

Ultimate Guide to Dependency Injection (DI) & IoC — Beginner Friendly + Deep Explanation

This note explains **DI**, **IoC**, **types of injection**, **why DI is needed**, **polymorphism in DI**, **Spring DI**, **Autowired**, **@Primary**, **Bean selection**, and includes easy code examples.



1. What is Dependency?

A **dependency** is simply an object your class needs to work.

Example:

```
class Car {  
    Engine engine = new Engine();  
}
```

Here: - Car **depends** on Engine → Engine is a *dependency*.



2. What is Dependency Injection (DI)?

DI means:  **Do NOT create the object yourself.**
 **Someone else gives (injects) it to you.**

Example without DI (tight coupling):

```
class Driver {  
    Car car = new Car(); // Driver creates its own dependency  
}
```

Example with DI:

```
class Driver {  
    private Car car;  
  
    public Driver(Car car) { // Car is injected  
        this.car = car;  
    }  
}
```

The object is **given**, not created.



3. DI vs Normal Object Creation

| Without DI | With DI |
|-------------------------------------|---------------------------|
| <code>new Car()</code> inside class | Car passed from outside |
| Class creates its own dependency | Class receives dependency |
| Harder to test | Easier to test |
| Tight coupling | Loose coupling |



4. What is IoC (Inversion of Control)?

IoC = Give control to someone else.

Normally: - You create objects - You control lifecycle

With IoC: - Spring creates objects - Spring manages their lifecycle - You "use" the objects provided by Spring

IoC Container = **ApplicationContext**.

```
ApplicationContext ctx = SpringApplication.run(App.class);
Student s = ctx.getBean(Student.class);
```



5. Types of Dependency Injection

There are 3 main types:



Constructor Injection (Most recommended)

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

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

```
    }
}
```

⚠️ Best for required dependencies. ⚠️ Immutable. ⚠️ Works great with testing.

Setter Injection

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

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

Used when: - Dependency is optional - Value may change later

Field Injection (Not recommended but simplest)

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

Easy but: - Hard to test - No immutability - Not recommended in production

6. Why DI Enables Polymorphism

Polymorphism works because you depend on **interfaces or parent types**, not concrete classes.

Example:

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

class Driver {
    private Car car;
    public Driver(Car car) { this.car = car; }
```

```
    void drive() { car.start(); }  
}
```

If Spring injects `ElectricCar`, Driver automatically uses it. This is **loose coupling**.

7. If both Car & ElectricCar exist — Which one does Spring inject?

Spring gets confused if multiple beans of same type exist. Solutions:

Use `@Primary`

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

Use `@Qualifier`

```
@Autowired  
public Driver(@Qualifier("electricCar") Car car) { ... }
```

8. `@Autowired` — Why do we need it?

`@Autowired` tells Spring:  "Find a bean of this type and inject it here."

But...

 If a class has only one constructor, `@Autowired` is optional.

Spring automatically injects dependencies.

9. What if the class has multiple constructors?

Then `@Autowired` is required.

```
@Component  
class Student {
```

```

@Autowired
public Student(Router router) { }

public Student(Laptop laptop) { }
}

```

Otherwise, Spring won't know which constructor to pick.



10. DI Failure Example (NullPointerException)

```

@Component
class Student {
    private CodeEditor editor;

    public void code() {
        editor.usingEditor(); // NPE
    }
}

```

Why? ➔ Because Spring never injected `editor`.

➔ No constructor/setter or `@Autowired`.

Fix:

```

@Autowired
public void setEditor(CodeEditor editor) {
    this.editor = editor;
}

```



11. DI in Spring Core vs Spring Boot

| Feature | Spring Core | Spring Boot |
|---------------|---------------------------------------|---|
| IoC Container | ApplicationContext | ApplicationContext |
| Bean creation | XML or Java config | Auto config + <code>@ComponentScan</code> |
| Bean scanning | Need explicit component-scan | Automatic (if same package) |
| Bootstrapping | No <code>SpringApplication.run</code> | Uses <code>SpringApplication.run</code> |

12. What is a Bean?

A **bean** is simply an object created & managed by Spring IoC.

Ways to create a bean: - `@Component` - `@Service` - `@Repository` - `@Controller` - `@Bean`
method in config class

13. Spring Without `@ComponentScan`

If you don't write `@ComponentScan`, then: - Spring Boot still scans the **main package** - Because `@SpringBootApplication` includes component scanning for its base package

But in Spring Core, XML must configure:

```
<context:component-scan base-package="com.example"/>
```

14. DI with XML (Spring Core)

Constructor Injection

```
<bean id="student" class="com.Student">
    <constructor-arg ref="laptop" />
</bean>

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

Setter Injection

```
<bean id="student" class="com.Student">
    <property name="laptop" ref="laptop" />
</bean>
```

Autowiring in XML

By name:

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

By type:

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



15. Singleton Beans in Spring

Spring beans are **singleton by default**.

```
Student s1 = ctx.getBean(Student.class);
Student s2 = ctx.getBean(Student.class);
```

Both references point to the **same object**.

To change:

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



16. Final Summary

- DI = object given to class, not created inside
- IoC = Spring creates & manages objects
- Constructor injection is best
- @Autowired auto wires beans
- @Primary & @Qualifier solve bean conflicts
- Spring IoC container = ApplicationContext
- Beans are singletons unless changed
- DI supports polymorphism

If you want, I can add **diagrams, flowcharts, or real project examples** to this note.