In **Spring Boot**, dependency injection (DI) is one of the core features inherited from the **Spring Framework**. It allows the framework to automatically provide ("inject") the required dependencies into a class instead of creating them manually.

There are **three main ways** to perform dependency injection in Spring Boot:

🧩 1. Constructor Injection (Recommended)

This is the **preferred and most modern approach**.

How it works:

- You declare dependencies as constructor parameters.
- Spring automatically injects the required beans when creating the object.

Example:

```
@Component
public class OrderService {

    private final PaymentService paymentService;

    // Constructor injection
    @Autowired // (optional since Spring 4.3 if only one constructor)
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder() {
        paymentService.pay();
    }
}
```

✅ **Advantages:**

- Encourages immutability (`final` fields).

- Makes dependencies explicit.
- Easier to write unit tests (you can pass mock dependencies).
- Avoids potential `NullPointerException` issues.

💉 2. Setter Injection

Uses **setter methods** to inject dependencies after the object is created.

Example:

```
@Component
public class OrderService {

    private PaymentService paymentService;

    @Autowired
    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder() {
        paymentService.pay();
    }
}
```

✅ **Advantages:**

- Useful when a dependency is **optional**.
- Can be reconfigured after object creation.

⚠️ **Disadvantages:**

- Allows mutable state (can change dependency after initialization).

- Not ideal for required dependencies.

⚙️ 3. Field Injection

Injects dependencies **directly into fields** using `@Autowired`.

Example:

```
@Component
public class OrderService {

    @Autowired
    private PaymentService paymentService;

    public void processOrder() {
        paymentService.pay();
    }
}
```

✅ Advantages:

- Very concise and simple to write.

⚠️ Disadvantages:

- Harder to test (you can't easily pass mocks).
- Violates **inversion of control** principles.
- Makes the class less flexible and harder to extend.
- Not recommended for production-level code.

🧠 Bonus: Other Related Injection Features

Feature	Description	Example

`@Qualifier`	Used to specify **which bean** to inject when multiple beans of the same type exist.	`@Qualifier("paypalService") PaymentService service;`
`@Primary`	Marks a bean as the **default** when multiple candidates exist.	`@Primary @Service public class DefaultPaymentService {}`
`@Value`	Inject **simple values** (from properties or literals).	`@Value("${app.name}") private String appName;`
`@Resource` / `@Inject`	Alternatives to `@Autowired` (from JSR-250 / JSR-330).	`@Resource private PaymentService service;`

✅ Summary

Injection Type	When to Use	Pros	Cons
Constructor	Always prefer	Immutable, testable, explicit	Slightly more verbose
Setter	Optional dependencies	Reconfigurable	Mutable state
Field	Quick prototypes	Simple syntax	Hard to test, less flexible