1. Spring Core Basics

📌 What is Spring Core?

Spring Core is the foundation of the entire Spring Framework. It provides the fundamental features of the Spring container, primarily **Dependency Injection (DI)** and **Inversion of Control (IoC)**, which manage the life cycle and configuration of application objects (also called *beans*).

📌 Why use Spring Core?

Before Spring, developers used to create and manage objects manually using the new keyword, making the code tightly coupled and difficult to manage, test, or reuse.

Spring Core solves this by:

- Creating objects automatically (IoC)
- Injecting dependencies automatically (DI)
- Managing object lifecycle and configuration
- Supporting loose coupling and better testability
- Providing integration with other technologies (JDBC, JPA, etc.)

📌 Key Concepts in Spring Core:

- **Bean**: An object that is managed by the Spring IoC container.
- **IoC Container**: The core container that creates, configures, and manages beans.
- **ApplicationContext**: Advanced IoC container used to get beans.

📌 Example: Basic Spring App using XML

1. POJO (Plain Old Java Object)

```
public class Hello {
    public void sayHi() {
        System.out.println("Hello from Spring!");
    }
}
```

2. Configuration (beans.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">

    //here you can use any bean id of your choice
    <bean id="helloBean" class="com.example.Hello"/>
</beans>
```

3. Main class to run

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
Hello h = (Hello) context.getBean("helloBean");
h.sayHi();
```

Spring Core helps us manage objects smartly using IoC and DI. Instead of manually writing object creation code everywhere, we let Spring do that. This makes our code **clean, modular, and testable**.

2. Dependency Injection (DI)

📌 What is Dependency Injection?

Dependency Injection is a design pattern used in Spring where one object supplies the dependencies (objects it needs) of another object.

In simple terms, instead of a class creating its own dependencies using `new`, they are **"injected"** from the outside—by the Spring container.

A *dependency* is just another object your class needs to work. For example, a `Car` needs an `Engine`. So `Engine` is a dependency of `Car`.

📌 Why use Dependency Injection?

Without DI:

- You manually create objects (`new Engine()` inside `Car`).
- Your classes become tightly coupled.
- Testing becomes harder (you cannot easily replace the `Engine` with a mock).

With DI:

- Spring provides and manages the `Engine`.
- `Car` doesn't care how the `Engine` is created.
- Code is loosely coupled, easier to test and maintain.

📌 Types of Dependency Injection in Spring:

1. **Constructor Injection** – dependencies are passed via constructor.
2. **Setter Injection** – dependencies are set using setter methods.
3. **Field Injection** – dependencies are injected directly into fields using annotations (not recommended for complex systems, but convenient).

📌 Example 1: Constructor Injection (XML)

1. Engine Class

```
public class Engine {
    public void start() {
        System.out.println("Engine started.");
    }
}
```

2. Car Class

```
public class Car {
    private Engine engine;

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

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

3. beans.xml Configuration

```
<bean id="engine" class="com.example.Engine"/>
<bean id="car" class="com.example.Car">
    <constructor-arg ref="engine"/>
</bean>
```

4. Main class

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
Car car = (Car) context.getBean("car");
car.drive();
```

📌 Example 2: Setter Injection (Annotation)

Car.java

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is running via setter DI...");
    }
}
```

Engine.java

```
@Component
public class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}
```

Main class (with component scanning)

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

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Car car = context.getBean(Car.class);
        car.drive();
    }
}
```

Dependency Injection is like **giving objects from outside** instead of creating them yourself.

It reduces **tight coupling**, makes your app **flexible**, and helps in **unit testing**.

In Spring, DI is done automatically using **constructor, setter, or field injection**, either by XML or annotations.

3. Inversion of Control (IoC)

📌 What is IoC?

Inversion of Control (IoC) is a principle in which the control of creating objects and managing their lifecycle is shifted from the programmer to the **Spring container**.

Normally in Java, you write code like this:

```
Car car = new Car(new Engine());
```

Here, you are controlling the creation of `Car` and also supplying its dependency `Engine`.

With **IoC**, you don't create objects using `new`. Instead, Spring creates them for you, injects their dependencies, and manages them.

📌 Why use IoC?

- Helps in creating **loosely coupled** systems.
- Developers focus on business logic — Spring handles object creation.
- Promotes **modular, testable, and maintainable** code.
- Makes unit testing easy because you can plug in mock dependencies.

📌 How IoC Works in Spring

1. Spring reads the configuration (XML or annotations).
2. It creates and assembles the objects (called **beans**).
3. It injects their dependencies.
4. You simply **ask Spring** for the object when needed using `getBean()`.

The container responsible for doing this is called the **IoC Container**. The main implementation is:

- `ApplicationContext` – commonly used container in real-world applications.

📌 Analogy:

Think of Spring IoC like a **coffee shop**. You don't go to the kitchen and make coffee. You just **ask for coffee**, and the shop gives it to you, already prepared.

📌 Example Using IoC

Engine.java

```
public class Engine {
    public void start() {
        System.out.println("Engine is starting...");
    }
}
```

Car.java

```
public class Car {
    private Engine engine;

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

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

beans.xml

```
<bean id="engine" class="com.example.Engine"/>
<bean id="car" class="com.example.Car">
    <constructor-arg ref="engine"/>
</bean>
```

Main.java

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

```
Car car = (Car) context.getBean("car");
car.move();
```

Here, the `Car` and `Engine` objects are not created by us — Spring creates them, **controls them**, and injects dependencies using **IoC**.

IoC = giving up control of object creation and wiring to Spring.

It’s the foundation of Spring Framework.

Enables **Dependency Injection**, making applications more **decoupled and testable**.

Think of Spring as a **factory or container** that gives you ready-to-use objects (beans).

4. Bean Scopes in Spring

📌 What is a Bean Scope?

In Spring, a **bean** is just a Java object managed by the Spring container.
A **bean scope** defines **how many instances** of a bean are created and **how long they live** inside the container.

Spring provides **different scopes** to control the lifecycle and visibility of beans — especially useful when building web apps, REST APIs, or desktop apps.

📌 Why are Bean Scopes Important?

- They help **optimize memory** and **control behavior** based on application needs.
- You can define whether a bean should be:
    - **Shared** (single instance)
    - **Created new** for each use
    - **Tied to a web request/session**, etc.

📌 Types of Bean Scopes

Scope	Description
`singleton`	(Default) A single shared instance is created and reused for the container.
`prototype`	A **new instance** is created **each time** the bean is requested.
`request`	**A new bean is created for every HTTP request (web apps only).**

`session`	**A new bean is created per HTTP session.**
`application`	**One bean for the entire lifecycle of a web application.**
`websocket`	**One bean instance per WebSocket connection.**

1. Singleton (Default Scope)

📌 **What is it?**

- Spring creates **one object** for the bean and **shares it** wherever needed.
- Only **one object** is created per Spring container.
- No matter how many times you request the bean, you get the **same instance**.

📌 **Why use it?**

- Saves memory by **reusing the same object**.
- Best for **stateless** services (e.g., a service class that just processes data).
- Fast and efficient for shared components.

```java
@Component
@Scope("singleton") // Optional, as it's the default
public class Printer {
    public Printer() {
        System.out.println("Printer object created");
    }
}
```

Behavior: Even if you call `getBean()` 10 times, you get the **same object**.

---

2. **Prototype**

📌 **What is it?**

- Spring creates **a new object every time** you request the bean.

📌 **Why use it?**

- Needed when:
  - You need **fresh objects** each time (like `new` keyword behavior).
  - The bean holds **state/data specific** to one task.
- Useful in **non-shared** or **mutable** scenarios.

```
@Component
@Scope("prototype")
public class User {
    public User() {
        System.out.println("New User object created");
    }
}
```

Behavior: If you call `getBean()` 3 times, you get **3 different objects**.

3. **Request(Web Only)**

📌 **What is it?**

- A new object is created for **each HTTP request**.

📌 **Why use it?**

- Useful when you want to bind bean lifecycle to a single **HTTP request**.
- Example: a form submission or login attempt.

```
@Component
@Scope("request")
public class RequestScopedBean {
    public RequestScopedBean() {
        System.out.println("Request Bean Created");
    }
}
```

---

4. **Session (Web Only)**

📌 **What is it?**

- One bean is created per **HTTP session** (i.e., per logged-in user).

📌 **Why use it?**

- Helpful when you want to store **user-specific** information like shopping cart or profile.

```
@Component
@Scope("session")
public class SessionScopedBean {
    public SessionScopedBean() {
        System.out.println("Session Bean Created");
    }
}
```

5. **Application (Web Only)**

📌 **What is it?**

- One object is created and **shared across the entire web application**.

📌 **Why use it?**

- Used for **global data/configuration** like site settings or constants.

```
@Component
@Scope("application")
public class AppScopedBean {
    public AppScopedBean() {
        System.out.println("Application Bean Created");
    }
}
```

6. **WebSockets (WebSocket Application Only)**

📌 **What is it?**

- One object is created per **WebSocket session**.

📌 **Why use it?**

- Perfect for **real-time applications** like chat apps, where each socket connection needs a separate instance.

```
@Component
@Scope("websocket")
public class ChatScopedBean {
    public ChatScopedBean() {
        System.out.println("WebSocket Bean Created");
    }
}
```

- Use `singleton` for **common services** (e.g., calculations, config).
- Use `prototype` when each object must be **different** (e.g., forms, tasks).
- Both are defined using `@Scope("...")` with `@Component`.

5. Autowiring in Spring

📌 **What is it?**

- **Autowiring** is a feature in Spring where the container **automatically injects dependencies** into a bean without the need for manual `setters` or configuration.
- It removes the need to explicitly write the code to inject one bean into another.

📌 **Why use it?**

- **Reduces boilerplate code** — no need to manually wire beans.
- Promotes **loose coupling** and cleaner code.
- Makes dependency management more **automatic and consistent**.

`@Autowired` Annotation

Spring provides the `@Autowired` annotation to enable autowiring. It can be used:

- On **fields**
- On **constructors**

Autowiring by Field

```
@Component
public class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}
@Component
public class Car {

    @Autowired  // Spring automatically injects the Engine bean here
    private Engine engine;

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

Autowiring by Constructor

```
@Component
public class Car {

    private Engine engine;

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
    public void move() {
        engine.start();
        System.out.println("Car is moving...");
    }
}
```

- `@Autowired` works on **fields, constructors, and setters**.
- Spring uses **type matching** to inject the dependency.
- Use `@Qualifier` when more than one bean of the same type exists.
- You can make autowiring optional using:
  `@Autowired(required = false)`

6. Important Spring Annotations

🔹 **@Component**

- Marks a class as a **Spring-managed bean**.
- Detected automatically during component scanning.

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

🔹 **@Service, @Repository, @Controller**

- Same as @Component, but with **specific roles**:
    - @Service → Business logic
    - @Repository → Database access
    - @Controller → Web controller

```
@Service
public class OrderService { }

@Repository
public class OrderRepository { }

@Controller
public class OrderController { }
```

🔹 @Autowired

- **Injects** dependencies automatically by type.

```
@Component
public class Car {

    @Autowired
    private Engine engine;

    public void drive() {
        engine.start();
    }
}
```

🔹 @Qualifier("beanName")

- Helps choose the correct bean **when multiple types exist**.

```
@Component("dieselEngine")
public class DieselEngine implements Engine { }

@Component
public class Car {

    @Autowired
    @Qualifier("dieselEngine")
    private Engine engine;
}
```

🔹 @Configuration

- Marks a class that **defines beans manually** using @Bean.

🔹 @Bean

- Used inside a @Configuration class to **register a bean manually**.

```java
@Configuration
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine();
    }
}
```

🔹 @ComponentScan

- Tells Spring **which packages** to scan for beans.

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

7. Life Cycle Callbacks in Spring

📌 What is it?

Spring allows you to hook into the **lifecycle of a bean** – you can run custom logic:

- **Right after the bean is created**
- **Right before the bean is destroyed**

This is useful for resource management, logging, opening/closing connections, etc.

📌 Why we use it?

- To **initialize** resources (e.g., database connections, caches) when the bean is created.
- To **release resources** (e.g., close file streams, stop services) when the bean is destroyed.
- Clean and structured setup/teardown logic for beans.

---

1. Ways to Handle Life Cycle Callbacks

🔹 1. Using @PostConstruct and @PreDestroy (Recommended)

These are **Java standard annotations**, supported by Spring.

Example:

```
@Component
public class MyBean {

    @PostConstruct
    public void init() {
```

```
        System.out.println("Bean initialized");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Bean about to be destroyed");
    }
}
```

🔹 2. Implementing InitializingBean and DisposableBean Interfaces

Spring-specific interfaces.

Example:

```
@Component
public class MyBean implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() {
        System.out.println("Bean initialized via InitializingBean");
    }

    @Override
    public void destroy() {
        System.out.println("Bean destroyed via DisposableBean");
    }
}
```

🔹 3. Using @Bean(initMethod = "", destroyMethod = "")

If you're defining a bean manually in a @Configuration class.

Example:

```java
@Configuration
public class AppConfig {

    @Bean(initMethod = "start", destroyMethod = "stop")
    public MyBean myBean() {
        return new MyBean();
    }
}
```

```java
public class MyBean {
    public void start() {
        System.out.println("Custom init method");
    }

    public void stop() {
        System.out.println("Custom destroy method");
    }
}
```

8. Bean Configuration Styles in Spring

📌 What is it?

Spring allows you to **configure and register beans** in **three main styles**:

1. **XML-based Configuration**
2. **Annotation-based Configuration**
3. **Java-based Configuration (@Configuration class)**

Each style tells Spring how to **create**, **configure**, and **inject** beans.

---

1. **XML-Based Configuration (Old Style)**
    - Configuration is done in an XML file (applicationContext.xml).
    - You define beans manually using <bean> tags.

Example:

```
<beans>
    <bean id="myBean" class="com.example.MyBean" />
</beans>
```

Load XML config:

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");
MyBean bean = context.getBean("myBean", MyBean.class);
```

2. **Annotation-Based Configuration (Modern, Recommended)**
    - Uses annotations like @Component, @Service, @Autowired, etc.
    - Requires @ComponentScan to detect components.

Example:

```
@Component
public class MyBean { }

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

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(AppConfig.class);
MyBean bean = context.getBean(MyBean.class);
```

---

3. Java-Based Configuration (@Bean and @Configuration)

- Manually define beans in a configuration class using @Bean.
- Full control, no XML needed.

Example:

```
@Configuration
public class AppConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(AppConfig.class);
MyBean bean = context.getBean(MyBean.class);
```