Spring Beans, IoC, and Dependency Injection — Complete Detailed Note (Beginner Friendly + Deep Understanding)

This document explains **everything** about:

- **Spring Beans**
- **IoC (Inversion of Control)**
- **Dependency Injection (DI)**
- **How Spring manages objects**
- **How wiring actually happens internally**

With simple language + diagrams + examples.

⭐1. What Is a Bean in Spring?

A **Bean** is simply **an object created, managed, and destroyed by the Spring IoC Container**.

> Bean = Object created by Spring instead of `new` keyword

Example (Without Spring):

```
Laptop laptop = new Laptop();    // You create object
```

Example (With Spring):

Spring creates the object, not you.

```
@Component
class Laptop {}
```

Then you get it from Spring:

```
@Autowired
Laptop laptop;    // Spring injects object
```

✔️A bean is **just a normal class** marked for Spring management using:

- `@Component`
- `@Service`
- `@Repository`

- `@Controller`
- `@Configuration`
- `@Bean`

⭐2. What Is IoC (Inversion of Control)?

IoC means:

> **You give control to Spring. Spring creates objects, gives you dependencies, and manages their lifecycle.**

Without IoC (normal Java):

You control everything.

```
Student s = new Student();  // You create
Laptop l = new Laptop();    // You create
s.setLaptop(l);             // You wire
```

With IoC:

You hand over control.

```
@Autowired
Laptop laptop;  // Spring injects
```

IoC is the **big idea**.
DI is the **way it is implemented**.

⭐3. What Is Dependency Injection (DI)?

DI means:

> **Spring injects (gives) required objects instead of you creating them manually.**

Example:

```
class Student {
    private Laptop laptop;

    public Student(Laptop laptop) {  // Dependency injected
        this.laptop = laptop;
    }
}
```

DI ≠ creating objects
DI = giving ready-made objects

⭐4. How Spring Creates Beans (Step-by-Step Internal Understanding)

When the application starts:

1️⃣Spring scans packages for `@Component`, `@Service`, etc.
2️⃣Creates 1 object of each bean (singleton by default)
3️⃣Stores them in **ApplicationContext** (Spring Container)
4️⃣When another class needs that bean → Spring injects it

⭐5. DI Types (Important!)

Spring supports 3 types:

🦝5.1 Constructor Injection (BEST)

Constructor Injection is the **most recommended DI method in Spring Core**, and is considered the cleanest, safest, and most test-friendly way of injecting dependencies.

🌥️What is Constructor Injection?

Spring calls a class's constructor and **passes the required dependency objects** into it.

You do **not** create objects yourself. You do **not** call setters. Spring handles everything.

🔥Why Constructor Injection is Best (Spring Core reasoning)

- ✔️Makes dependencies **required** → object cannot be created without them
- ✔️Supports **immutability** (final fields)
- ✔️Great for **unit testing** (via constructor mocking)
- ✔️Avoids partially-initialized objects
- ✔️Works **without @Autowired** when a class has only one constructor
- ✔️Prevents field-level circular dependency issues

🦜 Constructor Injection Example in Spring Core (XML Based)

Laptop.java

```
public class Laptop {
    public void compile() {
        System.out.println("Laptop compiling...");
    }
}
```

Student.java

```
public class Student {

    private Laptop laptop; // dependency

    public Student(Laptop laptop) {   // Constructor Injection
        this.laptop = laptop;
    }

    public void code() {
        laptop.compile();
    }
}
```

spring.xml (constructor-arg)

```
<bean id="laptop" class="org.example.Laptop"/>

<bean id="student1" class="org.example.Student">
    <constructor-arg ref="laptop" />
</bean>
```

Main.java

```
ApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
Student student = (Student) context.getBean("student1");
student.code();
```

✔️ Spring creates Laptop → creates Student(laptop) → injects → returns ready object.

🦜Constructor Injection in Spring Core (Annotation Based)

Laptop.java

```
@Component
public class Laptop {
    public void compile() {
        System.out.println("Compiling...");
    }
}
```

Student.java

```
@Component
public class Student {

    private final Laptop laptop;  // immutable dependency

    @Autowired     // optional if only ONE constructor
    public Student(Laptop laptop) {
        this.laptop = laptop;
    }

    public void code() {
        laptop.compile();
    }
}
```

Why @Autowired is optional here?

Spring sees only 1 constructor → assumes that is the injection constructor.

⚠️ If you have multiple constructors

You MUST tell Spring which one to use:

```
@Autowired
public Student(Laptop laptop) {
    this.laptop = laptop;
}
```

Otherwise Spring throws:

```
NoSuchBeanDefinitionException
or
UnsatisfiedDependencyException
```

⛰️Constructor Injection vs Setter Injection vs Field Injection

Feature	Constructor DI	Setter DI	Field DI
Recommended?	🔲Best	Good for optional	❌Not recommended
Dependency required?	✔️Yes	Optional	Optional
Immutability	✔️Strong	❌No	❌No
Testing	✔️Best	Good	Worst
Circular dependency handling	✔️Best	Good	Worst

💐Quick Example Showing Why Constructor DI Is Better

```
public class Student {

    private Laptop laptop;

    public Student() {}         // Object can be created half-initialized

    @Autowired
    public void setLaptop(Laptop laptop) {   // Optional injection
        this.laptop = laptop;
    }
}
```

If setLaptop() is not called → laptop is null → NullPointerException.

Constructor DI prevents this:

```
public Student(Laptop laptop) {   // Cannot create object without dependency
    this.laptop = laptop;
}
```

✔️Always safe ✔️No null dependencies

🐁Summary of Constructor Injection

- Mandatory dependencies → Constructor
- Optional dependencies → Setter
- Avoid field injection except for simple demos

This is the DI method used in **enterprise, production, microservices, and Spring Boot apps** because it is clean and highly reliable.

```
@Component
class Student {
    private final Laptop laptop;

    public Student(Laptop laptop) {   // Inject here
        this.laptop = laptop;
    }
}
```

✔️Most recommended
✔️Immutable dependencies
✔️Works best with testing
✔️Auto-wired without annotation if only 1 constructor

🦝5.2 Setter Injection

```
@Component
class Student {
    private Laptop laptop;

    @Autowired
    public void setLaptop(Laptop laptop) {  // Inject here
        this.laptop = laptop;
    }
}
```

Used when:

- Dependency is optional
- Dependency can change at runtime

🦝5.3 Field Injection (Not recommended)

```
@Component
class Student {
    @Autowired
    private Laptop laptop;  // Inject directly
}
```

Problems:

- Hard to test
- No immutability
- Cannot see dependencies clearly

Still works and is easy for beginners.

⭐6. Why Constructor DI Doesn't Need @Autowired

If your class has **only 1 constructor**, Spring assumes it is for DI.

```
@Component
class Driver {
    private Car car;

    public Driver(Car car) {  // No @Autowired needed
        this.car = car;
    }
}
```

If you have multiple constructors → Spring gets confused.

Then you MUST add `@Autowired` to tell Spring which constructor to use.

⭐7. Understanding Upcasting in DI (Very Important!)

DI becomes powerful when combined with **polymorphism**.

```
class Car { void start() {} }
class ElectricCar extends Car { void start() {} }
```

```
@Component
class Driver {
    private Car car;

    public Driver(Car car) {  // Parent type
        this.car = car;
    }
}
```

Because Spring injects by **type**, it can give:

- Car
- ElectricCar
- DieselCar
- SportsCar

✔️This is why DI + polymorphism makes applications flexible.

⭐8. What If We Have Multiple Beans of Same Type?

Example:

```
@Component
class Car {}
@Component
class ElectricCar extends Car {}
```

Spring becomes confused:

> Which Car should I inject?

Solution 1: `@Primary`

```
@Primary
@Component
class Car {}
```

Solution 2: `@Qualifier`

```
@Component("electric")
class ElectricCar {}
```

```
@Autowired
@Qualifier("electric")
Car car;
```

⭐9. Lifecycle of a Bean (Beginner Friendly)

Spring controls:

- Creation
- Initialization
- Destruction

```
Create → Populate Dependencies → Initialize → Use → Destroy
```

You can hook into lifecycle using:

- `@PostConstruct`

```
@PostConstruct
public void init() {}
```

- `@PreDestroy`

```
@PreDestroy
public void cleanup() {}
```

⭐10. IoC Container Types

Before understanding container *types*, you must understand one important fact:

🦜The IoC Container in modern Spring = ApplicationContext

Spring has two IoC containers, but in real-world applications, **ApplicationContext is the actual IoC container used**.

🦝What does IoC Container mean?

> It is the part of Spring that **creates beans, manages their lifecycle, and injects dependencies**.

🦝 Why ApplicationContext is the real IoC container?

Because it provides:

- Bean creation & wiring
- Dependency injection
- AOP support
- Event publishing
- Internationalization
- Auto-detection of beans

This is why Spring Boot uses ApplicationContext internally when you write:

```
ApplicationContext context = SpringApplication.run(App.class, args);
```

Spring has 2 main containers:

1. BeanFactory (basic)

- Lazy loading
- Lightweight

2. ApplicationContext (advanced)

- Eager loading
- AOP support
- Event handling

Most apps use ApplicationContext.

⭐ 11. Scopes of Beans

Spring beans have different lifecycles:

Default: Singleton

One object for the whole application.

Prototype

New object every time you ask.

```
@Scope("prototype")
```

Web scopes:

- request
- session
- application

⭐12. Bean vs Object

Normal Java Object	Spring Bean
Created with `new`	Created by Spring
You manage lifecycle	Spring manages lifecycle
No auto DI	Fully auto-wired
No container	Managed by IoC container

⭐12. Autowiring (Very Important)

Autowiring means **Spring automatically finds the correct bean and injects it into your class** without you writing boilerplate code.

It works based on: - Type - Qualifier - Primary bean

🌥️12.1 What Is Autowiring?

Autowiring tells Spring:

> "Find a bean of this type and inject it here."

Example:

```
@Autowired
private Laptop laptop;
```

Spring searches its IoC container: - Is there a bean of type Laptop? - Yes → inject it

---

🌥️12.2 How Autowiring Works Internally

1. Spring scans classes (`@ComponentScan`)
2. Creates bean objects
3. Stores them inside ApplicationContext
4. When it sees `@Autowired`, it performs **dependency resolution**:

Step 1 — By Type

Spring checks bean type:

```
@Autowired
Laptop laptop;
```

Does Spring have a `Laptop` bean? → Injects it.

Step 2 — If there are multiple beans of same type

Spring becomes confused.

Example:

```
@Component class Laptop {}
@Component class GamingLaptop extends Laptop {}
```

Now `@Autowired Laptop laptop;` → ❌Confusion

Spring fails with:

```
NoUniqueBeanDefinitionException
```

🌥️12.3 Fixing Conflicts (IMPORTANT)

🦝Method 1: @Primary

Mark the preferred bean.

```
@Primary
@Component
class Laptop {}
```

🦝Method 2: @Qualifier

Tell Spring exactly which bean you want.

```
@Component("gaming")
class GamingLaptop extends Laptop {}
```

```
@Autowired
@Qualifier("gaming")
Laptop laptop;
```

⛅12.4 Autowiring Types (XML Based)

In Spring Core XML, you can configure autowiring using:

```
<bean id="student" class="org.example.Student" autowire="byType" />
```

Types:

- **byType** → inject bean by matching type
- **byName** → match setter name = bean id
- **constructor** → use constructor for DI
- **no** → default (no autowiring)

Example:

```
<bean id="laptop" class="org.example.Laptop" />

<bean id="student" class="org.example.Student" autowire="constructor" />
```

⛅12.5 When @Autowired Is Not Needed

If a class has **only one constructor**, Spring injects automatically.

```
@Component
class Student {
    private Laptop laptop;

    public Student(Laptop laptop) { } // No @Autowired needed
}
```

⛅12.6 Field, Setter, Constructor Autowiring

🦝Field Injection

```
@Autowired
private Laptop laptop;
```

Simple but not recommended.

🦝Setter Injection

```
@Autowired
public void setLaptop(Laptop laptop) {
    this.laptop = laptop;
}
```

🦝Constructor Injection (Best)

```
@Autowired
public Student(Laptop laptop) {
    this.laptop = laptop;
}
```

Or without annotation when only one constructor exists.

🌥️12.7 Why Autowiring Exists

Autowiring reduces boilerplate:

Without Autowiring:

```
<property name="laptop" ref="laptop" />
```

With Autowiring:

```
@Autowired
Laptop laptop;
```

Spring simply figures it out automatically.

⛅12.8 Summary of Autowiring

Feature	Meaning
@Autowired	Automatically inject bean by type
@Qualifier	Tell Spring which specific bean to inject
@Primary	Mark default bean when multiple exist
XML autowire	byType, byName, constructor
Not needed for single-constructor classes	Spring automatically injects

---

⭐12.5 Autowiring in Spring Core Using XML

Spring Core supports **autowiring without annotations**, using only XML. This is useful when working in legacy Spring applications or when annotations are not allowed.

Spring provides **5 autowire modes** in XML:

Mode	Meaning
no (default)	No autowiring, you must manually use `<property>`
byName	Injects bean whose name matches the property name
byType	Injects bean whose type matches the property type
constructor	Injects using constructor, matching by type
autodetect	(Deprecated) Chooses constructor or byType

⛅1. Autowiring byName (XML)

Works when **property name = bean id**.

Student.java

```
public class Student {
    private Laptop laptop;

    public void setLaptop(Laptop laptop) {
        this.laptop = laptop;
    }

    public void code() {
        laptop.compile();
    }
}
```

spring.xml

```
<bean id="laptop" class="org.example.Laptop" />

<bean id="student" class="org.example.Student" autowire="byName" />
```

✔️ Spring sees `setLaptop()` → property name = `laptop` ✔️ Finds bean with id= `laptop` ✔️ Injects it automatically

---

🌥️2. Autowiring byType (XML)

Spring injects dependency **based on matching type**, not name.

```
<bean id="laptop1" class="org.example.Laptop" />

<bean id="student" class="org.example.Student" autowire="byType" />
```

✔️If only ONE bean of type Laptop exists → injected ❌If multiple Laptop beans exist → Spring throws exception

---

🌥️3. Constructor Autowiring (XML)

Matches constructor parameter type.

Student.java

```
public class Student {
    private Laptop laptop;

    public Student(Laptop laptop) {    // constructor DI
        this.laptop = laptop;
    }
}
```

spring.xml

```xml
<bean id="laptop" class="org.example.Laptop" />

<bean id="student" class="org.example.Student" autowire="constructor" />
```

✔️Spring sees Student(Laptop) → injects Laptop bean.

🌥️4. Limitation: Multiple Beans of Same Type

If you have:

```xml
<bean id="lap1" class="org.example.Laptop" />
<bean id="lap2" class="org.example.Laptop" />
```

Then:

```xml
<bean id="student" class="org.example.Student" autowire="byType" />
```

Will FAIL because Spring cannot decide which Laptop to inject.

🌥️5. When to Use XML Autowiring?

✔️When working in pure Spring Core (no annotations)
✔️When migrating legacy XML-based applications
✔️When you want wiring rules controlled from config, not code

---

🌥️Summary: XML Autowiring

Mode	Works When	Good For
no	Manual wiring	Full control
byName	Bean id = property name	Simple setups
byType	Only one bean of that type exists	Clean DI
constructor	Constructor params match bean type	Strong DI

⭐13. Putting Everything Together (Example)

Laptop

```
@Component
class Laptop {
    void compile() {
        System.out.println("Compiling...");
    }
}
```

Student

`