Spring Framework

Introduction to Spring Framework:

The **Spring Framework** is a comprehensive, open-source Java framework designed to simplify enterprise application development by promoting modularity, loose coupling, and ease of testing. It provides a robust infrastructure for building scalable, maintainable applications, particularly for Java EE (Enterprise Edition). Unlike **Spring Boot**, which is a higher-level framework built on top of Spring to simplify configuration and setup, the Spring Framework focuses on providing core functionalities and flexibility, requiring more manual configuration.

Spring's philosophy is centered around **Inversion of Control (IoC)** and **Dependency Injection (DI)**, which allow developers to manage object creation and dependencies externally (typically via XML or Java-based configuration). It also emphasizes **Aspect-Oriented Programming (AOP)** for cross-cutting concerns like logging and transaction management.

Below is an explanation of the Spring Framework and its key components (modules):

Core Principles of Spring Framework

1. **Inversion of Control (IoC)**: The framework manages the creation and lifecycle of objects (beans). Developers define how objects are wired together, and the Spring IoC container handles instantiation and dependency injection.
2. **Dependency Injection (DI)**: Dependencies are injected into objects externally (via constructors, setters, or fields) rather than objects creating their own dependencies, promoting loose coupling.
3. **Aspect-Oriented Programming (AOP)**: Separates cross-cutting concerns (e.g., logging, security) from business logic by applying them modularly.
4. **Modularity**: Spring is divided into modules, allowing developers to use only the components they need.

Key Components (Modules) of Spring Framework

The Spring Framework is organized into several modules, grouped by functionality. These modules are part of the Spring distribution and can be included as dependencies in a project. Below are the primary modules:

1. Core Container

The core container is the foundation of the Spring Framework, providing IoC and DI capabilities.

- **spring-core**: Contains fundamental utilities and classes, including the BeanFactory (the basic IoC container).
- **spring-beans**: Provides the BeanFactory implementation and supports bean definition, creation, and management.
- **spring-context**: Extends spring-beans with features like application event handling, internationalization (i18n), and the ApplicationContext (a more advanced IoC container).
- **spring-expression**: Offers the Spring Expression Language (SpEL) for querying and manipulating objects at runtime.

2. AOP and Instrumentation

These modules enable aspect-oriented programming and class instrumentation.

- **spring-aop**: Implements AOP for cross-cutting concerns (e.g., logging, transaction management) using proxies or AspectJ integration.
- **spring-as-AspectJ**: Integrates with AspectJ for advanced AOP features (optional, not always required).
- **spring-instrument**: Provides class instrumentation and agent-based loading for specific use cases (e.g., monitoring).

3. Data Access/Integration

These modules simplify database access and integration with external systems.

- **spring-jdbc**: Provides a JDBC abstraction layer to simplify database operations and reduce boilerplate code.
- **spring-orm**: Integrates with Object-Relational Mapping (ORM) tools like Hibernate and JPA.
- **spring-tx**: Manages transactions programmatically or declaratively (e.g., via annotations like @Transactional).
- **spring-jms**: Supports Java Message Service (JMS) for messaging integration.
- **spring-oxm**: Enables Object-XML Mapping for XML serialization/deserialization.

4. Web

These modules support web application development.

- **spring-web**: Provides basic web features, including request handling, multipart file uploads, and integration with servlets.
- **spring-webmvc**: Implements the Model-View-Controller (MVC) pattern for building web applications.
- **spring-websocket**: Supports WebSocket-based communication for real-time applications.
- **spring-webflux** (introduced later): Offers reactive web programming (not part of the traditional Spring Framework but included in modern versions).

5. Test

- **spring-test**: Provides utilities for unit and integration testing, including mock objects and Spring context loading for tests.

6. Miscellaneous

- **spring-security** (optional): A separate module (not always bundled) for authentication, authorization, and security features.
- **spring-batch**: Supports batch processing for large-scale data operations (also a separate project but integrates with Spring).
- **spring-integration**: Facilitates enterprise integration patterns for messaging and system integration.

Key Features of Spring Framework

- **IoC Container**: Manages beans and their dependencies via ApplicationContext or BeanFactory.
- **AOP**: Allows modularization of cross-cutting concerns.
- **Transaction Management**: Simplifies declarative transaction management (e.g., @Transactional).
- **MVC Framework**: Provides a robust web framework for building RESTful APIs or traditional web apps.
- **JDBC/ORM Integration**: Reduces boilerplate code for database operations.
- **Testing Support**: Simplifies testing with mock objects and context loading.

How Spring Framework Works

1. **Configuration**: Developers define beans and their dependencies using:
    - **XML Configuration**: Legacy approach using XML files (e.g., applicationContext.xml).
    - **Java Configuration**: Modern approach using @Configuration classes and @Bean annotations.
    - **Annotation-Based Configuration**: Uses annotations like @Component, @Autowired, and @Service for automatic bean discovery and injection.

2. **IoC Container**: The ApplicationContext reads the configuration, creates beans, and injects dependencies.
3. **Application Logic**: Developers focus on business logic, while Spring handles infrastructure concerns like transactions, security, and logging.

Spring Framework vs. Spring Boot

- **Spring Framework**:
  - Requires manual configuration (XML or Java-based).
  - Offers fine-grained control over the application setup.
  - Suitable for complex, highly customized applications.
- **Spring Boot**:
  - Built on top of Spring Framework.
  - Provides auto-configuration, embedded servers (e.g., Tomcat), and starter dependencies to reduce setup time.
  - Ideal for rapid development and microservices.

Inversion of Control (IoC) and **Dependency Injection (DI):**
Inversion of Control (IoC) and Dependency Injection (DI) are core principles of the Spring Framework that promote loose coupling, modularity, and easier testing in applications.

what is loose and tight coupling?
Loose coupling and **tight coupling** are concepts in software engineering that describe the degree of dependency between components, classes, or modules in a system. These terms are particularly relevant in the context of object-oriented programming, design patterns, and frameworks like Spring, where principles like Inversion of Control (IoC) and Dependency Injection (DI) aim to achieve loose coupling.

Tight Coupling
Definition: Tight coupling occurs when two or more components (e.g., classes, modules) are highly dependent on each other, such that a change in one component directly affects the other. In tightly coupled systems, components often have direct knowledge of each other's implementation details.
Characteristics:

- Components directly create or instantiate their dependencies.
- Classes reference concrete implementations rather than abstractions (e.g., interfaces or abstract classes).
- Changes to one component (e.g., renaming a method, changing a class's structure) often require changes in dependent components.
- Difficult to test because dependencies are hardcoded, making it hard to mock or replace them.

Drawbacks:

- Reduced flexibility: Hard to swap or modify components without affecting others.
- Poor maintainability: Changes ripple through the system, increasing maintenance effort.
- Difficult testing: Unit tests require the real dependency, which may involve complex setup or external resources.
- Code is less reusable because components are tied to specific implementations.

Example of Tight Coupling:

```
public class MessageService {
   public String getMessage() {
      return "Hello, World!";
   }
}

public class MessagePrinter {
   private MessageService messageService = new MessageService ();

   public void printMessage() {
      System.out.println(messageService.getMessage());
   }

   public static void main(String[] args) {
      MessagePrinter printer = new MessagePrinter();
      printer.printMessage(); // Output: Hello, World!
   }
}
```

- **Why It's Tightly Coupled**:
  - MessagePrinter directly instantiates MessageServiceImpl, tying it to a specific implementation.
  - **If you want to use a different MessageService** (e.g., one that fetches messages from a database), you must modify the MessagePrinter class.
  - Testing MessagePrinter is hard because you can't easily mock MessageServiceImpl.

Loose Coupling

Definition: Loose coupling occurs when components are minimally dependent on each other, interacting through abstractions (e.g., interfaces or abstract classes) rather than concrete implementations. Changes to one component have little or no impact on others, as long as the abstraction remains unchanged.

Characteristics:

- Components rely on interfaces or abstract classes, not concrete implementations.
- Dependencies are provided externally (e.g., via Dependency Injection), often by a framework like Spring.
- Components can be swapped or modified independently without affecting others.
- Easier to test because dependencies can be mocked or stubbed.

Benefits:

- Increased flexibility: Components can be replaced or extended without modifying dependent code.
- Better maintainability: Changes to one component don't propagate to others.
- Easier testing: Dependencies can be mocked, allowing isolated unit tests.
- Improved reusability: Components can be reused in different contexts.

Example of Loose Coupling (Using Spring DI):

```
// Interface for abstraction
public interface MessageService {
   String getMessage();
}

// Concrete implementation
public class MessageServiceImpl implements MessageService {
   @Override
   public String getMessage() {
      return "Hello, Spring!";
   }
}

public class DbMessageServiceImpl implements MessageService {
   @Override
   public String getMessage() {
       // .... taken data from DB
      return msg;
   }
}
// Dependent class
public class MessagePrinter {
   private final MessageService messageService;

   public MessagePrinter(MessageService messageService) {
      this.messageService = messageService;
   }

   public void printMessage() {
      System.out.println(messageService.getMessage());
   }
                public static void main(String[] args) {
                        MessagePrinter printer = new MessagePrinter();
                        printer.printMessage(); // Output: Hello, World!
                }
}
```

Inversion of Control (IoC)
Definition: IoC is a design principle where the control of object creation, lifecycle management, and dependency resolution is inverted from the application code to an external framework or container. Instead of a class creating and managing its dependencies directly, the framework (e.g., Spring's IoC container) takes responsibility for instantiating objects (beans) and wiring them together.
Key Idea: The application code doesn't control the flow; the framework does. This reduces tight coupling and makes the code more modular and testable.

How It Works in Spring:

- The Spring IoC container (e.g., ApplicationContext) manages beans (objects) defined in a configuration (XML, Java, or annotations).
- The container creates instances, injects dependencies, and manages their lifecycle.
- Developers specify *what* dependencies are needed, not *how* to create them.

Dependency Injection (DI)

Definition: DI is a specific implementation of IoC where dependencies (objects or services that, class needs) are *injected* into a class from an external source (e.g., the IoC container) rather than the class creating them itself. This is typically done via constructors, setters, or fields.

Key Idea: DI decouples a class from its dependencies, allowing them to be swapped or mocked easily (e.g., for testing).

Types of DI in Spring:

1. **Constructor Injection**: Dependencies are passed through a class's constructor.
2. **Setter Injection**: Dependencies are set via setter methods.
3. **Field Injection**: Dependencies are injected directly into fields (using annotations like @Autowired, less common due to testing challenges).

Benefits of DI:

- Loose coupling between classes.
- Easier unit testing (dependencies can be mocked).
- Greater flexibility to swap implementations.

How the IoC Container is Created

The IoC container in Spring is created by initializing one of its implementations, which reads the configuration metadata to set up beans and their dependencies. The process involves the following steps:

1. **Choose an IoC Container Implementation**:
    - Spring provides two main IoC container interfaces:
        - **BeanFactory**: The basic IoC container, providing fundamental dependency injection and bean management. It uses lazy initialization (beans are created only when requested).
        - **ApplicationContext**: A more advanced container that extends BeanFactory. It adds features like event propagation, internationalization, resource loading, and eager initialization (beans are created at startup by default).
    - ApplicationContext is the most commonly used container in Spring applications due to its additional features.
2. **Provide Configuration Metadata**:
    - The container needs metadata to know which beans to create and how to wire their dependencies. This metadata can be provided via:
        - **XML-Based Configuration**: An XML file (e.g., applicationContext.xml) defines beans and dependencies.
        - **Java-Based Configuration**: Java classes annotated with @Configuration and @Bean define beans programmatically.
        - **Annotation-Based Configuration**: Annotations like @Component, @Service, and @Autowired are used with component scanning to auto-detect and wire beans.
3. **Instantiate the Container**:

- The container is created by instantiating an implementation of ApplicationContext (or BeanFactory for simpler use cases). Common ApplicationContext implementations include:
  - **ClassPathXmlApplicationContext**: Loads XML configuration from the classpath.
  - **FileSystemXmlApplicationContext**: Loads XML configuration from the file system.
  - **AnnotationConfigApplicationContext**: Loads Java-based or annotation-based configuration.
  - **WebApplicationContext**: Used in web applications (e.g., with Spring MVC).
- The container reads the configuration metadata, creates beans, and resolves dependencies.

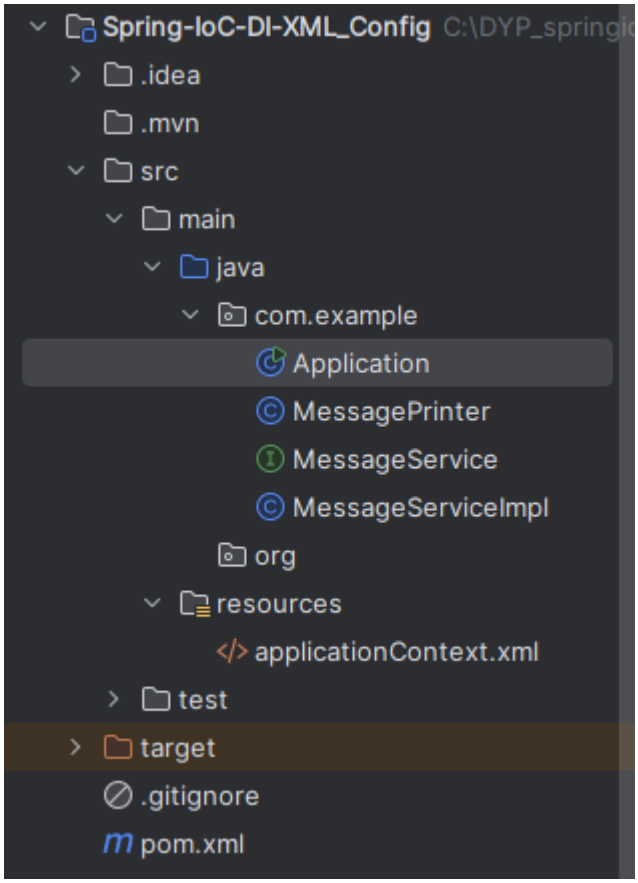
4. **Container Lifecycle**:
   - Once created, the container:
     - Parses the configuration metadata.
     - Creates and configures beans (eagerly for ApplicationContext, lazily for BeanFactory).
     - Injects dependencies using constructor, setter, or field injection.
     - Manages bean lifecycle (initialization, use, destruction).
   - The application can retrieve beans from the container using getBean() or dependency injection.

IoC and DI in Action: Example (Constructor Injection and XML Configuration)
Let's create a simple Spring application to demonstrate IoC and DI using **Java-based configuration** and **constructor injection**. The example involves a MessageService that a MessagePrinter depends on to print a message.

Overview of the Example
Objective:
- Create a Spring application that prints a message using a MessagePrinter class, which depends on a MessageService.
- Demonstrate how the Spring IoC container manages beans and injects dependencies using:
  - **XML-based configuration** (via applicationContext.xml). **(Used)**
  - **Java-based configuration** (via AppConfig.java with @Bean methods). **(Not Used)**
  - **Annotation-based configuration** (via @Component and @Autowired with component scanning). **(Not Used)**
- Show a consistent folder structure that supports all three approaches.

Components:
- **MessageService**: An interface defining a method to get a message.
- **MessageServiceImpl**: A concrete implementation of MessageService.
- **MessagePrinter**: A class that depends on MessageService and prints the message.
- **applicationContext.xml**: An XML file for XML-based configuration.
- **Application**: The main class that initializes the IoC container and runs the application.
- **pom.xml**: The Maven configuration file with Spring dependencies.

Folder Structure:
- Organizes Java classes in src/main/java/com/example.
- Places the XML configuration in src/main/resources.
- Includes a minimal Maven setup for dependencies.



1. pom.xml

- **Purpose**: Defines the project's dependencies and build configuration for Maven.
- **Content**:
  - Includes the spring-context dependency, which provides the core Spring IoC container and DI functionality.
  - Specifies Java 17 as the source and target version.
  - Configures the Maven compiler plugin.
- **Role in the Example**:
  - Ensures the Spring Framework libraries are available to compile and run the application.
  - Allows the project to be built and executed using Maven commands (e.g., mvn clean install).
- **Key Detail**:
  - The spring-context dependency includes the necessary classes for ApplicationContext, bean management, and DI.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
```

```
  <artifactId>Spring-IoC-DI-XML_Config</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <spring.version>6.0.0</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Spring Core -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.10.1</version>
        <configuration>
          <source>17</source>
          <target>17</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

---

6. src/main/resources/applicationContext.xml

- **Purpose**: Provides XML-based configuration for the IoC container.

Role in the Example:

- Defines two beans: messageService (an instance of MessageServiceImpl) and messagePrinter (an instance of MessagePrinter).
- Uses <constructor-arg ref="messageService"/> to inject the messageService bean into MessagePrinter’s constructor.
- The IoC container (via ClassPathXmlApplicationContext) reads this file to create and wire beans.

Key Detail:

- Provides an external, declarative way to configure the IoC container, which is useful for legacy systems or when configuration needs to be modified without changing code.

```
<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">
  <bean id="messageService" class="com.example.MessageServiceImpl"/>
  <bean id="messagePrinter" class="com.example.MessagePrinter">
    <constructor-arg ref="messageService"/>
  </bean>
</beans>
```

---

2. src/main/java/com/example/MessageService.java

- **Purpose**: Defines the MessageService interface, which serves as an abstraction for the service.

 Role in the Example:

- Provides a contract for the service, promoting loose coupling.
- MessagePrinter depends on this interface, not the concrete implementation, allowing different implementations to be swapped without changing MessagePrinter.

 Key Detail:

- The interface ensures that MessagePrinter is loosely coupled to the service implementation, a core principle of DI.

```
package com.example;
public interface MessageService {
  String getMessage();
}

//------------------------------------------------------------
```

3. src/main/java/com/example/MessageServiceImpl.java

Purpose: Implements the MessageService interface to provide a concrete service.

 Role in the Example:

Provides the actual logic for the getMessage() method, returning a hardcoded string.

- The @Component annotation marks it as a Spring bean for **annotation-based configuration**, enabling auto-detection during component scanning.
- For **XML-based** and **Java-based configurations**, the @Component annotation is ignored, and the bean is defined explicitly in applicationContext.xml or AppConfig.java.

Key Detail:

- The same implementation is used across all configurations, ensuring consistent behavior.
- The @Component annotation makes it reusable for annotation-based configuration without needing explicit bean definitions.

```
package com.example;
public class MessageServiceImpl implements MessageService {
  @Override
  public String getMessage() {
    return "Hello from XML!";
  }
}
```

---

4. src/main/java/com/example/MessagePrinter.java

Purpose: A class that depends on MessageService to print a message.

Role in the Example:

- Demonstrates Dependency Injection by receiving a MessageService instance via constructor injection.
- The @Component annotation marks it as a Spring bean for **annotation-based configuration**.
- The @Autowired annotation on the constructor enables automatic dependency injection for **annotation-based configuration**.
- For **XML-based configuration**, the dependency is injected via the <constructor-arg> tag in applicationContext.xml.
- For **Java-based configuration**, the dependency is injected programmatically in AppConfig.java.

Key Detail:

- Uses constructor injection, which is preferred for mandatory dependencies and immutability (the messageService field is final).
- The class is agnostic to how the IoC container is configured, demonstrating loose coupling.

```
package com.example;
public class MessagePrinter {
   private final MessageService messageService;
   // Constructor Injection
   public MessagePrinter(MessageService messageService) {
      this.messageService = messageService;
   }
   public void printMessage() {
      System.out.println(messageService.getMessage());
   }
}
```

7. src/main/java/com/example/Application.java

- **Purpose**: The main class that initializes the IoC container and runs the application.

Role in the Example:

- Demonstrates how to create the IoC container for each configuration type:
  - **XML-Based**: Uses ClassPathXmlApplicationContext to load applicationContext.xml.
  - **Java-Based**: Uses AnnotationConfigApplicationContext to load AppConfig with @Bean methods.
  - **Annotation-Based**: Uses AnnotationConfigApplicationContext to load AppConfig with @ComponentScan.
- Retrieves the MessagePrinter bean from the container and calls printMessage() to demonstrate the wired dependency.

Key Detail:

- The commented sections allow you to test each configuration by uncommenting the relevant block.
- All configurations produce the same output, showing that the IoC container achieves the same result regardless of the configuration approach.

```
package com.example;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class Application {
   public static void main(String[] args) {
      // Create IoC container
      ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
      MessagePrinter printer = context.getBean("messagePrinter",MessagePrinter.class);
      printer.printMessage(); // Output: Hello from XML!
   }
}
```

Implementing the Setter Injection in above application

- **Change MessagePrinter.java**

```
package com.example;

public class MessagePrinter {
    private MessageService messageService;

    // Setter Injection
    public void setMessageService(MessageService messageService) {
        this.messageService = messageService;
    }
    public void printMessage() {
        System.out.println(messageService.getMessage());
    }
}
```

- **Change applicationContext.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">

    <bean id="messageService" class="com.example.MessageServiceImpl"/>
    <bean id="messagePrinter" class="com.example.MessagePrinter">
        <property name="messageService" ref="messageService"/>
    </bean>
</beans>
```

Implementing the Field Injection in above application

- **Change MessagePrinter.java**

```
package com.example;

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

public class MessagePrinter {
    @Autowired
    private MessageService messageService;


    public void printMessage() {
        System.out.println(messageService.getMessage());
    }
}
```

- The @Autowired annotation is central to the field injection mechanism, particularly in enabling the XML-based configuration to support pure field injection without setters or constructors.

- **Change 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
    <bean id="messageService" class="com.example.MessageServiceImpl"/>
    <bean id="messagePrinter" class="com.example.MessagePrinter"/>
</beans>
```

Using Java-based configuration (via AppConfig.java with @Bean methods).

Java-based configuration of Inversion of Control (IoC) in the Spring Framework allows you to configure the Spring IoC container using Java classes instead of XML or annotations. It uses @Configuration and @Bean annotations to define beans and their dependencies explicitly in Java code, providing type safety and programmatic flexibility.

- **@Configuration:** Marks a class as a source of bean definitions for the Spring IoC container.
- **@Bean:** Defines a bean within a @Configuration class. The method name typically becomes the bean name, and the method returns the bean instance.

How It Works

1. Create a @Configuration class to define beans.
2. Use @Bean methods to instantiate and configure beans.
3. The Spring container uses these classes to wire dependencies and manage beans.

Advantages

- Compile-time type checking.
- Easier to refactor than XML.
- Programmatic control over bean creation (e.g., conditional logic).
- Better integration with modern IDEs for autocompletion and navigation.

When to Use

- When you prefer programmatic control over configuration.
- For projects avoiding XML entirely.
- When complex bean creation logic is needed.

In above program delete and add AppConfig,java

```
package example;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
   @Bean
   public MessageService messageService() {
      return new MessageServiceImpl();
   }
```

```
    @Bean
    public MessagePrinter messagePrinter() {
        return new MessagePrinter();
    }
}
```

Change the way of creating IoC Container in Application.java

```
package example;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Application {
    public static void main(String[] args) {
        // Create IoC container
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        MessagePrinter printer = context.getBean("messagePrinter",MessagePrinter.class);
        printer.printMessage();
    }
}
```

What is java-based configuration of Spring IoC and How constructor injection, setter injection and field injection achieved there?

Java-based configuration in Spring IoC (Inversion of Control) uses Java classes with annotations to define and configure beans, as opposed to XML-based configuration. It leverages annotations like @Configuration, @Bean, and others to create a programmatic and type-safe approach to configuring the Spring container.

Java-Based Configuration Overview

- **@Configuration**: Marks a class as a configuration class, equivalent to an XML configuration file. It tells Spring that this class contains bean definitions.
- **@Bean**: Defines a bean within a @Configuration class. Methods annotated with @Bean return objects that are registered as beans in the Spring IoC container.
- **Spring IoC**: The container manages bean creation, lifecycle, and dependency injection.

Example of a basic Java-based configuration:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;

@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Here, AppConfig is a configuration class, and myService is a bean of type MyService.

Dependency Injection Types in Java-Based Configuration

Spring supports three main types of dependency injection: **Constructor Injection**, **Setter Injection**, and **Field Injection**. These can all be achieved in Java-based configuration.

1. Constructor Injection

Constructor injection involves passing dependencies through a constructor. It ensures that the bean is fully initialized with its dependencies when created, promoting immutability.

Implementation in Java-Based Configuration:

- Define a bean with a constructor that accepts dependencies.
- In the @Configuration class, use the @Bean method to instantiate the bean and pass dependencies.

Example:

java
Copy

```
public class MyService {
   private final MyRepository repository;

   public MyService(MyRepository repository) {
      this.repository = repository;
   }
}

public class MyRepository {
   // Repository logic
}

@Configuration
public class AppConfig {

   @Bean
   public MyRepository myRepository() {
      return new MyRepository();
   }

   @Bean
   public MyService myService() {
      return new MyService(myRepository()); // Constructor injection
   }
}
```

- **How it works**: The myService bean is created by calling its constructor with the myRepository bean as an argument.
- **Advantages**: Ensures dependencies are set at creation, immutable, and easier to test.

2. Setter Injection

Setter injection uses setter methods to inject dependencies after the bean is instantiated. It allows for flexibility, as dependencies can be changed after bean creation.

Implementation in Java-Based Configuration:

- Define a bean with setter methods for dependencies.
- In the @Configuration class, create the bean and call the setter to inject dependencies.

Example:
java
Copy
```
public class MyService {
   private MyRepository repository;

   public void setRepository(MyRepository repository) {
      this.repository = repository;
   }
}

public class MyRepository {
   // Repository logic
}

@Configuration
public class AppConfig {

   @Bean
   public MyRepository myRepository() {
      return new MyRepository();
   }

   @Bean
   public MyService myService() {
      MyService service = new MyService();
      service.setRepository(myRepository()); // Setter injection
      return service;
   }
}
```

- **How it works**: The myService bean is created, and its setRepository method is called to inject the myRepository bean.
- **Advantages**: Allows optional dependencies and flexibility to reconfigure beans.

3. Field Injection

Field injection directly injects dependencies into fields, typically using the @Autowired annotation. It is less explicit and generally discouraged due to difficulties in testing and lack of immutability.

Implementation in Java-Based Configuration:

- Use @Autowired on a field in the bean class.
- Ensure the dependency is defined as a bean in the @Configuration class.
- Field injection typically requires component scanning or explicit bean registration with @Autowired.

Example:
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyService {
   @Autowired
```

```
    private MyRepository repository; // Field injection
}

public class MyRepository {
    // Repository logic
}

@Configuration
public class AppConfig {

    @Bean
    public MyRepository myRepository() {
        return new MyRepository();
    }
}
```

 How it works: Spring's component scanning detects @Component on MyService and injects the myRepository bean into the repository field using @Autowired.
 Note: To enable field injection, you need @ComponentScan or manual bean registration. Add @ComponentScan to the configuration class to scan for @Component classes:

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
    @Bean
    public MyRepository myRepository() {
        return new MyRepository();
    }
}
```

- **Disadvantages**: Harder to test, hides dependencies, and can lead to null pointer issues if not properly managed.

Key Points

- **Constructor Injection** is preferred for mandatory dependencies and immutability.
- **Setter Injection** is useful for optional dependencies or when dependencies might change.
- **Field Injection** is convenient but discouraged due to testing and maintenance issues.
- **@Autowired** is typically used for field injection or to resolve dependencies automatically in setter/constructor injection when combined with @Configuration.
- Java-based configuration is more type-safe and easier to refactor than XML-based configuration.
- Use @ComponentScan if you want to combine Java-based configuration with annotation-driven components (e.g., @Component, @Service).

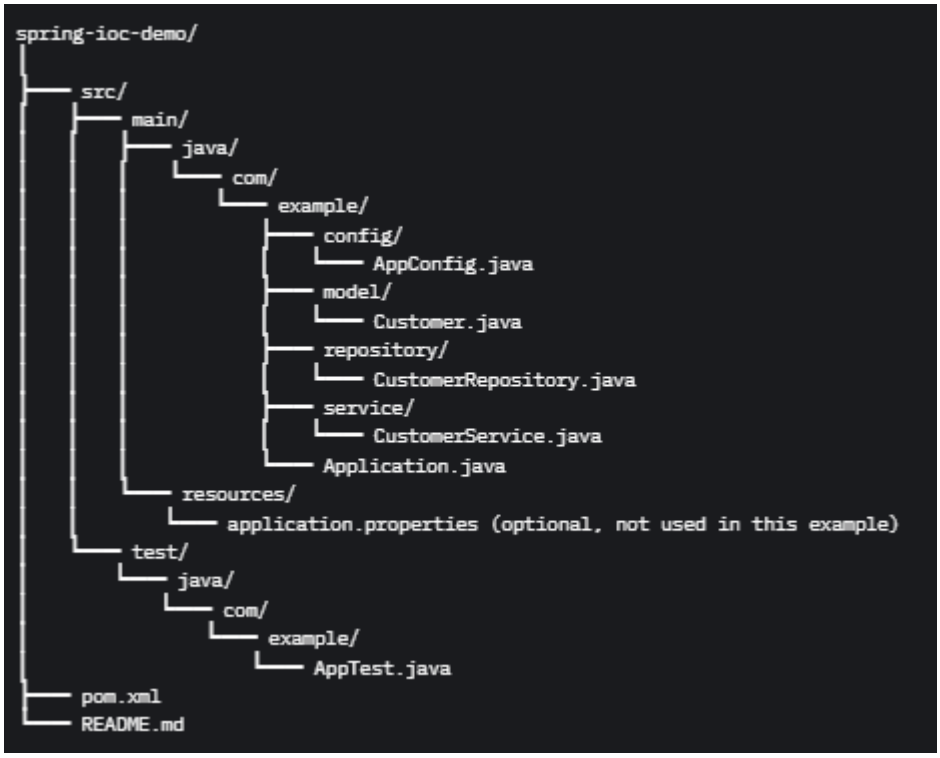
Example Combining All Injection Types

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
```

```
@Component
public class MyService {
   @Autowired
   private MyRepository fieldRepository; // Field injection

   private MyRepository constructorRepository;
   private MyRepository setterRepository;

   public MyService(MyRepository constructorRepository) { // Constructor injection
      this.constructorRepository = constructorRepository;
   }

   public void setSetterRepository(MyRepository setterRepository) { // Setter injection
      this.setterRepository = setterRepository;
   }
}

public class MyRepository {
   // Repository logic
}

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

   @Bean
   public MyRepository myRepository() {
      return new MyRepository();
   }

   @Bean
   public MyService myService() {
      MyService service = new MyService(myRepository()); // Constructor injection
      service.setSetterRepository(myRepository()); // Setter injection
      return service;
   }
}
```

Below is a complete Spring application using Java-based configuration to demonstrate Constructor Injection, Setter Injection, and Field Injection for Spring IoC. The application will be a simple console-based app with a service that interacts with a repository to simulate data retrieval.

The application will use Maven as the build tool and Spring Framework (Spring Core and Spring Context). It will have a CustomerService that depends on a CustomerRepository to retrieve customer data.



Explanation of Structure

- **src/main/java/com/example/**: Contains all Java source code.
  - **config/**: Holds the Spring configuration class (AppConfig.java) for Java-based configuration.
  - **model/**: Contains the Customer class, a simple POJO.
  - **repository/**: Contains the CustomerRepository class, which simulates data access.
  - **service/**: Contains the CustomerService class, which demonstrates all three injection types.
  - **Application.java**: The main class to run the Spring application.
- **src/main/resources/**: Optional folder for configuration files (e.g., application.properties). Not used in this example.
- **src/test/java/**: Contains a simple test class (AppTest.java).
- **pom.xml**: Maven configuration file with dependencies.
- **README.md**: Documentation for the project.

---

1. pom.xml

This file defines the Maven project and includes dependencies for Spring Framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>Spring_IoC-ClassConfig-All__DI</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
  <properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <spring.version>6.0.0</spring.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Spring Core -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.10.1</version>
        <configuration>
          <source>17</source>
          <target>17</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

---

2. src/main/java/com/example/model/Customer.java

A simple POJO representing a customer.

```
package com.example.model;

public class Customer {
  private Long id;
  private String name;

  public Customer(Long id, String name) {
    this.id = id;
    this.name = name;
  }

  public Long getId() {
    return id;
  }
```

```
    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Customer{id=" + id + ", name='" + name + "'}";
    }
}
```

3. src/main/java/com/example/repository/CustomerRepository.java
A repository class that simulates data access.

```
package com.example.repository;

import com.example.model.Customer;

public class CustomerRepository {

    public Customer findCustomerById(Long id) {
        // Simulate database access
        return new Customer(id, "Customer_" + id);
    }
}
```

4. src/main/java/com/example/service/CustomerService.java
A service class demonstrating **Constructor Injection**, **Setter Injection**, and **Field Injection**.

```
package com.example.service;

import com.example.model.Customer;
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomerService {

    // Field Injection
    @Autowired
    private CustomerRepository fieldRepository;

    // Constructor Injection
    private final CustomerRepository constructorRepository;

    // Setter Injection
    private CustomerRepository setterRepository;
```

```
    public CustomerService(CustomerRepository constructorRepository) {
        this.constructorRepository = constructorRepository;
    }

    @Autowired
    public void setSetterRepository(CustomerRepository setterRepository) {
        this.setterRepository = setterRepository;
    }

    public Customer getCustomerByIdUsingFieldInjection(Long id) {
        return fieldRepository.findCustomerById(id);
    }

    public Customer getCustomerByIdUsingConstructorInjection(Long id) {
        return constructorRepository.findCustomerById(id);
    }

    public Customer getCustomerByIdUsingSetterInjection(Long id) {
        return setterRepository.findCustomerById(id);
    }
}
```

5. src/main/java/com/example/config/AppConfig.java

The Java-based configuration class defining beans and enabling component scanning.

```
package example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public MessageService messageService() {
        return new MessageServiceImpl();
    }

    @Bean
    public MessagePrinter messagePrinter() {
        return new MessagePrinter();
    }
}
```

6. src/main/java/com/example/Application.java

The main class to bootstrap the Spring application and test the injections.

```
package com.example;

import com.example.config.AppConfig;
import com.example.service.CustomerService;
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

      // Get CustomerService bean
      CustomerService customerService = context.getBean(CustomerService.class);

      // Test all injection types
     System.out.println("Field Injection: " + customerService.getCustomerByIdUsingFieldInjection(1L));
     System.out.println("Constructor Injection: " + customerService.getCustomerByIdUsingConstructorInjection(2L));
     System.out.println("Setter Injection: " + customerService.getCustomerByIdUsingSetterInjection(3L));

      // Close context
      context.close();
   }
}
```

What is annotation-based configuration of Spring IoC and How constructor injection, setter injection and field injection achieved there?

Annotation-based configuration in Spring IoC (Inversion of Control) uses annotations to define and configure beans, reducing the need for XML or explicit Java-based configuration classes. It relies on annotations like @Component, @Autowired, @Service, @Repository, and others, combined with component scanning, to automatically detect and wire beans in the Spring container. This approach is concise and integrates seamlessly with Java code, making it widely used in modern Spring applications.

Annotation-Based Configuration Overview

Annotation-based configuration allows Spring to automatically discover and configure beans by scanning the classpath for annotated classes. Key annotations include:

- **@Component**: Marks a class as a Spring-managed bean. Variants include @Service (for service-layer classes) and @Repository (for data access classes).
- **@Autowired**: Injects dependencies automatically into fields, constructors, or setter methods.
- **@Configuration**: Optional in pure annotation-based configuration but often used to define additional beans or configure component scanning.
- **@ComponentScan**: Enables component scanning to detect @Component, @Service, @Repository, and other annotated classes in specified packages.
- **Spring IoC**: The container manages bean creation, lifecycle, and dependency injection based on these annotations.

To use annotation-based configuration:

1. Annotate classes with @Component, @Service, or @Repository to mark them as beans.
2. Use @Autowired to inject dependencies.
3. Enable component scanning via @ComponentScan in a @Configuration class or an XML file.

Example of Minimal Annotation-Based Setup:

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
   // No explicit @Bean definitions needed if all beans are annotated
}
```

Here, @ComponentScan tells Spring to scan the com.example package for annotated classes, and Spring will instantiate and wire them automatically.

Dependency Injection Types in Annotation-Based Configuration

Spring supports **Constructor Injection**, **Setter Injection**, and **Field Injection** in annotation-based configuration, primarily using @Autowired or related annotations (e.g., @Inject or @Resource). Below, I'll explain how each is achieved.

1. Constructor Injection

Constructor injection passes dependencies through a constructor, ensuring the bean is fully initialized with its dependencies. In annotation-based configuration, @Autowired is used to mark the constructor for dependency injection.

Implementation:

- Annotate the class with @Component (or @Service, @Repository).
- Define a constructor that accepts dependencies.
- Optionally annotate the constructor with @Autowired (since Spring 4.3, @Autowired is optional for single-constructor classes).

Example:

```
package com.example.service;

import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomerService {
   private final CustomerRepository constructorRepository;

   @Autowired // Optional if there's only one constructor
   public CustomerService(CustomerRepository constructorRepository) {
      this.constructorRepository = constructorRepository;
   }

   public Customer getCustomerById(Long id) {
      return constructorRepository.findCustomerById(id);
   }
}
```

- **How it works**: Spring detects the @Component annotation, instantiates CustomerService, and injects a CustomerRepository bean into the constructor.

- **Advantages**: Promotes immutability, ensures dependencies are set, and is ideal for mandatory dependencies.

2. Setter Injection

Setter injection uses setter methods to inject dependencies after the bean is instantiated. It's useful for optional dependencies or when dependencies may change.

Implementation:

- Annotate the class with @Component.
- Define a setter method and annotate it with @Autowired.

Example:

```
package com.example.service;

import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomerService {
   private CustomerRepository setterRepository;

   @Autowired
   public void setSetterRepository(CustomerRepository setterRepository) {
      this.setterRepository = setterRepository;
   }

   public Customer getCustomerById(Long id) {
      return setterRepository.findCustomerById(id);
   }
}
```

- **How it works**: Spring creates the CustomerService bean, then calls the setSetterRepository method to inject the CustomerRepository bean.
- **Advantages**: Allows flexibility to reconfigure dependencies and supports optional dependencies.

3. Field Injection

Field injection directly injects dependencies into fields using @Autowired. It's the least recommended due to testing difficulties and lack of encapsulation.

Implementation:

- Annotate the class with @Component.
- Annotate the field with @Autowired.

Example:

```
package com.example.service;

import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomerService {
```

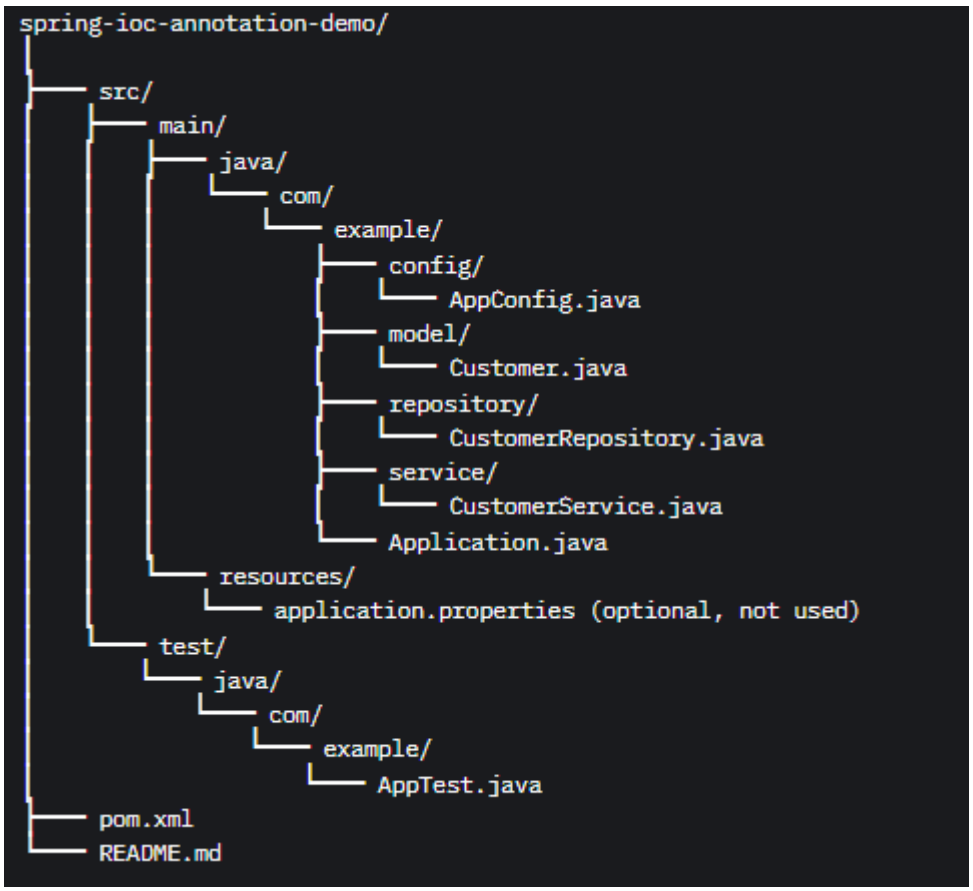
```
    @Autowired
    private CustomerRepository fieldRepository;

    public Customer getCustomerById(Long id) {
        return fieldRepository.findCustomerById(id);
    }
}
```

- **How it works**: Spring detects the @Component annotation, instantiates CustomerService, and uses reflection to inject the CustomerRepository bean into the fieldRepository field.
- **Disadvantages**: Harder to test, hides dependencies, and can lead to null pointer issues if misconfigured.

Complete Application with Annotation-Based Configuration

Since you previously requested a complete application with folder structure for Java-based configuration, I'll adapt the same application to use **annotation-based configuration**. The application will be a console-based app with a CustomerService that depends on a CustomerRepository, demonstrating all three injection types. Below is the folder structure and code.

Folder Structure



This structure is similar to the Java-based configuration example, but the configuration relies on annotations and component scanning instead of explicit @Bean definitions.

Complete Code for Each File

1. pom.xml

Same as the Java-based configuration, with Spring Context dependency.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>org.example</groupId>
   <artifactId>Spring_IoC_DI_AnnotationConfig</artifactId>
   <version>1.0-SNAPSHOT</version>

   <properties>
      <maven.compiler.source>21</maven.compiler.source>
      <maven.compiler.target>21</maven.compiler.target>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <spring.version>6.0.11</spring.version>
</properties>

<dependencies>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>${spring.version}</version>
</dependency>
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.13.2</version>
   <scope>test</scope>
</dependency>
</dependencies>
<build>
<plugins>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.10.1</version>
      <configuration>
         <source>11</source>
         <target>11</target>
      </configuration>
   </plugin>
</plugins>
</build>
</project>
```

2. src/main/java/com/example/model/Customer.java
A simple POJO.

```
package com.example.model;

public class Customer {
    private Long id;
    private String name;

    public Customer(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Customer{id=" + id + ", name='" + name + "'}";
    }
}
```

3. src/main/java/com/example/repository/CustomerRepository.java
A repository marked with @Repository.

```
package com.example.repository;

import com.example.model.Customer;
import org.springframework.stereotype.Repository;

@Repository
public class CustomerRepository {

    public Customer findCustomerById(Long id) {
        // Simulate database access
        return new Customer(id, "Customer_" + id);
    }
}
```

4. src/main/java/com/example/service/CustomerService.java
A service demonstrating all three injection types.

```
package com.example.service;

import com.example.model.Customer;
```

```
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CustomerService {

    // Field Injection
    @Autowired
    private CustomerRepository fieldRepository;

    // Constructor Injection
    private final CustomerRepository constructorRepository;

    // Setter Injection
    private CustomerRepository setterRepository;

    @Autowired
    public CustomerService(CustomerRepository constructorRepository) {
        this.constructorRepository = constructorRepository;
    }

    @Autowired
    public void setSetterRepository(CustomerRepository setterRepository) {
        this.setterRepository = setterRepository;
    }

    public Customer getCustomerByIdUsingFieldInjection(Long id) {
        return fieldRepository.findCustomerById(id);
    }

    public Customer getCustomerByIdUsingConstructorInjection(Long id) {
        return constructorRepository.findCustomerById(id);
    }

    public Customer getCustomerByIdUsingSetterInjection(Long id) {
        return setterRepository.findCustomerById(id);
    }
}
```

5. src/main/java/com/example/config/AppConfig.java
A minimal configuration class to enable component scanning.

```
package com.example.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
```

```
@ComponentScan(basePackages = "com.example")
public class AppConfig {
   // No @Bean definitions needed due to annotation-based configuration
}
```

6. src/main/java/com/example/Application.java

The main class to run the application.

```
package com.example;

import com.example.config.AppConfig;
import com.example.service.CustomerService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

      // Get CustomerService bean
      CustomerService customerService = context.getBean(CustomerService.class);

      // Test all injection types
     System.out.println("Field Injection: " + customerService.getCustomerByIdUsingFieldInjection(1L));
     System.out.println("Constructor Injection: " + customerService.getCustomerByIdUsingConstructorInjection(2L));
     System.out.println("Setter Injection: " + customerService.getCustomerByIdUsingSetterInjection(3L));

      // Close context
      context.close();
   }
}
```