what are the 3 dependency injection techniques in spring boot? Explain me with neat and clear examples and with coding examples and how internally it works with neat examples

**ChatGPT said:**

ChatGPT

Dependency Injection (DI) is a design pattern in Spring Boot where the control of creating and managing dependencies is shifted from the application code to the Spring Framework. This pattern promotes loose coupling and easier testing. Spring Boot provides three common DI techniques:

1. **Constructor Injection**
2. **Setter Injection**
3. **Field Injection**

Below is an explanation of each technique with code examples and internal working details:

**1. Constructor Injection**

**Concept:**

* Dependencies are provided to a class through its constructor.
* This is the most recommended approach as it ensures all required dependencies are available when the object is created.

**Example:**

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

import org.springframework.stereotype.Component;

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

@Component

class Car {

private final Engine engine;

// Constructor Injection

@Autowired

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

@Component

public class ApplicationRunner {

private final Car car;

@Autowired

public ApplicationRunner(Car car) {

this.car = car;

}

public void run() {

car.drive();

}

}

**Internal Working:**

* Spring scans the @Component classes and identifies that Car requires an Engine.
* It creates an instance of Engine first and passes it to the Car constructor during its creation.
* This ensures that the dependency (Engine) is available when the dependent (Car) is created.

**2. Setter Injection**

**Concept:**

* Dependencies are set using public setter methods after the object is created.
* This is useful when dependencies are optional or can change post object creation.

**Example:**

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

@Component

class Car {

private Engine engine;

// Setter Injection

@Autowired

public void setEngine(Engine engine) {

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

@Component

public class ApplicationRunner {

private final Car car;

@Autowired

public ApplicationRunner(Car car) {

this.car = car;

}

public void run() {

car.drive();

}

}

**Internal Working:**

* Spring creates an instance of Car.
* It identifies the @Autowired setter method and injects the Engine instance into the Car object.
* The dependency (Engine) is injected after the Car object is constructed.

**3. Field Injection**

**Concept:**

* Dependencies are injected directly into fields using the @Autowired annotation.
* This is the least recommended approach as it violates encapsulation and makes testing harder.

**Example:**

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

@Component

class Car {

@Autowired

private Engine engine; // Field Injection

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

@Component

public class ApplicationRunner {

private final Car car;

@Autowired

public ApplicationRunner(Car car) {

this.car = car;

}

public void run() {

car.drive();

}

}

**Internal Working:**

* Spring creates an instance of Car and directly injects the Engine instance into the engine field.
* The dependency (Engine) is injected into the field even before any method or constructor of the class is invoked.

**Comparison of Techniques**

| **Technique** | **Pros** | **Cons** |
| --- | --- | --- |
| Constructor Injection | Ensures immutability, dependencies are final, and easier testing | Can become verbose with many dependencies |
| Setter Injection | Useful for optional dependencies | Dependencies can be modified after object creation (less safe) |
| Field Injection | Simplifies code and reduces boilerplate | Harder to test and violates encapsulation principles |

**How Spring Boot Handles DI Internally**

1. **Component Scanning**:
   * Spring Boot scans classes annotated with @Component, @Service, @Repository, or @Controller and registers them in the Spring Context.
2. **Dependency Resolution**:
   * When creating a bean, Spring looks for dependencies marked with @Autowired and resolves them by searching the Spring Context for matching beans.
3. **Bean Lifecycle**:
   * Spring handles the lifecycle of beans, ensuring that dependencies are injected before the bean is used.
4. **Proxy Mechanism**:
   * If dependencies have specific behaviors (e.g., transactional or lazy loading), Spring may use dynamic proxies to manage their behavior transparently.

Would you like me to create a step-by-step explanation for the flow of one of these examples, or do you have further questions about DI?

4o

**You said:**

For field injection you mentioned below are the cons but I didn't understand so can you explain these cons with coding examples and show me the problem live "Harder to test and violates encapsulation principles"

**ChatGPT said:**

ChatGPT

Let's break down the **cons of field injection** with coding examples to understand the problems they introduce.

**1. Harder to Test**

**Explanation:**

Field injection makes testing harder because the dependency is private and cannot be easily accessed or replaced without using reflection. This violates the principle of ease of testing.

**Field Injection Example:**

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

@Component

class Car {

@Autowired

private Engine engine; // Field Injection

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Problem During Testing:**

When writing a unit test for the Car class, you can't replace the Engine dependency without reflection or special frameworks.

public class CarTest {

@Test

public void testDrive() {

Car car = new Car();

// Cannot set `engine` as it is private and no setter/constructor is available

car.drive(); // Will throw NullPointerException

}

}

**Solution with Constructor Injection:**

With constructor injection, you can easily provide a mock dependency.

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) { // Constructor Injection

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Test:**

public class CarTest {

@Test

public void testDrive() {

Engine mockEngine = Mockito.mock(Engine.class);

Mockito.when(mockEngine.start()).thenReturn("Mock Engine started!");

Car car = new Car(mockEngine); // Provide mock dependency

car.drive(); // Output: Mock Engine started! Car is running!

}

}

**Why Constructor Injection Wins**:

* Dependencies are visible and replaceable.
* No need for reflection or frameworks to inject mocks.

**2. Violates Encapsulation Principles**

**Explanation:**

Encapsulation is about hiding the internal implementation and providing access through controlled methods. Field injection exposes internal fields directly to the framework, breaking encapsulation.

**Field Injection Example:**

@Component

class Car {

@Autowired

private Engine engine; // Field Injection

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Problem:**

1. **Dependency Visibility**:
   * Other developers can't see dependencies from outside because they are injected privately.
   * The dependency lifecycle is unclear from the class's external interface.
2. **Direct Framework Dependency**:
   * The class is tightly coupled to Spring because @Autowired is directly used on fields.
   * This makes the class less portable to environments without Spring.

**Encapsulation with Constructor Injection:**

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) { // Constructor Injection

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Advantages**:

* The dependency is part of the class's API (Car(Engine engine)).
* Easier to understand what Car needs to function.

**Key Issues Demonstrated:**

1. **Hard to Test**:
   * Field injection requires workarounds for testing, such as reflection or Spring-specific utilities.
2. **Breaks Encapsulation**:
   * Dependencies are hidden and injected directly by the framework, reducing the clarity of the class design.

Would you like me to provide a full example showing reflection-based testing with field injection for better clarity?

4o

**You said:**

I didn't understand the below points so explain me with neat and clear explanations with coding examples "Problem: Dependency Visibility: Other developers can't see dependencies from outside because they are injected privately. The dependency lifecycle is unclear from the class's external interface. Direct Framework Dependency: The class is tightly coupled to Spring because @Autowired is directly used on fields. This makes the class less portable to environments without Spring."

**ChatGPT said:**

ChatGPT

Let's break down these problems with **field injection** into simpler explanations and provide detailed examples for each.

**Problem 1: Dependency Visibility**

**Explanation:**

* With field injection, the class depends on certain components (dependencies) to work, but these dependencies are **hidden** from the class's public interface.
* Developers cannot understand what dependencies are needed just by looking at the class definition.

**Example:**

@Component

class Car {

@Autowired

private Engine engine; // Field Injection

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **What's hidden?**
  + There is no indication that the Car class needs an Engine just by looking at its constructor or methods.
  + This makes the dependency invisible to developers inspecting the class.

**Problem During Usage:**

A developer might try to create a Car instance without realizing it requires an Engine:

public class Main {

public static void main(String[] args) {

Car car = new Car(); // Engine dependency is not obvious

car.drive(); // Throws NullPointerException because engine is null

}

}

**Solution: Constructor Injection:**

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) { // Constructor Injection

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **What changes?**
  + The Car class explicitly declares its dependency through the constructor. Anyone creating a Car must provide an Engine.

**Clear Usage:**

public class Main {

public static void main(String[] args) {

Engine engine = new Engine(); // Explicit dependency creation

Car car = new Car(engine); // Dependency passed explicitly

car.drive(); // Works as expected

}

}

**Problem 2: The Dependency Lifecycle Is Unclear**

**Explanation:**

* With field injection, the framework injects the dependency behind the scenes, and it's unclear when or how the dependency is set.
* Developers might assume the dependency is always available, leading to runtime errors like NullPointerException.

**Example:**

@Component

class Car {

@Autowired

private Engine engine;

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **Lifecycle Issue**:
  + Before Spring initializes the Car bean, the engine field is null.
  + If drive() is called before Spring finishes wiring, it throws a NullPointerException.

**Solution: Constructor Injection:**

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) {

this.engine = engine; // Ensures engine is initialized during object creation

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **What changes?**
  + The engine is initialized during object creation, ensuring it's never null.

**Problem 3: Direct Framework Dependency**

**Explanation:**

* Using @Autowired directly on fields tightly couples the class to the Spring Framework.
* If you decide to run this code in a non-Spring environment, it will fail because the framework isn't available to inject dependencies.

**Example:**

class Car {

@Autowired

private Engine engine; // Tightly coupled to Spring

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **Framework Dependency**:
  + The Car class relies on Spring's @Autowired to inject Engine. Without Spring, there's no way to initialize engine.

**Problem in Non-Spring Environment:**

public class Main {

public static void main(String[] args) {

Car car = new Car(); // `engine` is null because Spring isn't managing it

car.drive(); // Throws NullPointerException

}

}

**Solution: Constructor Injection:**

class Car {

private final Engine engine;

public Car(Engine engine) { // No Spring dependency here

this.engine = engine;

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

* **What changes?**
  + The Car class no longer depends on Spring-specific annotations. You can use it in any environment.

**Works Without Spring:**

public class Main {

public static void main(String[] args) {

Engine engine = new Engine();

Car car = new Car(engine); // Dependency passed explicitly

car.drive(); // Works as expected

}

}

**Comparison Table**

| **Aspect** | **Field Injection** | **Constructor Injection** |
| --- | --- | --- |
| **Dependency Visibility** | Hidden from the class interface | Clearly visible in the constructor |
| **Lifecycle Clarity** | Dependency injection happens behind the scenes | Dependency is initialized at creation |
| **Framework Coupling** | Tightly coupled to Spring (@Autowired) | No Spring-specific code in the class |
| **Reusability** | Difficult to use without Spring | Can be used in non-Spring environments |

**Key Takeaways**

1. **Field Injection** hides dependencies and relies on Spring to handle them behind the scenes, making it hard to test and reuse the code outside Spring.
2. **Constructor Injection** makes dependencies explicit, ensures they are initialized at creation, and decouples your class from the Spring Framework.

Would you like further examples or clarifications?

4o

**You said:**

This is not clear to me, you have mentioned earlier that filed injection happens before creation of object but you are saying opposite here and which is corect? "Lifecycle Issue: Before Spring initializes the Car bean, the engine field is null. If drive() is called before Spring finishes wiring, it throws a NullPointerException."

**ChatGPT said:**

ChatGPT

You're absolutely right to ask for clarification! Let me explain **how field injection works** in Spring and **why the lifecycle issue happens** in clearer terms.

**Field Injection and Bean Lifecycle**

**When Does Field Injection Happen?**

Field injection happens *after* the object (bean) is created but *before* the bean is fully initialized and ready for use.

**Why Is This Important?**

* When a bean (e.g., Car) is created, Spring first calls its no-argument constructor (even if you don't define one explicitly).
* After the object is created, Spring injects the dependencies into fields marked with @Autowired.
* Until Spring performs this injection, the fields (like engine) remain null.

**Lifecycle Issue Explained**

**Scenario:**

If you try to use the bean (e.g., call the drive() method) before Spring completes dependency injection, you will encounter a NullPointerException because the injected field (engine) is still null.

**Example of Field Injection with a Problem:**

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

@Component

class Car {

@Autowired

private Engine engine; // Field Injection

public Car() {

// Constructor is called before field injection

System.out.println("Car object created. Engine is: " + engine); // Prints "Engine is: null"

}

public void drive() {

System.out.println(engine.start() + " Car is running!"); // NullPointerException if engine is null

}

}

**Output when Spring initializes the Car bean:**

csharp

Car object created. Engine is: null

Engine started! Car is running!

Notice:

1. When the constructor is called, the engine field is null.
2. After the constructor finishes, Spring performs field injection to populate engine.

**Why Does This Cause Issues?**

If you use the object (e.g., call drive()) **before Spring finishes injecting dependencies**, you'll get a NullPointerException.

**Example:**

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

}

public class Main {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Car car = context.getBean(Car.class);

car.drive(); // Works because Spring has injected the dependency

}

}

Now let's break it:

public class Main {

public static void main(String[] args) {

Car car = new Car(); // Directly creating the object without Spring

car.drive(); // Throws NullPointerException because Spring is not managing the bean

}

}

**Constructor Injection: The Solution**

With **constructor injection**, the dependency (engine) is required at the time of object creation. This guarantees that the object is fully initialized and ready to use immediately.

**Constructor Injection Example:**

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) { // Constructor Injection

this.engine = engine;

System.out.println("Car object created. Engine is: " + engine); // Always prints a valid engine object

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Output:**

kotlin

Car object created. Engine is: Engine@12345

Engine started! Car is running!

**Key Difference:**

* In constructor injection, engine is guaranteed to be initialized before the Car object is used.
* There’s no window where the engine field is null.

**Summary**

| **Injection Type** | **When Dependencies Are Set?** | **Can Fields Be null Temporarily?** | **NullPointerException Risk?** |
| --- | --- | --- | --- |
| **Field Injection** | After the object is created | Yes | Yes, if used too early |
| **Constructor Injection** | At the time of object creation | No | No |

**Correct Lifecycle Understanding:**

* **Field injection happens after object creation but before full bean initialization**.
* If you call a method that uses a field-injected dependency too early, you may encounter a NullPointerException.

Would you like me to provide a more detailed lifecycle flow or address other concerns?

4o

**You said:**

I didn't understand this so explain me with neat and clear explanations an with coding examples "Internal Working: Spring scans the @Component classes and identifies that Car requires an Engine. It creates an instance of Engine first and passes it to the Car constructor during its creation. This ensures that the dependency (Engine) is available when the dependent (Car) is created. "

**ChatGPT said:**

ChatGPT

Let’s break this down step-by-step to clearly understand how Spring handles dependency injection internally, using **constructor injection** as an example.

**How Spring Internally Works During Constructor Injection**

**Step 1: Spring Scans the Classes**

When the application starts:

1. Spring scans for classes annotated with @Component, @Service, @Repository, etc., within the specified package.
2. It identifies dependencies that need to be injected.

**Step 2: Spring Resolves Dependencies**

1. For each class, Spring checks its constructor (in the case of constructor injection).
2. If the constructor has parameters, Spring resolves those parameters by creating or finding the required beans.

**Step 3: Bean Creation**

1. Spring first creates the beans for the dependencies (in this case, Engine).
2. It then creates the dependent bean (in this case, Car) by passing the resolved dependencies to its constructor.

**Example Code**

**Classes:**

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

import org.springframework.stereotype.Component;

// Dependency class

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

// Dependent class

@Component

class Car {

private final Engine engine;

@Autowired

public Car(Engine engine) { // Constructor Injection

this.engine = engine;

System.out.println("Car created with Engine: " + engine);

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**Configuration:**

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

}

**Main Class:**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Car car = context.getBean(Car.class); // Get the Car bean

car.drive();

}

}

**Internal Working Step-by-Step**

**1. Spring Scans the @Component Classes**

* Spring detects the Engine and Car classes as @Component beans during the application context initialization.

**2. Resolves Dependencies**

* Spring sees that the Car class requires an Engine in its constructor.
* It checks if a bean of type Engine exists.
  + If the Engine bean is already created, Spring uses it.
  + If not, Spring creates a new instance of Engine.

**3. Creates the Engine Bean**

* Spring creates the Engine bean first because it is needed to create the Car bean.

**4. Creates the Car Bean**

* Spring calls the constructor of the Car class, passing the Engine bean as an argument.
* The Car object is now fully initialized and ready for use.

**Output**

When you run the application, the output would look like this:

sql

Car created with Engine: com.example.Engine@1f2a3b4

Engine started! Car is running!

**Detailed Explanation**

1. **Step 1: Engine Bean Creation**
   * When Spring encounters the Engine class:

@Component

class Engine { ... }

It creates an instance of Engine:

Engine engine = new Engine();

This bean is stored in the Spring container for future use.

1. **Step 2: Car Bean Creation**
   * Spring then processes the Car class:

@Component

class Car {

@Autowired

public Car(Engine engine) { ... }

}

* + It sees that Car requires an Engine in its constructor.
  + Since the Engine bean already exists in the Spring container, Spring retrieves it and passes it to the Car constructor:

Car car = new Car(engine);

1. **Step 3: Dependency Injection**
   * During the constructor call, the Car object receives the Engine instance:

public Car(Engine engine) {

this.engine = engine; // Dependency is injected here

}

1. **Step 4: Bean Registration**
   * Spring registers the fully constructed Car bean in the container for later use.

**Key Points About the Process**

* **Order of Bean Creation**: Dependencies are created first, ensuring that all required beans are available when creating the dependent bean.
* **Dependency Resolution**: Spring uses its **IoC (Inversion of Control) container** to resolve dependencies.
* **Lifecycle Management**: Once created, beans are managed by Spring (e.g., for lifecycle callbacks like @PostConstruct or @PreDestroy).

**Advantages of Constructor Injection**

1. **Dependency Initialization Guarantee**:
   * The Car class cannot be created without providing an Engine.
   * This ensures all required dependencies are ready when the object is created.
2. **Clear Dependency Relationships**:
   * By looking at the constructor, developers can immediately see what dependencies the class needs:

public Car(Engine engine) { ... }

1. **Easier Testing**:
   * You can create a Car object with a mock Engine in unit tests without relying on Spring.

**Conclusion**

1. Spring resolves the dependency (Engine) first.
2. It injects the dependency into the constructor of the dependent class (Car).
3. This ensures that when the Car object is created, all its dependencies are ready and available.

Would you like to explore further, such as debugging Spring's bean creation process or analyzing more complex examples?

4o

**You said:**

I didn't understand below points so explain me with neat and clear explanations ad with coding examples "Internal Working: Spring creates an instance of Car. It identifies the @Autowired setter method and injects the Engine instance into the Car object. The dependency (Engine) is injected after the Car object is constructed."

**ChatGPT said:**

ChatGPT

Let’s clearly explain how **setter-based dependency injection** works internally in Spring. I’ll include **step-by-step explanations** and **coding examples** to make it easy to follow.

**Internal Working of Setter Injection**

**What Happens Internally?**

1. **Spring Creates the Instance of the Bean (e.g., Car)**:
   * Spring first creates an object of the dependent class (Car) using its no-argument constructor.
2. **Spring Identifies the @Autowired Setter Method**:
   * Spring scans the class for methods annotated with @Autowired to determine which dependencies need to be injected.
3. **Spring Injects the Dependency**:
   * After the object is created, Spring calls the identified setter method (e.g., setEngine()) to inject the dependency (e.g., Engine).
4. **The Dependency Is Ready for Use**:
   * At this point, the dependency (Engine) is fully initialized and ready to be used by the Car object.

**Example Code**

**1. Dependency Class (Engine):**

import org.springframework.stereotype.Component;

@Component

class Engine {

public String start() {

return "Engine started!";

}

}

**2. Dependent Class (Car):**

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

import org.springframework.stereotype.Component;

@Component

class Car {

private Engine engine;

public Car() {

System.out.println("Car object created. Engine is: " + engine); // Engine is null here

}

@Autowired

public void setEngine(Engine engine) { // Setter Injection

this.engine = engine;

System.out.println("Engine injected into Car via setEngine method.");

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**3. Configuration Class:**

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

}

**4. Main Class:**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Car car = context.getBean(Car.class); // Get the Car bean

car.drive();

}

}

**Step-by-Step Internal Working**

**Step 1: Spring Creates the Car Object**

* Spring invokes the no-argument constructor of the Car class:

Car car = new Car(); // Engine is null at this point

**Step 2: Spring Identifies the @Autowired Setter**

* Spring scans the Car class and finds the setEngine() method annotated with @Autowired.

**Step 3: Spring Creates the Engine Bean**

* Spring creates an instance of the Engine class (if it doesn’t already exist in the container):

Engine engine = new Engine();

**Step 4: Spring Injects the Dependency**

* Spring calls the setEngine() method to inject the Engine object into the Car object:

car.setEngine(engine);

**Step 5: Car Bean Is Fully Initialized**

* After dependency injection, the Car object is fully initialized and ready for use.

**Output**

When you run the application, the console output will be:

csharp

Car object created. Engine is: null

Engine injected into Car via setEngine method.

Engine started! Car is running!

**Key Observations**

1. **Before Injection**:
   * When the Car object is created using the constructor, the engine field is null.
   * This is because setter injection happens *after* object creation.
2. **During Injection**:
   * Spring calls the setEngine() method and passes the Engine object as a parameter.
   * The dependency (Engine) is injected into the Car object.
3. **After Injection**:
   * The Car object is fully initialized, and the drive() method can safely use the Engine dependency.

**Advantages of Setter Injection**

1. **Optional Dependencies**:
   * Setter injection allows injecting dependencies only when needed. You don’t have to provide all dependencies during object creation.
2. **Flexibility**:
   * You can change dependencies after the object is created, making it more flexible.

**Lifecycle Comparison: Constructor vs Setter Injection**

| **Aspect** | **Constructor Injection** | **Setter Injection** |
| --- | --- | --- |
| **When Dependency Is Set** | During object creation | After the object is created |
| **Field Availability** | Dependency is guaranteed to be initialized | Field is null until the setter method is called |
| **Flexibility** | Dependencies are fixed once the object is created | Dependencies can be changed after object creation |
| **Optional Dependencies** | Not suitable for optional dependencies | Ideal for optional dependencies |

**Key Takeaways**

1. **Spring creates the dependent bean (Car) first** using its no-argument constructor.
2. **Spring then injects the dependency (Engine) by calling the setter method** annotated with @Autowired.
3. Setter injection happens **after object creation**, which is why the engine field is null when the Car constructor is executed.

Would you like further clarifications or examples?

4o

**You said:**

Hey I understood, how do we change dependencies with setter injection, can you show me code examples live?

**ChatGPT said:**

ChatGPT

Sure! Let me demonstrate how you can **change dependencies dynamically** using setter injection in Spring. This flexibility is one of the key advantages of setter injection.

**Scenario**

Suppose you have two types of engines:

1. **PetrolEngine**
2. **DieselEngine**

You want to dynamically inject a different engine into the Car object and change it later using the setter method.

**Code Example**

**1. Define the Engine Interface**

public interface Engine {

String start();

}

**2. Create Two Implementations of Engine**

**PetrolEngine**:

import org.springframework.stereotype.Component;

@Component

class PetrolEngine implements Engine {

@Override

public String start() {

return "Petrol engine started!";

}

}

**DieselEngine**:

import org.springframework.stereotype.Component;

@Component

class DieselEngine implements Engine {

@Override

public String start() {

return "Diesel engine started!";

}

}

**3. Modify the Car Class to Support Dynamic Dependency Change**

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

import org.springframework.stereotype.Component;

@Component

class Car {

private Engine engine;

@Autowired

public void setEngine(Engine engine) { // Setter Injection

this.engine = engine;

System.out.println("Engine injected: " + engine.getClass().getSimpleName());

}

public void changeEngine(Engine newEngine) { // Dynamically change engine

this.engine = newEngine;

System.out.println("Engine changed to: " + newEngine.getClass().getSimpleName());

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

**4. Configuration Class**

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

}

**5. Main Class to Test Dynamic Dependency Change**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

public static void main(String[] args) {

// Initialize Spring context

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Get the Car bean

Car car = context.getBean(Car.class);

// Get the PetrolEngine and DieselEngine beans

PetrolEngine petrolEngine = context.getBean(PetrolEngine.class);

DieselEngine dieselEngine = context.getBean(DieselEngine.class);

// Initially drive with the injected engine

car.drive();

// Dynamically change the engine

car.changeEngine(dieselEngine);

// Drive again with the new engine

car.drive();

context.close();

}

}

**Output**

vbnet

Engine injected: PetrolEngine

Petrol engine started! Car is running!

Engine changed to: DieselEngine

Diesel engine started! Car is running!

**Explanation of Dynamic Dependency Change**

1. **Initial Dependency Injection**:
   * When Spring initializes the Car bean, it injects a default engine (e.g., PetrolEngine) via the @Autowired setter method.
2. **Changing Dependencies Dynamically**:
   * Later in the application, you can call the changeEngine() method to replace the existing engine with a new one (e.g., DieselEngine).

car.changeEngine(dieselEngine);

1. **New Dependency in Action**:
   * When you call drive() after changing the engine, the Car object uses the new engine dependency.

**When Is This Useful?**

* **Testing Different Configurations**: You can switch dependencies for testing different setups.
* **Runtime Customization**: Some applications may need to change their behavior dynamically based on user input or configuration changes.

Would you like to explore a real-world scenario or add enhancements to this example?

4o

**You said:**

Hey in the above code, why it took the PetrolEngine first, we have two implementations for the Engine and we didn't specify the qualifier for any implementation but how did it choose the PetrolEngine and not the DiselEngine? It should throw exception right like 2 beans identified and which one to use like that?

**ChatGPT said:**

ChatGPT

Great question! When you have multiple beans of the same type (in this case, PetrolEngine and DieselEngine, both implementing the Engine interface), Spring needs to determine which one to inject into the dependent class. Let’s understand **why it chose PetrolEngine by default** and why it **did not throw an exception**.

**How Spring Handles Multiple Candidates**

1. **Primary Bean**
   * Spring injects the bean marked with @Primary if no other qualifier is specified.
   * If no bean is marked as @Primary, Spring throws an exception, as it cannot decide which one to use.
2. **No Qualifier or Primary**
   * If neither @Primary nor @Qualifier is used and there are multiple candidates:
     + Spring may choose a bean **alphabetically** based on the class name or bean name. This behavior is not guaranteed and can depend on the Spring version or configuration.
3. **Exception**
   * If Spring cannot decide (and there is no deterministic fallback like the above), it throws a NoUniqueBeanDefinitionException.

**Why It Didn't Throw an Exception?**

**Possible Scenarios:**

1. **Spring Chose Alphabetically (Default Behavior)**:
   * In your case, PetrolEngine might have been chosen because its name comes before DieselEngine alphabetically.
2. **Implicit Behavior in Some Versions of Spring**:
   * In some versions, Spring injects one of the beans if there’s no explicit conflict resolution.

**How to Fix Ambiguity?**

To avoid such unpredictable behavior, you should explicitly specify which bean to inject. Here’s how you can resolve this:

**Solution 1: Use @Primary**

You can mark one implementation as the default using the @Primary annotation.

**Example:**

@Component

@Primary

class PetrolEngine implements Engine {

@Override

public String start() {

return "Petrol engine started!";

}

}

Now, Spring will always inject PetrolEngine unless a specific qualifier is used.

**Solution 2: Use @Qualifier**

You can use the @Qualifier annotation to explicitly specify which bean to inject.

**Example:**

@Component

class Car {

private Engine engine;

@Autowired

@Qualifier("dieselEngine") // Explicitly specify which implementation to inject

public void setEngine(Engine engine) {

this.engine = engine;

System.out.println("Engine injected: " + engine.getClass().getSimpleName());

}

public void drive() {

System.out.println(engine.start() + " Car is running!");

}

}

Now, Spring will inject the bean with the name dieselEngine.

**Solution 3: Rename Beans for Clarity**

You can specify a custom bean name for each implementation and use @Qualifier to refer to it.

**Example:**

**PetrolEngine**:

@Component("petrolEngine")

class PetrolEngine implements Engine {

@Override

public String start() {

return "Petrol engine started!";

}

}

**DieselEngine**:

@Component("dieselEngine")

class DieselEngine implements Engine {

@Override

public String start() {

return "Diesel engine started!";

}

}

Now, use @Qualifier as shown earlier to refer to the specific implementation.

**What If We Don't Resolve the Conflict?**

If you have multiple candidates and don’t use @Primary or @Qualifier, Spring will throw an exception like this:

vbnet

org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type 'Engine' available: expected single matching bean but found 2: petrolEngine,dieselEngine

**Key Takeaway**

* By default, Spring may pick one bean based on **alphabetical order**, but this behavior is **not reliable** and should not be relied upon.
* Always use @Primary or @Qualifier to avoid ambiguity and make your code explicit and predictable.

Would you like to try resolving this ambiguity in your code, or do you need further clarification?