# Spring Framework: Dependency Injection & IoC - Complete Notes

## Table of Contents



---

## 1. Dependency Injection (DI)

### What is Dependency Injection?

**First Principle**: A dependency is something an object needs to function. Injection means providing that dependency from outside rather than creating it inside.

Think of it like this: If you're making coffee, the coffee maker is dependent on electricity. You don't generate electricity inside the coffee maker - it gets injected from outside through a power cord.

---

### Without Dependency Injection (The Problem)

```java
// Tight coupling - BAD
public class Car {
    private Engine engine;

    public Car() {
        this.engine = new PetrolEngine(); // Car creates its own engine
    }

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

**Line-by-Line Explanation**:

- `public class Car {` - We define a Car class
- `private Engine engine;` - Car needs an Engine to function (this is the dependency)
- `public Car() {` - The constructor is called when creating a new Car
- `this.engine = new PetrolEngine();` - **HERE'S THE PROBLEM**: The Car class itself creates a PetrolEngine object using the `new` keyword. This creates tight coupling.

**Why is this bad?**

1. **Inflexible**: Car is hardcoded to use only PetrolEngine. What if you want an ElectricEngine? You must change the Car class code.
2. **Hard to Test**: You cannot test Car with a fake/mock engine. You're stuck with PetrolEngine.
3. **Tight Coupling**: Car knows too much about Engine's implementation. If PetrolEngine's constructor changes (e.g., needs fuel type parameter), you must change Car.
4. **Violates Single Responsibility**: Car is responsible for both its own behavior AND creating its Engine.

**Real-World Analogy**: It's like your phone manufacturing its own battery internally. If the battery dies, you can't replace it. If you want a better battery, you must redesign the entire phone.

---

**With Dependency Injection (The Solution)**

```java
```

```
// Loose coupling - GOOD
public class Car {
    private Engine engine;

    // Dependency is injected from outside
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.ignite();
    }
}

// Interface
interface Engine {
    void ignite();
}

// Implementations
class PetrolEngine implements Engine {
    public void ignite() {
        System.out.println("Petrol engine started");
    }
}

class ElectricEngine implements Engine {
    public void ignite() {
        System.out.println("Electric engine started");
    }
}

// Usage
Engine petrolEngine = new PetrolEngine();
Car car1 = new Car(petrolEngine); // Inject petrol engine

Engine electricEngine = new ElectricEngine();
Car car2 = new Car(electricEngine); // Inject electric engine
```

**Detailed Line-by-Line Explanation**:

**Car Class**:

```java
public class Car {
    private Engine engine;
```

- `private Engine engine;` - Car still has an Engine dependency, but notice it's the interface type `Engine`, not a concrete class. This is **dependency on abstraction**, not implementation.

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

- `public Car(Engine engine)` - The constructor now accepts an Engine as a parameter. Car doesn't create the Engine - someone else provides it.
- `this.engine = engine;` - We simply store the provided engine. Car doesn't care if it's Petrol, Electric, or Nuclear - as long as it implements Engine interface.

**Why is this better?**

The Car class is now:

- **Flexible**: Works with any Engine implementation
- **Testable**: You can inject a MockEngine for testing
- **Loosely Coupled**: Car knows nothing about PetrolEngine or ElectricEngine specifics
- **Single Responsibility**: Car only handles car behavior, not engine creation

---

**Engine Interface**:

```java
interface Engine {
    void ignite();
}
```

- `interface Engine` - This defines a contract. Any class implementing Engine MUST provide an `ignite()` method.

- `void ignite();` - This is an abstract method (no implementation). It says "every engine must be able to ignite, but HOW it ignites is up to the specific engine type."

**Why do we need this interface?**

Because Car needs to work with multiple engine types. The interface provides:

1. **Polymorphism**: Car can accept any Engine implementation
2. **Abstraction**: Car doesn't need to know implementation details
3. **Contract**: Guarantees that whatever engine we inject will have an `ignite()` method

---

**PetrolEngine Implementation**:

```
java

class PetrolEngine implements Engine {
    public void ignite() {
        System.out.println("Petrol engine started");
    }
}
```

- `implements Engine` - This tells Java that PetrolEngine is a type of Engine and must implement all interface methods
- `public void ignite()` - Here we provide the ACTUAL implementation. A petrol engine ignites by combustion (simulated by printing a message)

---

**ElectricEngine Implementation**:

```
java

class ElectricEngine implements Engine {
    public void ignite() {
        System.out.println("Electric engine started");
    }
}
```

- Same structure as PetrolEngine, but different implementation

- This demonstrates the power of interfaces: both engines implement the same contract but behave differently

---

**Usage (Dependency Injection in Action)**:

```java
Engine petrolEngine = new PetrolEngine();
Car car1 = new Car(petrolEngine);
```

- `new PetrolEngine()` - We create the engine object OUTSIDE the Car class
- `new Car(petrolEngine)` - We **inject** the engine into the Car through its constructor
- The Car receives the dependency from outside rather than creating it

```java
Engine electricEngine = new ElectricEngine();
Car car2 = new Car(electricEngine);
```

- Same Car class, different engine! No changes needed to Car.
- This is the beauty of DI: we can switch implementations without modifying the dependent class.

**Real-World Analogy**: Your phone has a removable battery. The phone (Car) doesn't create the battery (Engine) - you insert it from outside. Want a better battery? Just swap it. The phone doesn't need to change.

---

## 2. Inversion of Control (IoC)

### What is IoC?

**First Principle**: Instead of your code controlling the flow and creating objects, you hand over that control to a framework (Spring Container).

Traditional programming: **You are the boss**. You create objects, manage their lifecycle, call methods.

IoC: **The framework is the boss**. You tell it what you need, and it manages everything.

---

**Traditional Control Flow (Without IoC)**

```java
// You control everything
public class Application {
    public static void main(String[] args) {
        // You create objects
        Engine engine = new PetrolEngine();
        Car car = new Car(engine);

        // You control the flow
        car.start();
    }
}
```

**Line-by-Line Explanation**:

```java
public static void main(String[] args) {
```

- This is the entry point of our Java application. Execution starts here.

```java
    Engine engine = new PetrolEngine();
```

- **YOU** decide to create a PetrolEngine
- **YOU** use the `new` keyword to instantiate it
- **YOU** manage when it's created and store the reference in `engine` variable

```java
    Car car = new Car(engine);
```

- **YOU** create the Car object
- **YOU** manually inject the engine dependency
- **YOU** are responsible for wiring objects together

```java
    car.start();
```

- **YOU** call the method to start the car
- **YOU** control the entire flow of execution

**The Problem**: You're doing everything manually:

- Creating objects (what if Car needs 10 dependencies?)
- Managing dependencies (what order to create them?)
- Controlling lifecycle (when to create/destroy?)
- Wiring everything together (error-prone)

As your application grows, this becomes unmanageable. Imagine creating 50+ objects with complex dependencies by hand!

---

**Inversion of Control Flow (With IoC)**

```java
```

```java
// Spring controls object creation and lifecycle
@Configuration
public class AppConfig {
    @Bean
    public Engine engine() {
        return new PetrolEngine();
    }

    @Bean
    public Car car() {
        return new Car(engine()); // Spring manages this
    }
}

public class Application {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfig.class);

        // Spring creates and injects dependencies
        Car car = context.getBean(Car.class);
        car.start();
    }
}
```

**Detailed Line-by-Line Explanation**:

**Configuration Class**:

```java
@Configuration
public class AppConfig {
```

- `@Configuration` - This annotation tells Spring: "This class contains bean definitions and configuration"
- Spring will scan this class and process the methods marked with `@Bean`
- Think of this as a recipe book where you define how objects should be created

```java
```

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

- `@Bean` - This annotation tells Spring: "This method produces an object that should be managed by the Spring container"
- `public Engine engine()` - The method name `engine` becomes the bean name (identifier)
- `return new PetrolEngine();` - This is how the object is created, but **YOU don't call this method**. Spring calls it when needed.
- Spring will store this PetrolEngine object in its container and manage its lifecycle

**Why is this powerful?**

- Spring calls this method exactly once (by default) - singleton pattern
- Spring manages when to create the object
- Other beans can reference this bean by name or type

```
java

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

- Another `@Bean` method defining how to create a Car
- `engine()` - We call the engine() method here. Spring is smart enough to:
    1. See that `engine()` is also a `@Bean` method
    2. Reuse the existing Engine bean instead of creating a new one
    3. Inject that Engine into the Car
- This is **dependency injection happening at configuration level**

---

**Application Class**:

```
java
```

```
public static void main(String[] args) {
```

- Entry point of the application

```
java

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

- `ApplicationContext` - This is the Spring IoC container. It's like a box that holds all your objects (beans).
- `new AnnotationConfigApplicationContext(AppConfig.class)` - We create the Spring container and tell it to read configuration from `AppConfig.class`

**What happens behind the scenes?**

1. Spring scans `AppConfig` class
2. Finds methods annotated with `@Bean`
3. Calls `engine()` method → creates PetrolEngine → stores in container
4. Calls `car()` method → injects Engine → creates Car → stores in container
5. Now the container has both objects ready to use

```
java

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

- `getBean(Car.class)` - We ask Spring: "Give me the Car object"
- Spring looks in its container, finds the Car bean, and returns it
- **We didn't create it** - Spring did that for us when it started up
- The Car is already fully configured with its Engine dependency injected

```
java

    car.start();
```

- Now we use the Car normally
- The Engine is already injected and ready

---

**The Key Difference**:

**Without IoC**:

```
You → create Engine → create Car → inject Engine → call methods
```

**With IoC**:

```
You → tell Spring what you need
Spring → creates everything → manages dependencies → gives you ready objects
You → just use the objects
```

**Real-World Analogy**:

**Without IoC (Traditional)**: You're building a house. You must:

- Order bricks yourself
- Hire plumbers yourself
- Coordinate electricians yourself
- Assemble everything yourself
- Make sure everything is done in the right order

**With IoC (Spring)**: You hire a general contractor. You tell them:

- "I need a house with these specifications"
- The contractor handles all the details
- You just move into the finished house

The **control is inverted -** from you to the contractor (Spring).

---

## 3. Configuration Approaches

Spring offers three ways to configure beans and dependencies. Each approach tells Spring HOW to create and wire objects.

---

### 3.1 XML Configuration

**Use Case**: Legacy applications, external configuration without recompilation

**Why XML existed**: Before Java 5 (released 2004), Java didn't have annotations. XML was the only way to configure Spring externally without changing code.

```xml
<!-- applicationContext.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<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">

    <!-- Define beans -->
    <bean id="petrolEngine" class="com.example.PetrolEngine"/>

    <bean id="electricEngine" class="com.example.ElectricEngine"/>

    <!-- Constructor injection -->
    <bean id="car" class="com.example.Car">
        <constructor-arg ref="petrolEngine"/>
    </bean>

    <!-- Setter injection -->
    <bean id="driver" class="com.example.Driver">
        <property name="name" value="John"/>
        <property name="car" ref="car"/>
    </bean>
</beans>
```

**Detailed Line-by-Line Explanation**:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

- Standard XML declaration
- Tells parser: "This is XML version 1.0, encoded in UTF-8"
- Must be the first line of any XML file

```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">
```

- `<beans>` - Root element. Everything inside is Spring configuration
- `xmlns` (XML namespace) - Defines which tags are allowed. Prevents conflicts if multiple libraries use similar tag names
- `xsi:schemaLocation` - Points to the schema definition file. This file defines the rules for what's valid XML (like a grammar book)

**Why do we need this boilerplate?**

- XML parsers need to validate the structure
- Ensures you don't make typos (e.g., `<ben>` instead of `<bean>`)
- IDEs can provide auto-completion based on the schema

---

**Defining Simple Beans**:

```xml
  <bean id="petrolEngine" class="com.example.PetrolEngine"/>
```

- `<bean>` - Tells Spring: "Create an object and manage it"
- `id="petrolEngine"` - The unique identifier for this bean. Think of it as a variable name. You'll use this to reference the bean elsewhere
- `class="com.example.PetrolEngine"` - The fully qualified class name. Spring uses reflection to call `new PetrolEngine()` behind the scenes
- `/>` - Self-closing tag because this bean has no dependencies

**What Spring does**:

1. Uses reflection to load the `PetrolEngine` class
2. Calls the default constructor: `new PetrolEngine()`

3. Stores the object in the container with id "petrolEngine"

```xml
<bean id="electricEngine" class="com.example.ElectricEngine"/>
```

- Same concept - creates an ElectricEngine bean
- Now Spring's container has two Engine objects

---

**Constructor Injection in XML**:

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

- `<bean id="car"...>` - Define a Car bean
- `<constructor-arg>` - This tells Spring: "Pass this as an argument to the constructor"
- `ref="petrolEngine"` - Don't pass a new object, pass a REFERENCE to the existing "petrolEngine" bean

**What Spring does**:

1. Looks up the "petrolEngine" bean in the container
2. Calls `new Car(petrolEngine)` - passing the existing Engine object
3. Stores the Car in the container

**Equivalent Java code** Spring executes:

```java
Engine petrolEngine = container.get("petrolEngine");
Car car = new Car(petrolEngine);
```

---

**Setter Injection in XML**:

```xml
```

```
<bean id="driver" class="com.example.Driver">
    <property name="name" value="John"/>
    <property name="car" ref="car"/>
</bean>
```

- `<bean id="driver"...>` - Create a Driver bean
- `<property name="name" value="John"/>`:
  - `name="name"` - Refers to the setter method name. Spring will call `setName("John")`
  - `value="John"` - A literal string value (not a bean reference)
- `<property name="car" ref="car"/>`:
  - `name="car"` - Spring will call `setCar(...)`
  - `ref="car"` - Pass a reference to the existing "car" bean

**What Spring does**:

1. Calls `new Driver()` (default constructor)
2. Calls `driver.setName("John")`
3. Looks up the "car" bean
4. Calls `driver.setCar(car)`

**Equivalent Java code**:

```java
Driver driver = new Driver();
driver.setName("John");
Car car = container.get("car");
driver.setCar(car);
```

---

**Loading XML Configuration**:

```java
```

```
public class Application {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        Car car = context.getBean("car", Car.class);
        car.start();
    }
}
```

**Line-by-Line**:

```
java

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

- `ClassPathXmlApplicationContext` - A specific implementation of ApplicationContext that reads XML files
- `"applicationContext.xml"` - The XML file name. Spring looks for it in the classpath (usually `src/main/resources`)
- When this executes, Spring:
    1. Reads the XML file
    2. Creates all beans in dependency order (Engine first, then Car, then Driver)
    3. Injects dependencies as specified
    4. Stores everything in the context

```
java

Car car = context.getBean("car", Car.class);
```

- `getBean("car", Car.class)` - Get the bean with id "car" and cast it to Car type
- The second parameter `Car.class` provides type safety (no manual casting needed)

**Pros of XML Configuration**:

- Configuration separate from code (can change without recompilation)
- No code modification needed to change wiring
- Good for legacy systems

**Cons**:

- Verbose (lots of boilerplate)
- No compile-time checking (typos only caught at runtime)
- Refactoring is hard (if you rename a class, XML won't update automatically)
- Not modern (Java annotations are preferred now)

---

### 3.2 Java-Based Configuration

**Use Case**: Modern applications, type-safe configuration, better refactoring support

**Why Java Config is better**: Introduced in Spring 3.0, it provides compile-time checking and better IDE support.

```java
```

```java
@Configuration
public class AppConfig {

    @Bean
    public Engine petrolEngine() {
        return new PetrolEngine();
    }

    @Bean
    public Engine electricEngine() {
        return new ElectricEngine();
    }

    @Bean
    public Car car() {
        return new Car(petrolEngine());
    }

    @Bean
    public Driver driver(Car car) {
        Driver driver = new Driver();
        driver.setName("John");
        driver.setCar(car);
        return driver;
    }

    @Bean
    @Profile("production")
    public DataSource productionDataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://prod-db:5432/myapp");
        config.setUsername("prod_user");
        config.setPassword("prod_password");
        return new HikariDataSource(config);
    }
}
```

**Detailed Line-by-Line Explanation**:

```java
```

```
@Configuration
public class AppConfig {
```

- `@Configuration` - This annotation is CRITICAL. It tells Spring:
  - "This class contains bean definitions"
  - "Scan this class and execute @Bean methods"
  - "Apply special processing to @Bean methods"
- Without `@Configuration`, `@Bean` methods would just be regular methods
- `AppConfig` is just a regular Java class, but Spring treats it specially

---

**Simple Bean Definition**:

```
java

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

- `@Bean` - Tells Spring: "The object returned by this method should be a Spring bean"
- `public Engine petrolEngine()`:
  - `Engine` - The return type. Spring registers this bean as type Engine
  - `petrolEngine()` - The method name becomes the bean name (like `id="petrolEngine"` in XML)
- `return new PetrolEngine();` - This is how the bean is created

**What Spring does**:

1. Detects this method is annotated with `@Bean`
2. Calls the method: `Engine bean = appConfig.petrolEngine()`
3. Stores the result in the container with name "petrolEngine" and type Engine
4. By default, the method is called ONLY ONCE (singleton scope)

**Why is this better than XML?**

- Type-safe: If PetrolEngine doesn't exist, you get a compile error

- Refactoring: If you rename PetrolEngine, IDE updates this automatically
- Debugging: You can set breakpoints in this method

---

**Multiple Beans of Same Type**:

```java
@Bean
public Engine electricEngine() {
    return new ElectricEngine();
}
```

- We now have TWO beans of type Engine: petrolEngine and electricEngine
- Spring differentiates them by name
- If someone asks for Engine by type alone, Spring won't know which one to give (ambiguity)
- To resolve: use `@Qualifier` or `@Primary` (we'll see this later)

---

**Bean with Dependency (Method Call)**:

```java
@Bean
public Car car() {
    return new Car(petrolEngine());
}
```

- `new Car(petrolEngine())` - We call the `petrolEngine()` method directly
- **IMPORTANT**: Even though we're calling `petrolEngine()` as a regular method, Spring intercepts this call!

**What Spring does (magic behind the scenes)**:

1. You write: `petrolEngine()`
2. Spring intercepts and thinks: "Has petrolEngine() been called before?"
3. If YES: Return the cached instance (singleton)
4. If NO: Execute the method, cache the result, return it

5. This ensures the same Engine instance is used everywhere

**This is called CGLIB proxying**:

- Spring creates a subclass of AppConfig at runtime
- Overrides @Bean methods to add caching logic
- That's why `@Configuration` is required

Without `@Configuration`, `petrolEngine()` would create a NEW PetrolEngine every time it's called!

---

**Bean with Method Parameter (Automatic Injection)**:

```java
@Bean
public Driver driver(Car car) {
    Driver driver = new Driver();
    driver.setName("John");
    driver.setCar(car);
    return driver;
}
```

- `public Driver driver(Car car)` - Notice the parameter `Car car`
- **Spring automatically injects this parameter!**

**What Spring does**:

1. Sees the `driver()` method needs a Car parameter
2. Looks in the container for a bean of type Car
3. Finds the `car` bean from the `car()` method
4. Calls: `driver(carBean)` - passing the existing Car instance

**This is method parameter injection** - very clean and explicit about dependencies!

**The bean creation steps**:

```java
```

```
Driver driver = new Driver();  // Create new Driver
driver.setName("John");        // Set property
driver.setCar(car);            // Inject dependency (car parameter)
return driver;                 // Return to Spring to manage
```

---

**Conditional Bean Creation with Profiles**:

java

```
@Bean
@Profile("production")
public DataSource productionDataSource() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl("jdbc:postgresql://prod-db:5432/myapp");
    config.setUsername("prod_user");
    config.setPassword("prod_password");
    return new HikariDataSource(config);
}
```

- `@Profile("production")` - This bean is ONLY created when the "production" profile is active
- Profiles let you have different configurations for different environments

**What Spring does**:

1. Checks active profiles (set via `spring.profiles.active=production`)
2. If "production" is active: Create this bean
3. If "production" is NOT active: Skip this bean entirely

**Why is this useful?**

java

```
@Bean
@Profile("production")
public DataSource productionDataSource() {
    // Real production database
    return new HikariDataSource(...);
}

@Bean
@Profile("development")
public DataSource devDataSource() {
    // Local H2 database for development
    return new EmbeddedDatabaseBuilder().build();
}
```

- Same method name or type
- Different implementations based on environment
- No need for if/else statements in your code!

---

**Loading Java Configuration**:

```
java

public class Application {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfig.class);

        Car car = context.getBean(Car.class);
        Driver driver = context.getBean(Driver.class);
    }
}
```

**Line-by-Line**:

```
java

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

- `AnnotationConfigApplicationContext` - This implementation reads Java configuration classes (not XML)

- `AppConfig.class` - The configuration class to scan
- Spring will:
    1. Create an instance of AppConfig
    2. Scan for `@Bean` methods
    3. Execute them in dependency order
    4. Store all beans in the context

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

- `getBean(Car.class)` - Get bean by type (no need for bean name)
- Works because there's only ONE Car bean
- If there were multiple Car beans, this would throw an exception

**Pros of Java Configuration**:

- Type-safe (compile-time checking)
- Refactoring-friendly (IDE support)
- Can use Java logic (if statements, loops, etc.)
- Better debugging (can set breakpoints)
- Modern and recommended approach

**Cons**:

- Requires recompilation to change configuration
- Less separation of concerns (config mixed with code)

---

### 3.3 Annotation-Based Configuration

**Use Case**: Rapid development, minimal configuration, component scanning

**Philosophy**: Instead of defining beans in a central configuration file, annotate classes directly. Spring scans packages and auto-discovers beans.

java

```
@Component
public class PetrolEngine implements Engine {
    public void ignite() {
        System.out.println("Petrol engine started");
    }
}

@Component
public class Car {
    private Engine engine;

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

    public void start() {
        engine.ignite();
    }
}

@Service
public class CarService {

    @Autowired
    private Car car;

    public void startCar() {
        car.start();
    }
}

@Repository
public class CarRepository {

    @Autowired
    private DataSource dataSource;

    public void saveCar(Car car) {
        // Database operations
    }
}
```

```
@Controller
public class CarController {

    @Autowired
    private CarService carService;

    public void handleRequest() {
        carService.startCar();
    }
}
```

**Detailed Line-by-Line Explanation**:

**@Component - Basic Spring Bean**:

```
java

@Component
public class PetrolEngine implements Engine {
```

- `@Component` - This is the most generic Spring stereotype annotation. It tells Spring:
  - "This class should be a Spring bean"
  - "Create an instance of this class and manage it"
  - "Scan and register this during component scanning"
- `public class PetrolEngine` - A regular class, but now Spring-managed

**What Spring does**:

1. During startup, scans packages for `@Component` classes
2. Finds PetrolEngine
3. Calls `new PetrolEngine()` using reflection
4. Registers it as a bean with name "petrolEngine" (lowercase class name by default)

**No explicit @Bean method needed!** The annotation on the class itself is enough.

---

**@Autowired - Constructor Injection**:

```
java
```

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

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

- `@Component` - Car is a Spring bean
- `@Autowired` on constructor - Tells Spring: "Inject dependencies through this constructor"

**What Spring does**:

1. Sees Car needs an Engine parameter
2. Looks in the container for a bean of type Engine
3. Finds PetrolEngine (because PetrolEngine implements Engine)
4. Calls `new Car(petrolEngine)` - passing the found bean
5. Registers the Car bean

**Note**: In Spring 4.3+, if there's only ONE constructor, `@Autowired` is optional! Spring automatically uses it.

---

**Stereotype Annotations - Specialized @Component**:

**@Service - Business Logic Layer**:

```
java

@Service
public class CarService {

   @Autowired
   private Car car;
```

- `@Service` - Technically identical to `@Component`, but with semantic meaning
- Indicates: "This class contains business logic"
- Spring treats it the same as `@Component`, but:

- Developers know this is a service layer class
- Future Spring versions might add service-specific features
- Better code organization and clarity

- `@Autowired private Car car;` **- Field injection**
  - Spring directly injects into the field using reflection
  - No constructor or setter needed
  - We'll discuss why this is problematic later

**What Spring does**:

1. Creates CarService: `new CarService()`
2. Sees `@Autowired` on `car` field
3. Uses reflection to set: `carService.car = carBean`
4. Field is private but Spring bypasses access control

---

**@Repository - Data Access Layer**:

```
java

@Repository
public class CarRepository {

    @Autowired
    private DataSource dataSource;
```

- `@Repository` - Specialized `@Component` for data access classes
- **Additional feature**: Spring adds automatic exception translation
  - Database exceptions (SQLException) are translated to Spring's DataAccessException
  - Makes exception handling consistent across different databases

**Why the different name?**

- Follows the DAO (Data Access Object) pattern
- Clearly separates concerns: Repository = talks to database
- Exception translation: Spring wraps SQLException into DataAccessException

---

**@Controller - Presentation Layer**:

```java
@Controller
public class CarController {

    @Autowired
    private CarService carService;

    public void handleRequest() {
        carService.startCar();
    }
}
```

- `@Controller` - Specialized `@Component` for web controllers
- Used in Spring MVC for handling HTTP requests
- Spring adds additional processing for request mapping and response handling

**Architecture Flow**:

```
@Controller → @Service → @Repository → Database
(Web Layer) → (Business) → (Data Access) → (Storage)
```

Each annotation helps organize code into layers!

---

**Enabling Component Scanning**:

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
    // No need to manually define beans
    // Spring scans and registers @Component classes
}
```

**Line-by-Line**:

```java
@Configuration
```

- Still needed! Marks this as a configuration class

```java
@ComponentScan(basePackages = "com.example")
```

- `@ComponentScan` - Tells Spring: "Scan these packages for annotated classes"
- `basePackages = "com.example"` - The root package to scan
- Spring will recursively scan:
  - `com.example`
  - `com.example.controllers`
  - `com.example.services`
  - `com.example.repositories`
  - All subpackages...

**What Spring does**:

1. Scans `com.example` package
2. Finds all classes with `@Component`, `@Service`, `@Repository`, `@Controller`
3. Creates instances of each
4. Processes `@Autowired` annotations to inject dependencies
5. Registers all beans in the container

**Empty configuration class?**

- Yes! Because all beans are defined by annotations on classes
- The config class just triggers component scanning

---

**Comparison of Annotations**:

| Annotation | Purpose | Special Feature |
|---|---|---|
| @Component | Generic bean | None |
| @Service | Business logic | None (semantic only) |
| @Repository | Data access | Exception translation |
| @Controller | Web requests | Request mapping support |

All four create Spring beans, but the specialized ones provide semantic meaning and may have extra features.

---

### Configuration Comparison

Let's see the SAME bean configuration in all three approaches:

**Scenario**: Car needs Engine, CarService needs Car

**XML Configuration**:

```xml
<bean id="petrolEngine" class="com.example.PetrolEngine"/>
<bean id="car" class="com.example.Car">
    <constructor-arg ref="petrolEngine"/>
</bean>
<bean id="carService" class="com.example.CarService">
    <property name="car" ref="car"/>
</bean>
```

- 7 lines
- Verbose XML tags
- No compile-time checking

**Java Configuration**:

```java
```

```
@Configuration
public class AppConfig {
    @Bean
    public Engine petrolEngine() {
        return new PetrolEngine();
    }

    @Bean
    public Car car() {
        return new Car(petrolEngine());
    }

    @Bean
    public CarService carService(Car car) {
        CarService service = new CarService();
        service.setCar(car);
        return service;
    }
}
```

- 13 lines
- Type-safe
- Centralized
- Full control

**Annotation Configuration**:

```
java
```

```java
@Component
public class PetrolEngine implements Engine { }

@Component
public class Car {
    @Autowired
    public Car(Engine engine) { }
}

@Service
public class CarService {
    @Autowired
    private Car car;
}

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

- 6 lines of actual code (if we count annotations)
- Decentralized (annotations on each class)
- Minimal configuration
- Less control over wiring

---

## 4. Types of Injection

Spring supports three ways to inject dependencies. Each has specific use cases.

---

### 4.1 Constructor Injection ⭐ (Recommended)

**Mechanism**: Dependencies provided through constructor parameters

```java
```

```
public class Car {
    private final Engine engine;
    private final Transmission transmission;

    // Dependencies injected via constructor
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }

    public void start() {
        engine.ignite();
        transmission.engage();
    }
}
```

**Detailed Explanation**:

```
java

private final Engine engine;
```

- `private` - Encapsulation: only Car can access its engine
- `final` - **CRITICAL**: Once set, it cannot be changed
    - Immutability: Car's engine cannot be swapped after construction
    - Thread-safe: No need for synchronization
    - Clear intent: This is a required, permanent dependency

**Why final is important**:

```
java
```

```
// Without final
private Engine engine;

public void someMethod() {
    this.engine = null; // Oops! Bug introduced
}

// With final
private final Engine engine;

public void someMethod() {
    this.engine = null; // COMPILER ERROR! Cannot reassign final variable
}
```

---

```
java

public Car(Engine engine, Transmission transmission) {
```

- Constructor with TWO dependencies
- Parameters explicitly show what Car needs to function
- Anyone reading this code immediately knows Car's dependencies

**What this achieves**:

1. **Explicit dependencies**: Clear what's required
2. **Compile-time safety**: Cannot create Car without dependencies
3. **Complete initialization**: Car is fully ready after construction
4. **Immutability**: Dependencies cannot change

---

**XML Configuration for Constructor Injection**:

```
xml
```

```xml
<bean id="engine" class="com.example.PetrolEngine"/>
<bean id="transmission" class="com.example.ManualTransmission"/>

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

**Line-by-Line**:

```xml
xml

<constructor-arg ref="engine"/>
```

- `<constructor-arg>` - Pass this to the constructor
- `ref="engine"` - Pass the bean with id "engine"
- Order matters! First `<constructor-arg>` → first constructor parameter

**What if parameters are in wrong order?**

```xml
xml

<!-- Wrong order -->
<constructor-arg ref="transmission"/>
<constructor-arg ref="engine"/>
```

- Spring will try: `new Car(transmission, engine)`
- **Compile error!** Types don't match
- XML config fails at startup, not compile time

**Better approach with index**:

```xml
xml

<constructor-arg index="0" ref="engine"/>
<constructor-arg index="1" ref="transmission"/>
```

- Explicitly specify which parameter
- Less error-prone if constructor has multiple params of same type

---

**Java Configuration for Constructor Injection**:

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

@Bean
public Transmission transmission() {
    return new ManualTransmission();
}

@Bean
public Car car(Engine engine, Transmission transmission) {
    return new Car(engine, transmission);
}
```

**Line-by-Line**:

```java
public Car car(Engine engine, Transmission transmission) {
```

- Method parameters define dependencies
- Spring automatically injects them by type

**What Spring does**:

1. Sees `car()` method needs Engine and Transmission
2. Looks for beans of these types
3. Calls: `car(engineBean, transmissionBean)`

**Clean and type-safe!** If Engine bean doesn't exist, you get a compile error (if using proper IDE) or startup error.

---

**Annotation Configuration for Constructor Injection**:

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

    @Autowired // Optional in Spring 4.3+ if only one constructor
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

**Line-by-Line**:

```java
@Autowired
```

- Tells Spring: "Inject dependencies through this constructor"
- Spring 4.3+: If there's only ONE constructor, this is optional

**Why is @Autowired optional for single constructors?**

- Spring's philosophy: "If there's only one way to create the object, I'll use it"
- No ambiguity, so no annotation needed
- Less boilerplate code

**Multiple constructors (requires @Autowired)**:

```java
```

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

    public Car() {
        // Default constructor
    }

    @Autowired // Required! Tells Spring which constructor to use
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

---

**When to Use Constructor Injection**:

✅ **Required dependencies** (cannot be null)

java

```
public class Car {
    private final Engine engine; // MUST have engine

    public Car(Engine engine) {
        if (engine == null) {
            throw new IllegalArgumentException("Engine required!");
        }
        this.engine = engine;
    }
}
```

- Car cannot exist without an Engine
- Constructor enforces this rule
- Fail fast if dependency is missing

✅ **Immutable objects** (use final fields)

java

```
public class Car {
    private final Engine engine; // Cannot be changed

    public Car(Engine engine) {
        this.engine = engine;
    }
    // No setter for engine - immutable!
}
```

- Thread-safe by design
- No unexpected changes to dependencies

✅ **Better testability** (easy to pass mocks)

java

```
@Test
public void testCar() {
    // Easy to create mock
    Engine mockEngine = mock(Engine.class);

    // Inject through constructor
    Car car = new Car(mockEngine);

    // Test
    car.start();
    verify(mockEngine).ignite();
}
```

- No Spring context needed in tests
- Simple, fast unit tests
- No reflection magic

✅ **Circular dependency detection** (fails fast)

java

```
// Class A
public class A {
    public A(B b) { }
}

// Class B
public class B {
    public B(A a) { }
}
```

- A needs B, B needs A
- Constructor injection: Spring detects this at startup and throws exception
- Better than runtime errors!

---

**Pros of Constructor Injection**:

1. **Ensures complete initialization**

```
java

Car car = new Car(engine); // Car is fully ready
car.start(); // Safe to use immediately
```

2. **Supports immutability**

```
java

private final Engine engine; // Cannot change after construction
```

3. **Clear dependency requirements**

```
java

public Car(Engine e, Transmission t, Wheels w) {
    // Looking at constructor, I know Car needs 3 things
}
```

4. **Prevents partially constructed objects**

```java
// Impossible with constructor injection:
Car car = new Car();
car.setEngine(null); // Oops, forgot to set engine!
car.start(); // NullPointerException!
```

5. **Fail-fast behavior**

- If dependency is missing, application won't start
- Better than discovering bugs at runtime

---

**Cons of Constructor Injection**:

1. **Many dependencies lead to large constructors**

```java
public Car(Engine e, Transmission t, Wheels w,
           Brakes b, Seats s, Steering st,
           Dashboard d, Lights l, Battery bat) {
    // Constructor with 9 parameters - hard to read!
}
```

- **Solution**: This is actually a code smell! Your class is doing too much
- **Refactor**: Split into smaller classes

2. **Circular dependencies fail**

```java
// A needs B, B needs A - won't work!
```

- **Solution**: Redesign your architecture to avoid circular dependencies
- **Alternative**: Use setter injection for one of them (but this is a band-aid)

---

### 4.2 Setter Injection

**Mechanism**: Dependencies provided through setter methods

```java
public class Car {
    private Engine engine;
    private GPS gps; // Optional dependency

    // Required dependency
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    // Optional dependency
    public void setGps(GPS gps) {
        this.gps = gps;
    }

    public void start() {
        engine.ignite();
        if (gps != null) {
            gps.findRoute();
        }
    }
}
```

**Detailed Explanation**:

```java
private Engine engine;
```

- NOT final (unlike constructor injection)
- Can be changed after object creation
- Might be null initially

```java
```

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

- Standard JavaBean setter pattern: `setXxx`
- Spring recognizes this naming convention
- Called AFTER object construction

**Object lifecycle with setter injection**:

```
java

// 1. Object created
Car car = new Car(); // engine is null here!

// 2. Dependency injected
car.setEngine(engine); // Now engine is set

// 3. Object ready to use
car.start();
```

**The problem**: Between step 1 and 2, Car is in an incomplete state!

---

```
java

private GPS gps; // Optional dependency

public void setGps(GPS gps) {
    this.gps = gps;
}

public void start() {
    if (gps != null) { // Must check for null!
        gps.findRoute();
    }
}
```

**Key point**: With setter injection, dependencies might be null

- Must add null checks

- Optional dependencies are acceptable
- Required dependencies are risky

---

**XML Configuration for Setter Injection**:

```xml
<bean id="engine" class="com.example.PetrolEngine"/>
<bean id="gps" class="com.example.GPS"/>

<bean id="car" class="com.example.Car">
   <property name="engine" ref="engine"/>
   <property name="gps" ref="gps"/>
</bean>
```

**Line-by-Line**:

```xml
<property name="engine" ref="engine"/>
```

- `<property>` - Use setter injection (not constructor)
- `name="engine"` - Calls `setEngine()` method
  - Spring converts "engine" → "setEngine"
  - First letter capitalized, "set" prefix added
- `ref="engine"` - Pass the bean with id "engine"

**What Spring does**:

```java
Car car = new Car();                // 1. Create object
car.setEngine(engineBean);          // 2. Call setter with bean
car.setGps(gpsBean);                // 3. Call another setter
```

---

**Java Configuration for Setter Injection**:

```java
@Bean
public Car car(Engine engine) {
    Car car = new Car();
    car.setEngine(engine);      // Explicit setter call
    car.setGps(new GPS());      // Optional dependency
    return car;
}
```

**Line-by-Line**:

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

- Create Car using default (no-arg) constructor
- Car is created in incomplete state

```java
car.setEngine(engine);
```

- Manually call setter to inject dependency
- More verbose than constructor injection
- But gives you control over the process

**Why use this approach?**

```java
```

```
@Bean
public Car car(Engine engine) {
    Car car = new Car();
    car.setEngine(engine);

    // Can add logic here!
    if (productionMode) {
        car.setGps(new PremiumGPS());
    } else {
        car.setGps(new BasicGPS());
    }

    return car;
}
```

- Conditional dependency injection
- More flexibility in bean creation

---

**Annotation Configuration for Setter Injection**:

```
java

@Component
public class Car {
    private Engine engine;

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

    @Autowired(required = false) // Optional dependency
    public void setGps(GPS gps) {
        this.gps = gps;
    }
}
```

**Line-by-Line**:

```
java
```

```
@Autowired
public void setEngine(Engine engine) {
```

- `@Autowired` on setter method
- Spring calls this method after creating the Car object
- Passes the Engine bean as parameter

```
java

@Autowired(required = false)
public void setGps(GPS gps) {
```

- `required = false` - **CRITICAL** parameter
- Tells Spring: "If GPS bean doesn't exist, skip this setter"
- Without this, missing GPS bean would cause application to fail at startup

**What required=false means**:

```
java

// GPS bean exists
@Autowired(required = false)
public void setGps(GPS gps) {
    // Spring calls this with GPS bean
}

// GPS bean doesn't exist
@Autowired(required = false)
public void setGps(GPS gps) {
    // Spring skips this method - NOT called
    // gps field remains null
}

// Without required=false and GPS missing:
// Application startup FAILS with exception
```

---

**When to Use Setter Injection**:

✅ **Optional dependencies** (can be null)

```java
@Autowired(required = false)
public void setGps(GPS gps) {
    this.gps = gps; // Might be null - that's OK!
}

public void navigate() {
    if (gps != null) { // Must check!
        gps.findRoute();
    } else {
        System.out.println("Manual navigation");
    }
}
```

✅ **Dependencies that may change** after object creation

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

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

    // Can swap engine later
    public void upgradeEngine(Engine newEngine) {
        this.engine = newEngine;
    }
}
```

- Not possible with constructor injection (`final` prevents this)
- Useful for hot-swapping components

✅ **Circular dependencies** (Spring can handle via setters)

```java
```

```
@Component
public class A {
    private B b;

    @Autowired
    public void setB(B b) { this.b = b; }
}

@Component
public class B {
    private A a;

    @Autowired
    public void setA(A a) { this.a = a; }
}
```

**How Spring handles this**:

1. Create A: `A a = new A()` (b is null)
2. Create B: `B b = new B()` (a is null)
3. Call `a.setB(b)` (now A has reference to B)
4. Call `b.setA(a)` (now B has reference to A)
5. Circular dependency resolved!

**With constructor injection, this would fail**:

- To create A, need B
- To create B, need A
- Chicken and egg problem!

✅ **Large number of dependencies** (avoid constructor bloat)

```
java
```

```
// Constructor with 10 parameters - hard to read
public Car(Engine e, Trans t, Wheels w, Brakes b, ...) { }

// vs Setters - each dependency is clear
public void setEngine(Engine e) { }
public void setTransmission(Trans t) { }
public void setWheels(Wheels w) { }
```

✅ **Default values** for some dependencies

java

```
@Component
public class Car {
    private GPS gps = new BasicGPS(); // Default

    @Autowired(required = false)
    public void setGps(GPS gps) {
        this.gps = gps; // Override default if premium GPS available
    }
}
```

---

**Pros of Setter Injection**:

1. **Supports optional dependencies**

java

```
@Autowired(required = false) // Won't fail if missing
```

2. **Can change dependencies after construction**

java

```
car.setEngine(newEngine); // Hot swap
```

3. **Handles circular dependencies better**

java

```
// A and B can reference each other
```

4. **More readable with many dependencies**

```
java

// 10 setters vs 10-parameter constructor
```

---

**Cons of Setter Injection**:

1. **Object can be in partially initialized state**

```
java

Car car = new Car(); // Created but engine is null!
car.start(); // NullPointerException if engine not set yet
```

2. **Cannot use final fields**

```
java

private final Engine engine; // IMPOSSIBLE with setter injection
// final requires initialization in constructor
```

3. **Less clear which dependencies are required**

```
java

// Looking at setters, which are required vs optional?
public void setEngine(Engine e) { }  // Required?
public void setGps(GPS g) { }        // Optional?
// Not clear without reading documentation!
```

4. **Must remember to call all setters**

```
java
```

```
Car car = new Car();
car.setEngine(engine);
// Forgot to set transmission!
car.start(); // Might fail later
```

---

### 4.3 Field Injection ⚠️ (Generally Discouraged)

**Mechanism**: Dependencies injected directly into fields using reflection

```
java

@Component
public class Car {

    @Autowired
    private Engine engine;

    @Autowired(required = false)
    private GPS gps;

    // No constructor or setter needed!

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

**Detailed Explanation**:

```
java

@Autowired
private Engine engine;
```

- `@Autowired` directly on the field
- `private` - Field is private, but Spring uses reflection to access it
- No constructor or setter needed
- Looks very clean and simple!

**What Spring does**:

```java
// 1. Create object
Car car = new Car();

// 2. Use reflection to set private field
Field field = Car.class.getDeclaredField("engine");
field.setAccessible(true); // Bypass private access control
field.set(car, engineBean); // Directly set the field value
```

**Reflection bypasses normal Java access control**:

```java
// Normal Java:
car.engine = engineBean; // COMPILER ERROR! engine is private

// Spring with reflection:
field.set(car, engineBean); // Works! Reflection ignores private
```

---

**Why Field Injection Looks Appealing**:

```java
```

```java
// Constructor injection - more code
@Component
public class Car {
    private final Engine engine;

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

// Field injection - less code
@Component
public class Car {
    @Autowired
    private Engine engine;
}
```

**Fewer lines = must be better, right?**

**NO! Let's see why field injection is problematic...**

---

**Only Available With**: Annotations (requires reflection)

**Not possible in XML or Java Config** because:

- XML/Java config create objects normally
- Field injection requires reflection magic
- Only annotation processing can inject into private fields

---

**When People Use Field Injection**:

⚠️ **Quick prototypes**

```java
```

```
// Fast to write, minimal boilerplate
@Autowired private Engine engine;
@Autowired private Trans transmission;
@Autowired private Wheels wheels;
```

⚠️ **Simple applications** with few dependencies

```
java

// Small app, not concerned about testing
@Autowired private UserService userService;
```

⚠️ **When you understand the trade-offs**

- Know it's harder to test
- Accept tight coupling to Spring
- Aware of limitations

**But in professional/production code**: Constructor injection is strongly preferred!

---

**Why Field Injection is Problematic**:

❌ **Problem 1: Cannot use final fields**

```
java

@Autowired
private final Engine engine; // COMPILER ERROR!
// final requires initialization at declaration or in constructor
```

- Loses immutability benefits
- Can't prevent accidental changes
- Less thread-safe

❌ **Problem 2: Hard to test**

```
java
```

```java
// Unit test WITHOUT Spring
@Test
public void testCar() {
    Car car = new Car();

    // How do I inject engine???
    // car.engine = mockEngine; // PRIVATE! Can't access

    // Need reflection (ugly!)
    Field field = Car.class.getDeclaredField("engine");
    field.setAccessible(true);
    field.set(car, mockEngine);

    car.start();
}

// vs Constructor injection (clean!)
@Test
public void testCar() {
    Engine mockEngine = mock(Engine.class);
    Car car = new Car(mockEngine); // Simple!
    car.start();
}
```

**Testing field injection requires**:

- Spring test context (slower tests)
- Or reflection (ugly, brittle code)
- Or special testing frameworks

### ❌ Problem 3: Hides dependencies

java

```
// Constructor injection - dependencies are obvious
public Car(Engine e, Trans t, Wheels w, Brakes b) {
    // Looking at constructor, I see Car needs 4 things
}

// Field injection - dependencies are hidden
public class Car {
    @Autowired private Engine engine;
    @Autowired private Transmission transmission;
    @Autowired private Wheels wheels;
    @Autowired private Brakes brakes;

    // No visual cue about dependencies until you read entire class
}
```

**Why this matters**:

- Can't see dependencies at a glance
- Easy to add more and more dependencies
- Class grows without noticing (violates Single Responsibility Principle)

### ❌ Problem 4: NullPointerException if Spring context not loaded

java

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

    public void start() {
        engine.ignite(); // NullPointerException if not in Spring context!
    }
}

// Usage outside Spring:
Car car = new Car(); // Created without Spring
car.start(); // BOOM! engine is null
```

**With constructor injection**:

java

```
Car car = new Car(null); // Compiler or runtime error immediately!
// Fail fast!
```

### ❌ Problem 5: Tight coupling to Spring framework

```
java

// Field injection - requires Spring
@Autowired
private Engine engine; // Can't work without Spring

// Constructor injection - works anywhere
public Car(Engine engine) {
    this.engine = engine;
}

// Can use without Spring:
Engine engine = new PetrolEngine();
Car car = new Car(engine); // Works fine!
```

### ❌ Problem 6: Cannot inject dependencies manually

```
java

// Want to create Car manually?
Car car = new Car();
// Now what? engine is private, no way to set it!

// vs Constructor:
Car car = new Car(new PetrolEngine()); // Easy!
```

---

**Field Injection Comparison Table**:

| Scenario | Field Injection | Constructor Injection |
|---|---|---|
| Unit Testing | ❌ Needs reflection or Spring context | ✅ Simple object creation |
| Immutability | ❌ Cannot use `final` | ✅ Can use `final` |
| Dependency Visibility | ❌ Hidden in class body | ✅ Visible in constructor |
| Null Safety | ❌ Fields can be null | ✅ Constructor ensures not null |
| Manual Creation | ❌ Impossible to inject manually | ✅ Easy to create manually |
| Framework Coupling | ❌ Tightly coupled to Spring | ✅ Works with or without Spring |
| Code Brevity | ✅ Very concise | ⚠️ More verbose |

---

**Why Spring Still Allows Field Injection**:

1. **Backwards compatibility -** Removing it would break existing code
2. **Convenience for demos/prototypes -** Fast to write for small examples
3. **Developer choice -** Spring trusts developers to make informed decisions

**But Spring's official documentation recommends constructor injection!**

---

**Injection Type Comparison Summary**

| Feature | Constructor | Setter | Field |
|---|---|---|---|
| **Required Dependencies** | ✅ Best | ⚠️ Possible | ⚠️ Possible |
| **Optional Dependencies** | ❌ Verbose | ✅ Best | ✅ Good |
| **Immutability (final)** | ✅ Yes | ❌ No | ❌ No |
| **Testability** | ✅ Excellent | ✅ Good | ❌ Poor |
| **Null Safety** | ✅ Best | ⚠️ Must check | ⚠️ Risky |
| **Circular Dependencies** | ❌ Fails | ✅ Works | ✅ Works |

| Feature | Constructor | Setter | Field |
|---|---|---|---|
| **Code Clarity** | ✅ Clear | ✅ Clear | ⚠️ Hidden |
| **Boilerplate** | ⚠️ Medium | ⚠️ Medium | ✅ Minimal |
| **Framework Coupling** | ✅ Loosely coupled | ✅ Loosely coupled | ❌ Tightly coupled |

---

## Practical Recommendation

### Decision Tree for Choosing Injection Type:

Is the dependency REQUIRED for the class to function?
├─ YES → Use Constructor Injection
│ ✅ Ensures dependency is provided
│ ✅ Supports immutability (final)
│ ✅ Clear and testable
│
└─ NO → Is it truly optional?
├─ YES → Use Setter Injection
│ ✅ @Autowired(required = false)
│ ✅ Can provide default value
│ ✅ Can change after initialization
│
└─ Avoid Field Injection unless:
⚠️ Quick prototype only
⚠️ You understand the trade-offs
❌ Never in production if possible

---

### Best Practice Priority:

**1. Constructor Injection** (Default Choice) ⭐⭐⭐⭐⭐

java

```
@Component
public class Car {
    private final Engine engine; // Required, immutable

    @Autowired // Optional in Spring 4.3+
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

**When**:

- Dependency is required
- Want immutability
- Need better testability
- **Use this 90% of the time**

---

**2. Setter Injection** (Secondary Choice) ⭐⭐⭐

```
java

@Component
public class Car {
    private GPS gps; // Optional

    @Autowired(required = false)
    public void setGps(GPS gps) {
        this.gps = gps;
    }
}
```

**When**:

- Dependency is truly optional
- Dependency might change after creation
- Need to handle circular dependencies
- **Use this 10% of the time**

---

**3. Field Injection** (Avoid) ⭐

```java
@Component
public class Car {
    @Autowired
    private Engine engine; // Avoid this!
}
```

**When**:

- Quick prototype or demo
- Simple Spring Boot tutorial
- **Never in production code**
- **Use this 0% of the time in serious projects**

---

## Complete Real-World Example

Let's build an e-commerce order processing system with all concepts:

```java
```

```java
// ==================== DOMAIN LAYER ====================

// Interface for payment processing
public interface PaymentProcessor {
    PaymentResult processPayment(double amount, PaymentDetails details);
}

// Concrete implementation
@Component
public class StripePaymentProcessor implements PaymentProcessor {

    @Override
    public PaymentResult processPayment(double amount, PaymentDetails details) {
        // Simulate Stripe API call
        System.out.println("Processing $" + amount + " via Stripe");
        return new PaymentResult(true, "txn_" + System.currentTimeMillis());
    }
}

// ==================== REPOSITORY LAYER ====================

@Repository // Data access layer
public class OrderRepository {

    private final DataSource dataSource; // Database connection

    // Constructor injection - dataSource is required
    @Autowired
    public OrderRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void saveOrder(Order order) {
        // Save to database
        System.out.println("Saving order: " + order.getId());
    }

    public Order findById(String orderId) {
        // Retrieve from database
        return new Order(orderId);
    }
}
```

```
// ==================== SERVICE LAYER ====================

@Service // Business logic layer
public class OrderService {

    // Required dependencies - constructor injection
    private final PaymentProcessor paymentProcessor;
    private final OrderRepository orderRepository;
    private final EmailService emailService;

    // Optional dependency - setter injection
    private NotificationService notificationService;

    /**
     * Constructor injection for REQUIRED dependencies
     * All three services are essential for order processing
     */
    @Autowired
    public OrderService(
            PaymentProcessor paymentProcessor,
            OrderRepository orderRepository,
            EmailService emailService) {

        this.paymentProcessor = paymentProcessor;
        this.orderRepository = orderRepository;
        this.emailService = emailService;
    }

    /**
     * Setter injection for OPTIONAL dependency
     * SMS notifications are nice-to-have, not essential
     */
    @Autowired(required = false)
    public void setNotificationService(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    /**
     * Main business logic - process an order
     * This method orchestrates the entire workflow
     */
    public OrderResult placeOrder(Order order) {
        // 1. Process payment
        PaymentResult paymentResult = paymentProcessor.processPayment(
```

```
            order.getTotal(),
            order.getPaymentDetails()
        );

        if (!paymentResult.isSuccess()) {
            return OrderResult.failure("Payment failed");
        }

        // 2. Save to database
        orderRepository.saveOrder(order);

        // 3. Send email confirmation (required)
        emailService.sendOrderConfirmation(order);

        // 4. Send SMS notification (optional - might be null)
        if (notificationService != null) {
            notificationService.sendSMS(
                order.getCustomerPhone(),
                "Your order " + order.getId() + " is confirmed!"
            );
        }

        return OrderResult.success(order.getId());
    }
}

@Service
public class EmailService {

    private final MailSender mailSender;

    @Autowired
    public EmailService(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void sendOrderConfirmation(Order order) {
        System.out.println("Sending email to: " + order.getCustomerEmail());
    }
}

@Service
public class NotificationService {
```

```
    private final SMSGateway smsGateway;

    @Autowired
    public NotificationService(SMSGateway smsGateway) {
        this.smsGateway = smsGateway;
    }

    public void sendSMS(String phone, String message) {
        System.out.println("Sending SMS to " + phone + ": " + message);
    }
}

// ==================== CONFIGURATION ====================

@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    /**
     * DataSource bean - database connection pool
     * This is infrastructure, so we define it in configuration
     */
    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://localhost:5432/orders");
        config.setUsername("admin");
        config.setPassword("password");
        config.setMaximumPoolSize(10);
        return new HikariDataSource(config);
    }

    /**
     * MailSender bean for email functionality
     */
    @Bean
    public MailSender mailSender() {
        return new MailSender("smtp.gmail.com", 587);
    }

    /**
     * SMSGateway is optional - only create in production
     */
    @Bean
```

```
    @Profile("production")
    public SMSGateway smsGateway() {
        return new TwilioSMSGateway("account_sid", "auth_token");
    }

    /**
     * Different payment processors for different environments
     */
    @Bean
    @Profile("production")
    public PaymentProcessor stripeProcessor() {
        return new StripePaymentProcessor();
    }

    @Bean
    @Profile("development")
    public PaymentProcessor mockProcessor() {
        return new MockPaymentProcessor(); // Fake processor for dev
    }
}

// ==================== MAIN APPLICATION ====================

public class ECommerceApplication {

    public static void main(String[] args) {
        // 1. Create Spring container with configuration
        ApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfig.class);

        // 2. Retrieve the fully-wired OrderService
        OrderService orderService = context.getBean(OrderService.class);

        // 3. Create an order
        Order order = new Order("ORD-001");
        order.setTotal(99.99);
        order.setCustomerEmail("customer@example.com");
        order.setCustomerPhone("+1234567890");
        order.setPaymentDetails(new PaymentDetails("card_token_123"));

        // 4. Process the order
        OrderResult result = orderService.placeOrder(order);

        if (result.isSuccess()) {
```

```
            System.out.println("Order placed successfully: " + result.getOrderId());
        } else {
            System.out.println("Order failed: " + result.getMessage());
        }
    }
}
```

---

**Line-by-Line Explanation of the Complete Example:**

**OrderService - The Heart of the Application**:

```
java

@Service
public class OrderService {
```

- `@Service` - Marks this as a Spring-managed service layer component
- Contains the core business logic for order processing

---

```
java

    private final PaymentProcessor paymentProcessor;
    private final OrderRepository orderRepository;
    private final EmailService emailService;
```

- Three REQUIRED dependencies declared as `final`
- `final` ensures they cannot be null or changed after construction
- All three are necessary - an order cannot be processed without them

---

```
java

    private NotificationService notificationService;
```

- Optional dependency - NOT final
- Can be null (SMS is nice-to-have, not essential)

- Will be injected via setter if available

---

```java
@Autowired
public OrderService(
        PaymentProcessor paymentProcessor,
        OrderRepository orderRepository,
        EmailService emailService) {
```

- **Constructor injection for required dependencies**
- Spring will look for beans of these three types
- Application won't start if any are missing (fail-fast)
- All parameters passed by Spring automatically

**What Spring does**:

1. Finds PaymentProcessor bean → finds StripePaymentProcessor
2. Finds OrderRepository bean → creates it (with DataSource)
3. Finds EmailService bean → creates it (with MailSender)
4. Calls: `new OrderService(stripeProcessor, orderRepo, emailService)`

---

```java
        this.paymentProcessor = paymentProcessor;
        this.orderRepository = orderRepository;
        this.emailService = emailService;
```

- Assigns dependencies to final fields
- After this, OrderService is fully initialized and ready
- Dependencies cannot be changed (immutability)

---

```java
```

```
@Autowired(required = false)
public void setNotificationService(NotificationService notificationService) {
    this.notificationService = notificationService;
}
```

- **Setter injection for optional dependency**
- `required = false` - Key difference! If NotificationService bean doesn't exist, this setter won't be called
- Application continues working without SMS notifications

**What Spring does**:

1. Checks if NotificationService bean exists
2. If YES: Calls `orderService.setNotificationService(notificationBean)`
3. If NO: Skips this method, `notificationService` remains null

---

```
java

    public OrderResult placeOrder(Order order) {
```

- Main business method - public API of this service
- Called by controllers or other services

---

```
java

        PaymentResult paymentResult = paymentProcessor.processPayment(
            order.getTotal(),
            order.getPaymentDetails()
        );
```

- Use injected PaymentProcessor
- Don't care if it's Stripe, PayPal, or Mock - polymorphism!
- In production: StripePaymentProcessor
- In development: MockPaymentProcessor

---

```java
if (!paymentResult.isSuccess()) {
    return OrderResult.failure("Payment failed");
}
```

- Business logic: Only proceed if payment succeeds
- Fail fast - don't save order if payment failed

---

```java
orderRepository.saveOrder(order);
```

- Use injected repository to persist order
- Repository handles database complexity

---

```java
emailService.sendOrderConfirmation(order);
```

- Required service - we know it's not null (constructor injection)
- No null check needed

---

```java
if (notificationService != null) {
    notificationService.sendSMS(...);
}
```

- **MUST check for null** because this is optional (setter injection)
- If SMS gateway not configured, order processing still succeeds
- Graceful degradation

---

**Configuration Class**:

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
```

- `@Configuration` - Spring configuration class
- `@ComponentScan` - Automatically discovers @Component, @Service, @Repository classes
- Combines annotation-based and Java-based config

---

```java
    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://localhost:5432/orders");
        config.setUsername("admin");
        config.setPassword("password");
        config.setMaximumPoolSize(10);
        return new HikariDataSource(config);
    }
```

- **Explicitly defined bean** (not @Component)
- Why? Because:
  - DataSource is a third-party class (HikariCP library)
  - We can't add @Component to classes we don't own
  - Configuration requires specific setup
- Spring calls this method once, stores the DataSource in container
- OrderRepository will receive this via constructor injection

---

```java
```

```
@Bean
@Profile("production")
public SMSGateway smsGateway() {
    return new TwilioSMSGateway("account_sid", "auth_token");
}
```

- `@Profile("production")` - Only create this bean in production
- In development, this bean doesn't exist
- NotificationService won't be injected (required=false)
- Application still works without SMS in development!

**How to activate profiles**:

```
java -Dspring.profiles.active=production -jar app.jar
```

---

```
java

@Bean
@Profile("production")
public PaymentProcessor stripeProcessor() {
    return new StripePaymentProcessor();
}

@Bean
@Profile("development")
public PaymentProcessor mockProcessor() {
    return new MockPaymentProcessor();
}
```

- **Two different implementations, same interface**
- Production: Real Stripe API calls
- Development: Fake processor (no real charges)
- Spring chooses based on active profile
- OrderService doesn't know which one - just uses PaymentProcessor interface

---

**Main Application**:

```java
ApplicationContext context =
    new AnnotationConfigApplicationContext(AppConfig.class);
```

- Creates Spring container
- Loads AppConfig
- Scans com.example package for components
- Creates all beans in correct order
- Injects dependencies

**Bean creation order (Spring figures this out)**:

1. DataSource (no dependencies)
2. MailSender (no dependencies)
3. PaymentProcessor (no dependencies)
4. OrderRepository (needs DataSource)
5. EmailService (needs MailSender)
6. NotificationService (needs SMSGateway - might not exist)
7. OrderService (needs PaymentProcessor, OrderRepository, EmailService)

---

```java
OrderService orderService = context.getBean(OrderService.class);
```

- Retrieve the fully-initialized OrderService
- All dependencies already injected
- Ready to use immediately

---

```java
```

```
OrderResult result = orderService.placeOrder(order);
```

- Use the service normally
- Don't worry about dependencies - Spring handled everything

---

## Key Takeaways

### 1. Dependency Injection = Loose Coupling

- Objects don't create their dependencies
- Dependencies provided from outside
- Easy to swap implementations
- Better testability

### 2. Inversion of Control = Framework Takes Control

- You define WHAT you need
- Spring handles HOW to create and wire
- Focus on business logic, not plumbing

### 3. Configuration Approaches

**XML** (Legacy):

```
xml

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

- Verbose, external, no type safety
- Use for: Legacy apps

**Java Config** (Modern):

```
java
```

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

- Type-safe, refactor-friendly, centralized
- Use for: Complex setup, conditional logic

**Annotations** (Fastest):

```
java

@Component
public class Car {
    @Autowired
    public Car(Engine engine) { }
}
```

- Minimal code, auto-discovery
- Use for: 90% of your beans

### 4. Injection Types

**Constructor** ⭐⭐⭐⭐⭐ (Use this):

```
java

private final Engine engine;

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

- Required dependencies
- Immutable (final)
- Testable
- Clear

**Setter** ⭐⭐⭐ (When needed):

```java
@Autowired(required = false)
public void setGps(GPS gps) {
    this.gps = gps;
}
```

- Optional dependencies
- Can change after creation
- Circular dependencies

**Field** ⚠️ (Avoid):

```java
@Autowired
private Engine engine;
```

- Hard to test
- No immutability
- Hides dependencies
- Only for quick prototypes

**Golden Rule**

**"Constructor for required, Setter for optional, Never field in production!"**

---

## Summary Table

| Concept | What It Solves | How to Use |
|---|---|---|
| **DI** | Tight coupling | Inject dependencies from outside |
| **IoC** | Manual object management | Let Spring create and wire objects |
| **XML Config** | External configuration | Legacy apps, no recompilation |
| **Java Config** | Type-safe config | Modern apps, complex setup |

| Concept | What It Solves | How to Use |
|---|---|---|
| **Annotation Config** | Minimal boilerplate | Most beans, rapid development |
| **Constructor Injection** | Required dependencies | Use `final` fields, best practice |
| **Setter Injection** | Optional dependencies | Use `required=false` |
| **Field Injection** | Quick prototypes | Avoid in production |

This completes the comprehensive Spring DI and IoC notes. Each concept is explained from first principles with detailed code explanations!