
package com.example.noautowire;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        College college = (College) context.getBean("college");
        college.show();
        ((ClassPathXmlApplicationContext)context).close();
    }
}
```

2□Project: Autowiring by Name (XML)

Change XML college bean to:

```
<bean id="college" class="com.example.byname.College" autowire="byName"
/>
<bean id="student" class="com.example.byname.Student" />
```

Java classes same as above but package com.example.byname.

3□Project: Autowiring by Type (XML)

Change XML college bean to:

```
<bean id="college" class="com.example.bytype.College" autowire="byType" />
<bean id="student" class="com.example.bytype.Student" />
```

Java classes same as above but package com.example.bytype.

4□Project: Autowiring by Constructor (XML)

Change College.java:

```java
package com.example.constructor;

public class College {
   private Student student;

   public College(Student student) {
      this.student = student;
   }

   public void show() {
      student.show();
   }
}
```

Change XML:

```xml
<bean id="college" class="com.example.constructor.College" autowire="constructor" />
<bean id="student" class="com.example.constructor.Student" />
```

5️⃣ Project: Annotation-Based Autowiring

Student.java

```java
package com.example.annotation;

import org.springframework.stereotype.Component;

@Component
```

```
public class Student {
    public void show() {
        System.out.println("Hello from Student (Annotation Autowiring)");
    }
}
```

College.java

```
package com.example.annotation;

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

@Component
public class College {

    @Autowired
    private Student student;

    public void show() {
        student.show();
    }
}
```

MainApp.java

```
package com.example.annotation;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;


@Configuration
@ComponentScan("com.example.annotation")
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(MainApp.class);
      College college = context.getBean(College.class);
      college.show();
   }
}
```

How to run?

- Create separate projects or packages for each example.
- For XML-based examples: place applicationContext.xml in src/main/resources.
- Run the MainApp.java class.
- Observe console output showing injected bean behavior.

==

AOP AND SPRING JDBC TEMPLATE:

Sure! Here's a clear, beginner-friendly explanation with simple examples of both **AOP (Aspect-Oriented Programming)** and **Spring JDBC Template**.

1AOP (Aspect-Oriented Programming) in Spring

What is AOP?

- AOP lets you **separate cross-cutting concerns** (like logging, security, transactions) from your main business logic.
- It helps you apply behaviors **across multiple points** in your application without cluttering core code.
- Core concepts:
  - **Aspect:** Modularization of concern (e.g., logging).
  - **Join Point:** A point in execution (e.g., method call).
  - **Advice:** Code to execute at join points (before, after, around).
  - **Pointcut:** Expression to select join points.

Simple Example: Logging Before Method Execution

Step 1: Add Spring AOP dependencies (in Maven):

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.3.25</version>
</dependency>
```

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-aop</artifactId>
   <version>5.3.25</version>
</dependency>
<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjweaver</artifactId>
   <version>1.9.9</version>
</dependency>
```

Step 2: Create Business Class (BankService.java)

```
package com.example.aop;

import org.springframework.stereotype.Component;

@Component
public class BankService {
   public void withdraw() {
      System.out.println("Withdraw method called");
   }
}
```

Step 3: Create Aspect Class (LoggingAspect.java)

```
package com.example.aop;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.aop.BankService.withdraw(..))")
    public void logBeforeWithdraw() {
        System.out.println("Logging before withdraw");
    }
}
```

Step 4: Spring Config and Main App

```
package com.example.aop;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.example.aop")
@EnableAspectJAutoProxy  // Enable Spring AOP
```

```
public class AppConfig {}


public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        BankService bankService = context.getBean(BankService.class);
        bankService.withdraw();
    }
}
```

Output:

```
Logging before withdraw
Withdraw method called
```

2️⃣ Spring JDBC Template

What is Spring JDBC Template?

- Simplifies working with JDBC (Java Database Connectivity).
- Helps you avoid boilerplate code (like connection, statement, resultset handling).
- Supports query execution, updates, and transaction management.

Simple Example: Querying Data from a Database

Step 1: Add dependency (Maven):

```
<dependency>
```

```xml
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.25</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
</dependency>
```

Step 2: Configure DataSource and JdbcTemplate

```java
package com.example.jdbctemplate;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("org.h2.Driver");
        ds.setUrl("jdbc:h2:mem:testdb");
        ds.setUsername("sa");
        ds.setPassword("");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource ds) {
        return new JdbcTemplate(ds);
    }
}
```

Step 3: Create DAO class

```
package com.example.jdbctemplate;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Component
public class StudentDAO {
    private final JdbcTemplate jdbcTemplate;

    public StudentDAO(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        jdbcTemplate.execute("CREATE TABLE student(id INT PRIMARY KEY, name
VARCHAR(50))");
        jdbcTemplate.update("INSERT INTO student VALUES (?, ?)", 1, "John Doe");
    }

    public List<Student> getAllStudents() {
        return jdbcTemplate.query("SELECT * FROM student", new
RowMapper<Student>() {
            @Override
            public Student mapRow(ResultSet rs, int rowNum) throws SQLException
{
                return new Student(rs.getInt("id"), rs.getString("name"));
            }
        });
    }
}
```

Step 4: Student model

```
package com.example.jdbctemplate;

public class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name + "'}";
    }
}
```

Step 5: Main Application

```
package com.example.jdbctemplate;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(JdbcConfig.class, StudentDAO.class);

        StudentDAO dao = context.getBean(StudentDAO.class);
        dao.getAllStudents().forEach(System.out::println);
    }
}
```

Output:

```
Student{id=1, name='John Doe'}
```

Summary

Topic	What it Does	When to Use
AOP	Separates cross-cutting concerns (e.g., logging)	When you want to add behavior around method calls without changing business logic
Spring JDBC Template	Simplifies JDBC operations with less boilerplate	When you want easy database access without full ORM

==

Frequently used Annotations:

1. @Bean

What it does:
Marks a method inside a @Configuration class. The method returns an object that Spring will register as a bean.

Example:

java

CopyEdit

```
@Configuration
public class AppConfig {
   @Bean
   public String greeting() {
      return "Hello, Spring!";
   }
}
```

Explanation:

- Spring runs the greeting() method and registers "Hello, Spring!" as a bean named greeting.
- You can inject this string bean elsewhere.

---

2. @Autowired

What it does:
Automatically injects dependencies by type.

Example:

java

CopyEdit

```
@Component
public class MyService {
   public void serve() {
      System.out.println("Serving...");
   }
}


@Component
public class MyController {
   @Autowired
   private MyService myService;

   public void doWork() {
      myService.serve();
   }
}
```

Explanation:

- Spring injects MyService into MyController automatically.
- MyController can call myService.serve() without creating it.

---

3. @Primary

What it does:
Marks a bean as the default when multiple beans of the same type exist.

Example:

java

CopyEdit

```
@Component
@Primary
public class PrimaryService implements MyService {}

@Component
public class SecondaryService implements MyService {}

@Component
public class Client {
    @Autowired
```

```
   private MyService service;  // Injects PrimaryService by default
}
```

Explanation:

- Spring injects PrimaryService into Client because of @Primary, avoiding ambiguity.

---

4. @Qualifier

What it does:
Specifies which bean to inject when multiple candidates exist.

Example:

java

CopyEdit

```
@Component("serviceOne")
public class ServiceOne implements MyService {}

@Component("serviceTwo")
public class ServiceTwo implements MyService {}

@Component
public class Client {
   @Autowired
   @Qualifier("serviceTwo")
   private MyService service;  // Injects ServiceTwo explicitly
}
```

Explanation:

- @Qualifier tells Spring to inject the bean named serviceTwo.

---

5. @Configuration

What it does:
Marks a class that defines beans using @Bean methods.

Example:

java

CopyEdit

```
@Configuration
public class AppConfig {
   @Bean
   public String example() {
      return "Example";
   }
}
```

Explanation:

- Spring treats AppConfig as a source of bean definitions.

---

6. @ComponentScan

What it does:
Tells Spring where to scan for components (@Component, @Service, etc.).

Example:

java

CopyEdit

```
@Configuration
@ComponentScan("com.example.services")
public class AppConfig {}
```

Explanation:

- Spring scans com.example.services package and registers annotated beans automatically.

---

7. @Scope

What it does:
Defines the bean scope, like singleton or prototype.

Example:

java

CopyEdit

```
@Component

@Scope("prototype")

public class PrototypeBean {}
```

Explanation:

- A new instance of PrototypeBean is created every time it's requested.

---

8. @Required

What it does:
Marks a setter method as required to be set by Spring.

Example:

java

CopyEdit

```
@Component

public class RequiredExample {

  private String name;


  @Required

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

Explanation:

- Spring throws an error if name is not injected via the setter.

---

9. @Value

What it does:
Injects literal or property values into fields.

Example:

java

CopyEdit

```
@Component
public class MyBean {
    @Value("${app.name}")
    private String appName;

    @Value("42")
    private int number;
}
```

Explanation:

- Injects app.name property value and a literal number 42.

---

10. @PropertySource

What it does:
Loads a properties file for use with @Value.

Example:

java

CopyEdit

```
@Configuration

@PropertySource("classpath:application.properties")

public class AppConfig {}
```

Explanation:

- Loads application.properties so @Value can use its values.

---

11. @Lazy

What it does:
Delays bean creation until it's actually needed.

Example:

java

CopyEdit

```
@Component

@Lazy

public class LazyBean {}
```

Explanation:

- LazyBean is created only when first requested, not at startup.

---

12. @Import

What it does:
Imports other configuration classes.

Example:

java

CopyEdit

```
@Configuration
```

```
@Import(AnotherConfig.class)

public class MainConfig {}
```

Explanation:

- Beans in AnotherConfig are included in Spring context.

---

13. @ImportResource

What it does:
Imports XML config files into Java config.

Example:

java

CopyEdit

```
@Configuration

@ImportResource("classpath:beans.xml")

public class AppConfig {}
```

Explanation:

- Loads beans from beans.xml into Spring context.

---

14. @Profile

What it does:
Activates beans only for certain profiles (e.g., dev, prod).

Example:

java

CopyEdit

```
@Component

@Profile("dev")

public class DevBean {}
```

Explanation:

- DevBean is created only when the dev profile is active.