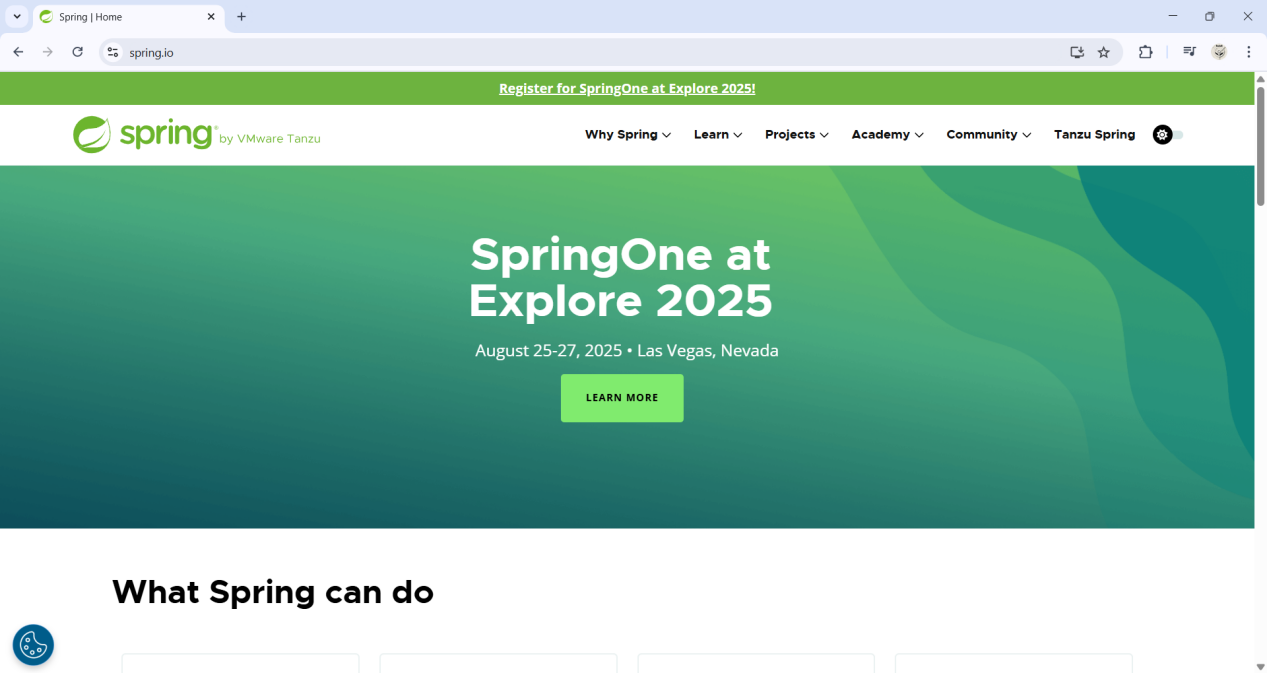
# Spring Framework

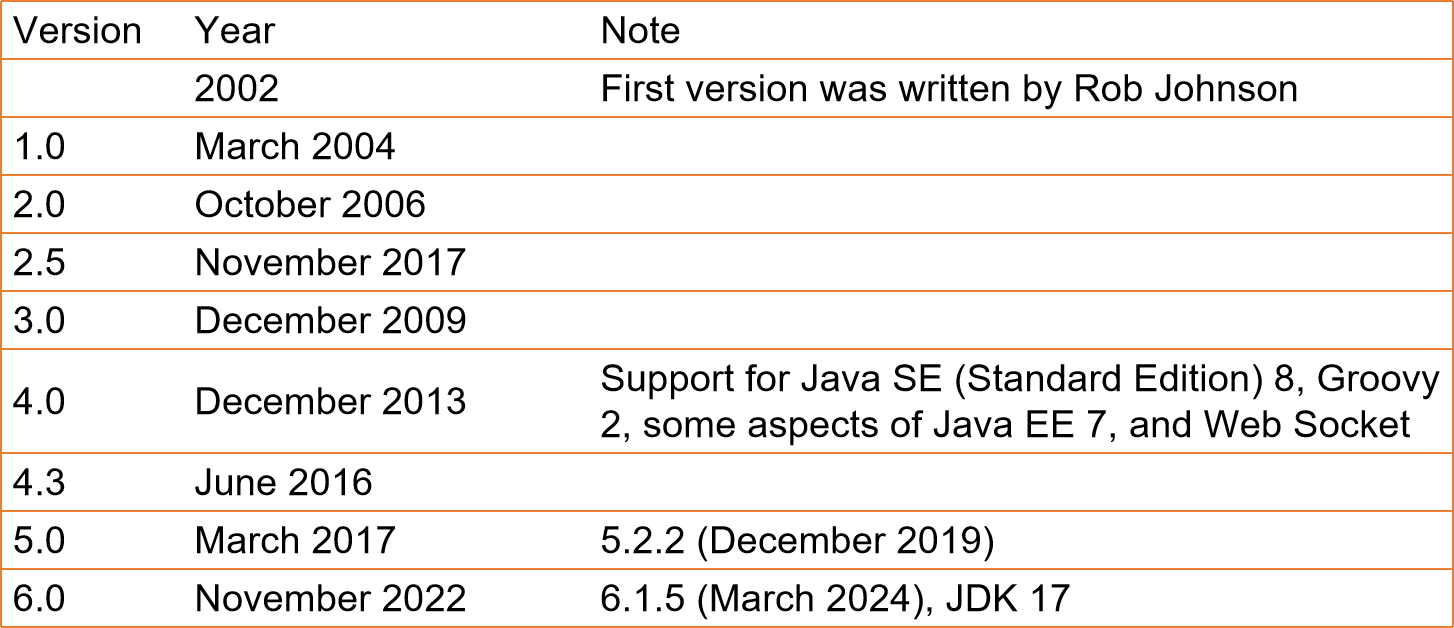
## Tổng quan về Spring Framework

### Spring Framework là gì?

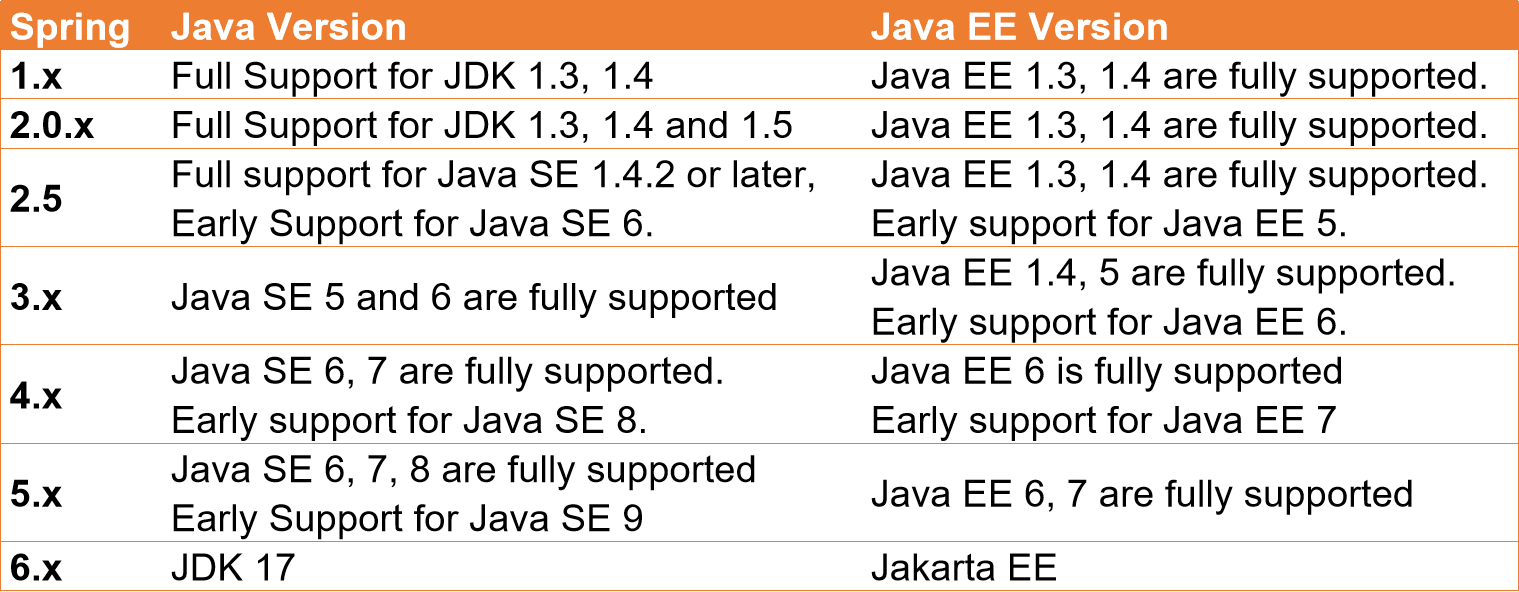
* Spring là framework phát triển ứng dụng phổ biến nhất.
* Spring xử lý hạ tầng bên dưới vì vậy chúng ta chỉ cần tập trung vào logic ứng dụng của ta.
* Hàng triệu lập trình viên khắp thế giới sử dụng Spring Framework để tạo mã nguồn hiệu năng cao, dễ kiểm thử và có thể tái sử dụng.



### Lịch sử phiên bản Spring

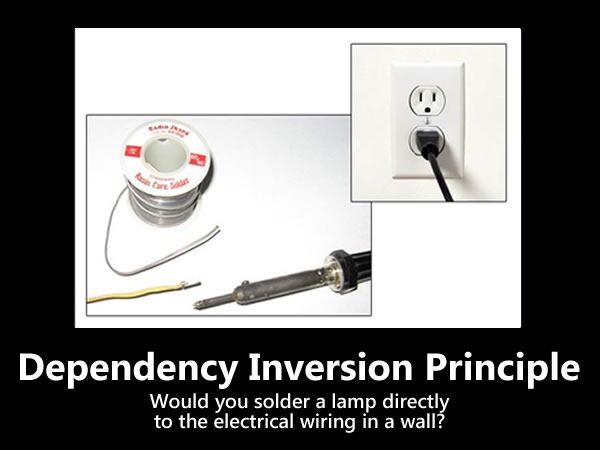


### Khả năng tương thích giữa Spring, Java và Java EE



### Spring - Phát triển nhờ vào thiết kế thông minh

* **Dependency Inversion Principle (DIP)** là khái niện chủ yếu trong thiết kế và kiến trúc phầm mềm, và nó nguyên lý chính đằng sau Spring Framework.
* **Dependency Inversion Principle**:
* Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
* Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. ( Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)



### Những tính năng của Spring

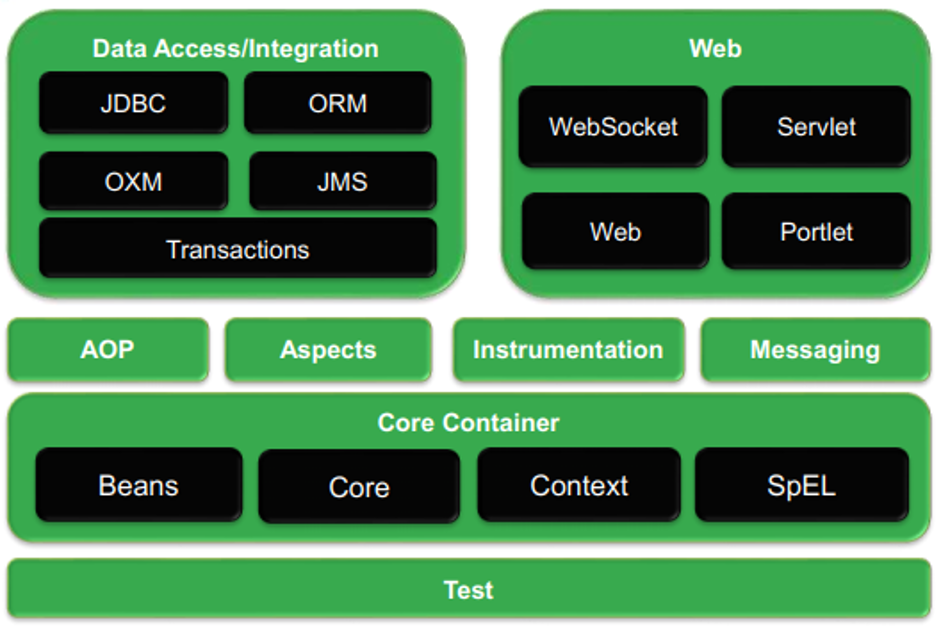
* **Core technologies**: dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
* **Testing**: mock objects, TestContext framework, Spring MVC Test, WebTestClient.
* **Data Access**: transactions, DAO support, JDBC, ORM, Marshalling XML.
* **Spring MVC and Spring WebFlux web frameworks**.
* **Integration**: remoting, JMS, JCA, JMX, email, tasks, scheduling, cache and observability.
* **Languages**: Kotlin, Groovy, dynamic languages.

### Maven

Những lý do nên sử dụng Maven cho những ứng dụng Spring:

* Quản lý các dependency
* Build Automation
* Cấu trúc dự án thích hợp
* Hệ sinh thái plugin
* Cấu hình dự án dễ dàng

### Tổng quan kiến trúc của Spring Framework



### Core Container

Core Container chứa **Beans**, **Core**, **Context** và **SpEL**:

* **Core** module cung cấp các phần chủ yếu của Spring, bao gồm IoC và Dependency Injection.
* **Beans** module cung cấp BeanFactory - là một sophisticated implementation của mẫu factory.
* **Context** module cung cấp truy cập đến những đối tượng được cấu hình như một context. ApplicationContext interface là điểm trọng tâm của Context module.
* **SpEL** module cung cấp Powerful Expression Language cho querying và manipulating an object graph tại runtime

### Data Access/Integration

Tầng Data Access/Integration gồm các module sau:

* **JDBC** module cung cấp một JDBC-abstraction layer mà cắt giảm đi nhu cầu viết những đoạn code dài dòng hoặc chán ngắt liên quan đến JDBC

* **ORM** module cung cấp các tầng tích hợp cho các ORM APIs phổ biến, bao gồm JPA, JDO, Hibernate, và iBatis.
* **OXM** module cung cấp một tầng trừu tượng mà hỗ trợ những Object/XML mapping implementations cho JAXB, Castor, XMLBeans, JiBX và XStream.
* Java Messaging Service **JMS** module chứa những tính năng cho producing và consuming messages.
* **Transactions** module hỗ trợ programmatic and declarative transaction management cho những class mà implement những interface đặt biệt và cho tất cả POJOs.

### Web

Tầng Web bao gồm các module sau:

* **Web** module cung cấp basic web-oriented integration features như multipart file-upload functionality và khởi tạo IoC container sử dụng servlet listeners và một web-oriented application context.
* **Web-MVC** module chứa Spring's Model-View-Controller (MVC) implementation cho những ứng dụng web.
* **Web-Socket** module cung cấp hỗ trợ cho WebSocket-based, two-way communication giữa client và server trong những ứng dụng web.
* **Web-Portlet** module cung cấp MVC implementation được sử dụng trong portlet environment và phản ánh chức năng của Web-Servlet module.

## Dependency Injection & Inversion of Control

### Tight Coupling (Liên kết chặt chẽ)

Giả sử ta có 2 class như sau:

**class B {**

**properties**

**...**

**methods**

**...**

**}**

**class A {**

**B obj = new B(…);**

**...**

**}**

* Lúc này class B được coi là một dependency của class A hay class A phụ thuộc vào class B.
* Đối tượng của class A quản lý vòng đời của đối tượng class B (từ việc khởi tạo, sử dụng và hủy đối tượng của class B).
* Nếu ta muốn thay thế class B bằng một class khác trong class A thì ta phải sửa lại code khá nhiều(những chỗ có B phải thay bằng class ấy) và có thể làm “hỏng” class A.

=> Vì vậy ta cần giảm sự phụ thuộc của 2 class này => **Loose Coupling**

### Loose Coupling (Liên kết lỏng lẻo)

#### Inversion of Control

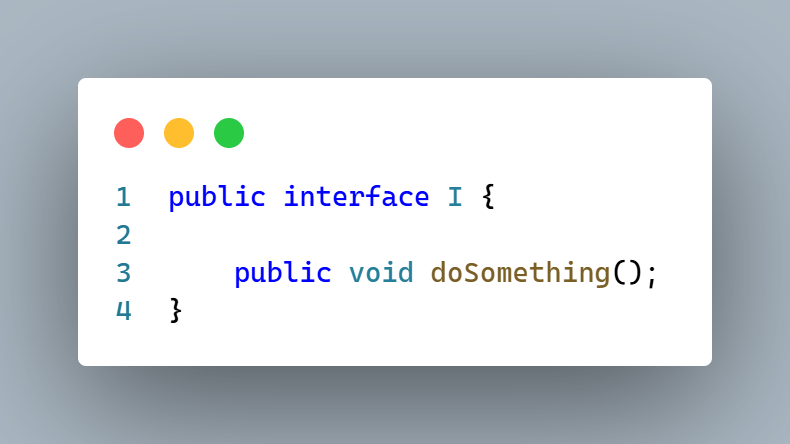
* Đảo ngược điều khiển (IoC) là **nguyên tắc thiết kế** trong đó quyền kiểm soát việc tạo và quản lý đối tượng được tách khỏi chính các đối tượng đó và được đảo ngược (kiểm soát từ bên ngoài).
* IoC là ý tưởng, là lý thuyết, là lời kêu gọi, chứ nó không nói cụ thể làm thế nào, nó **trừu tượng.**

#### Dependency Injection (“Tiêm” phụ thuộc)

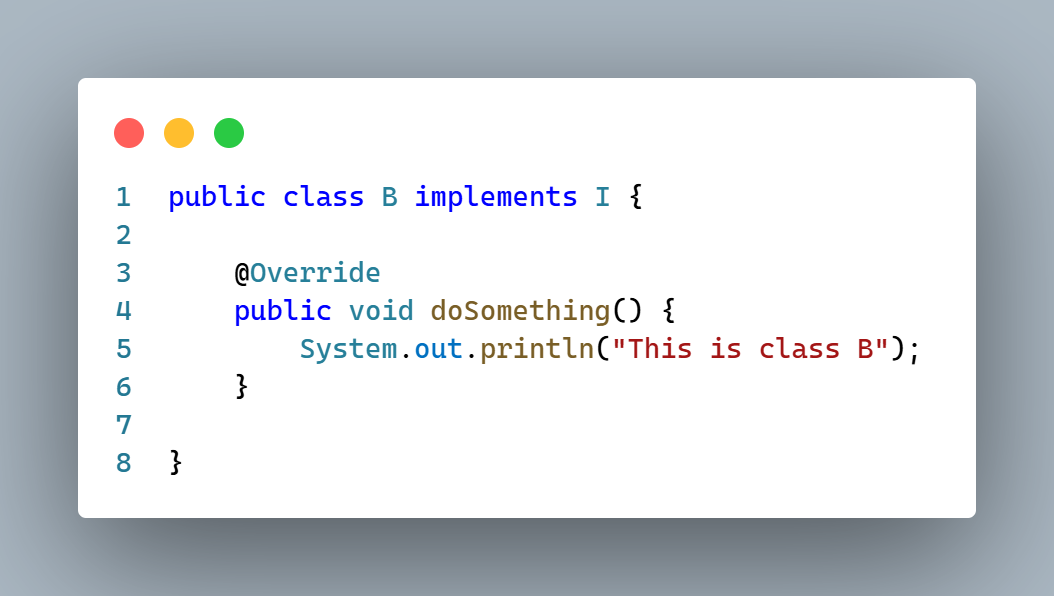
* Dependency Injection (DI) là một mẫu thiết kế, là một phiên bản cụ thể, **hiện thực hóa** IoC.
* Ngoài DI, còn 1 vài cách khác để đạt được IoC.
* Có nhiều cách để thực hiện Dependency Injection:
* Cách 1: Tiêm trực tiếp vào field
* Cách 2: Thông qua constructor
* Cách 3: Thông qua setter

VD:

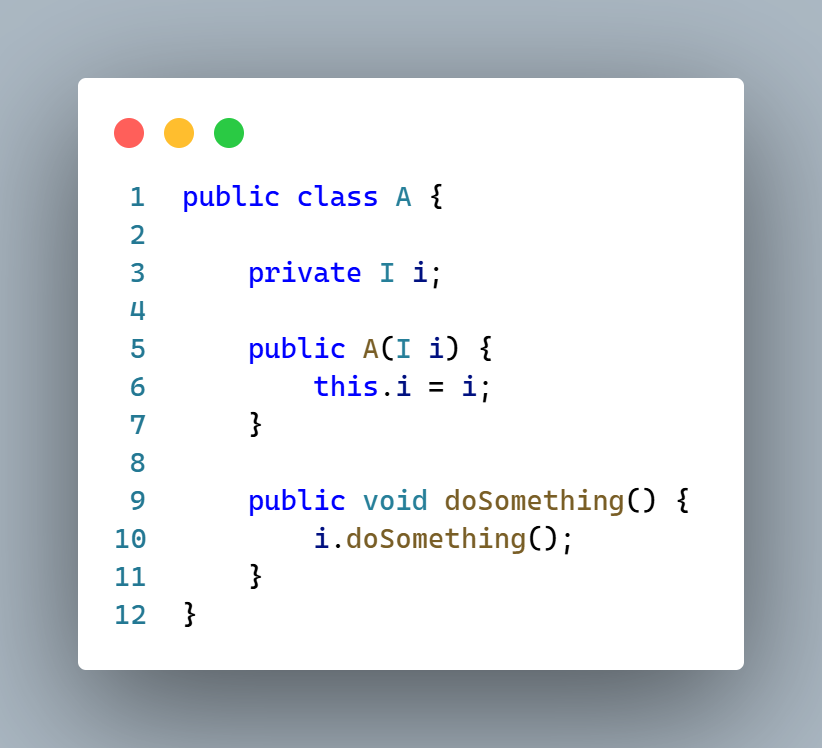
Tạo interface **I** với phương thức trừu tượng **doSomething()**:



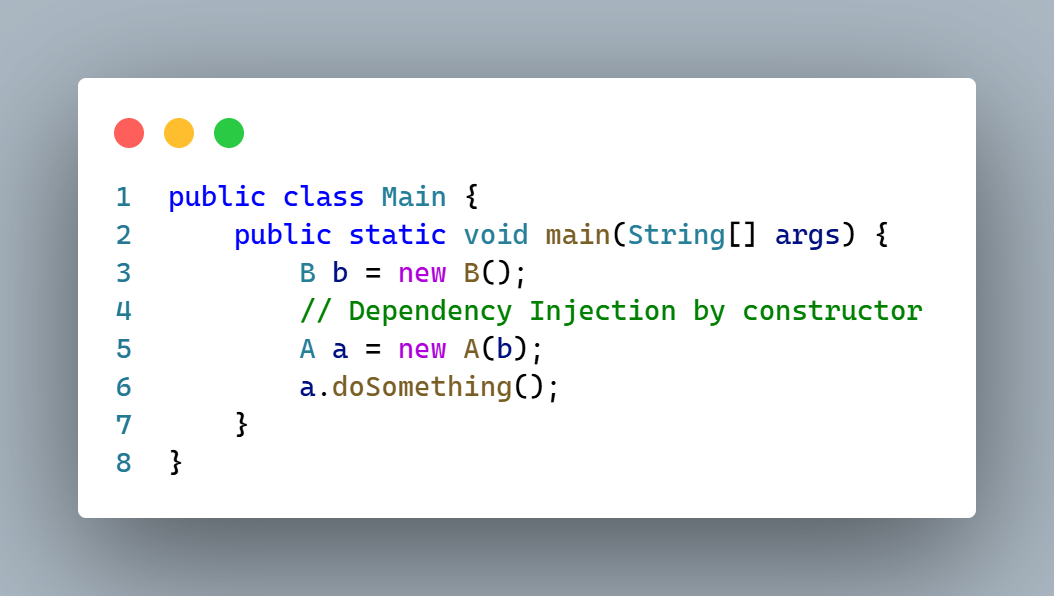
Tạo class **B** implements interface **I** và cài đặt hàm doSomething():



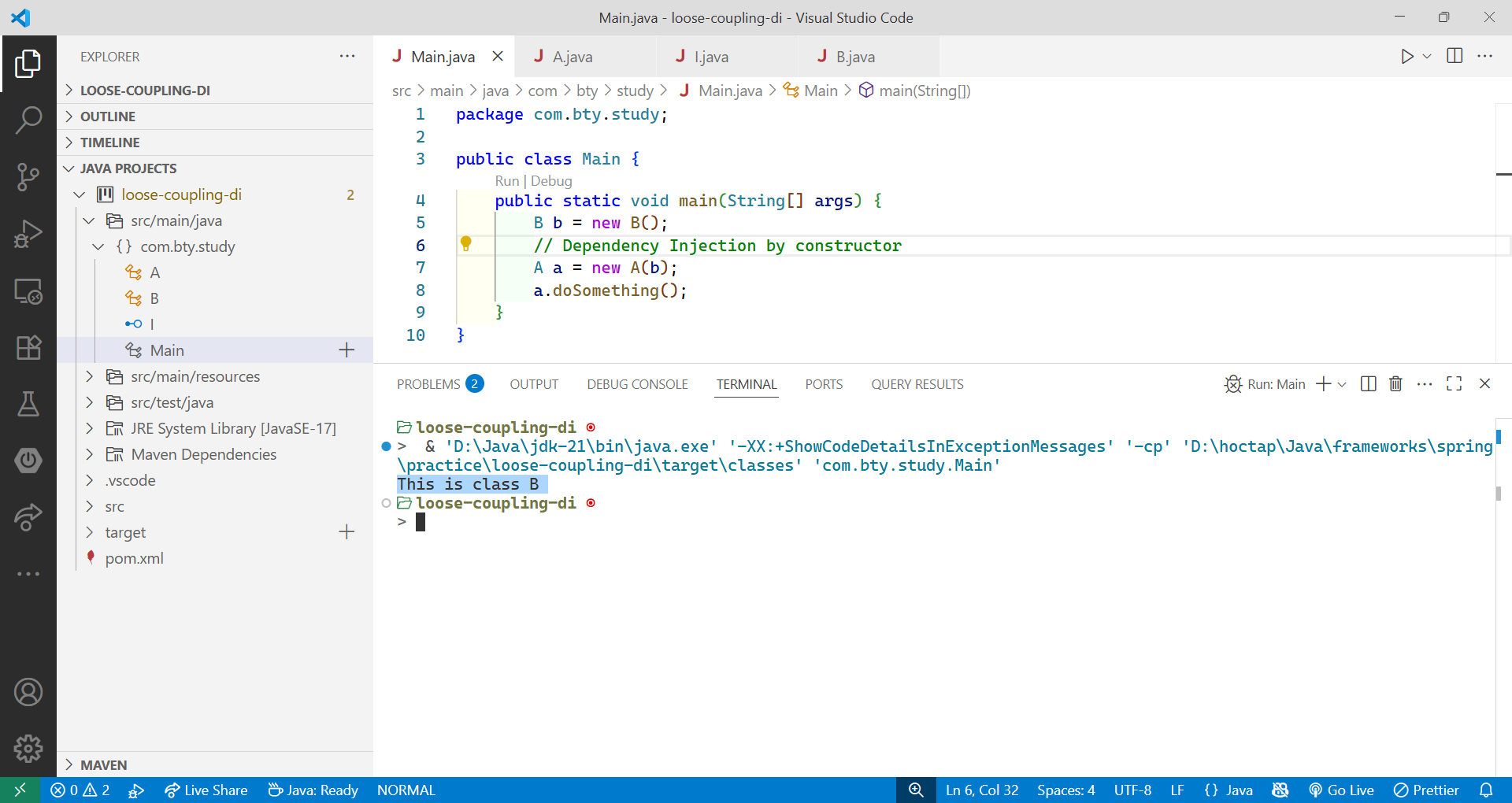
Tạo class **A** có thuộc tính thuộc interface **I** và thực hiện gọi hàm doSomething() của interface I trong phương thức của class A.



Thực hiện truyền phụ thuộc (đối tượng của class B) vào đối tượng class A thông qua **constructor**:

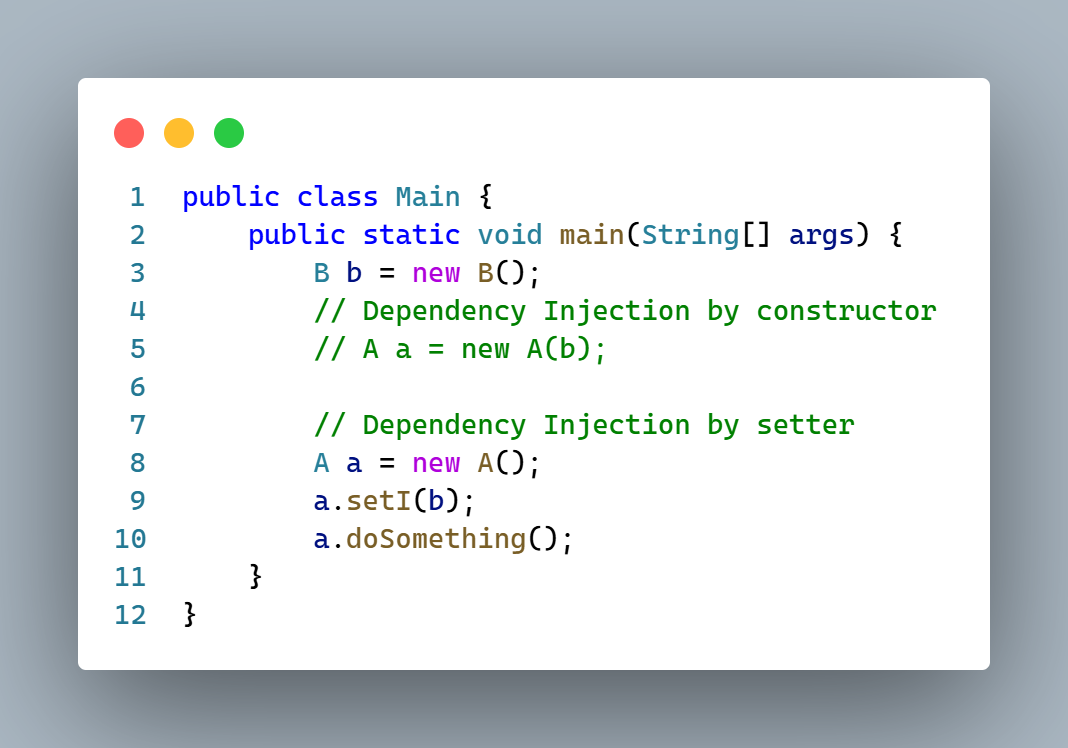


Kết quả:

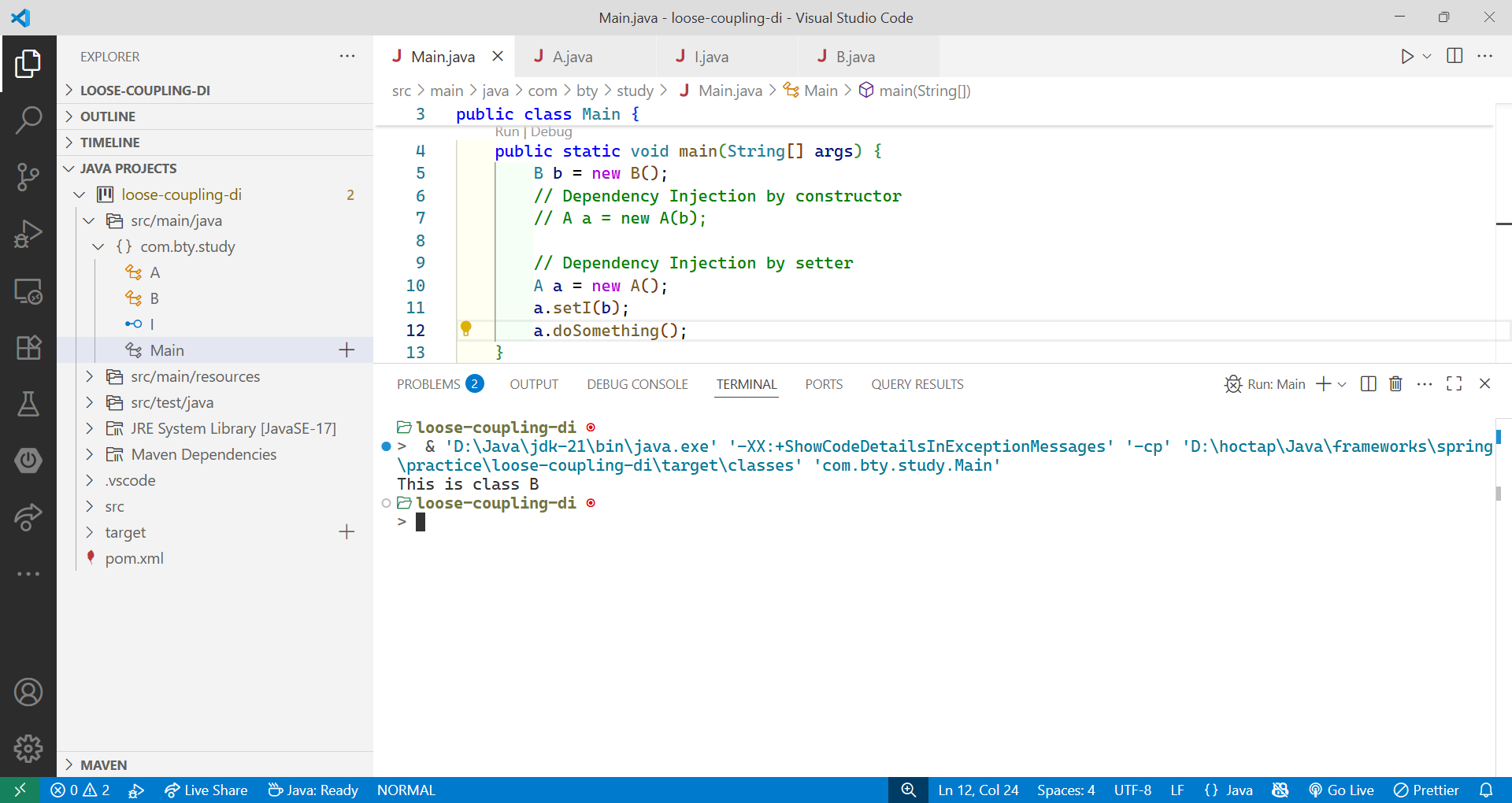


Ta có thể thấy đối tượng của class B đã được truyền vào đối tượng class A và đối tượng class B này đã gọi hàm doSomething() để hiển thị ra dòng chữ “This is class B”.

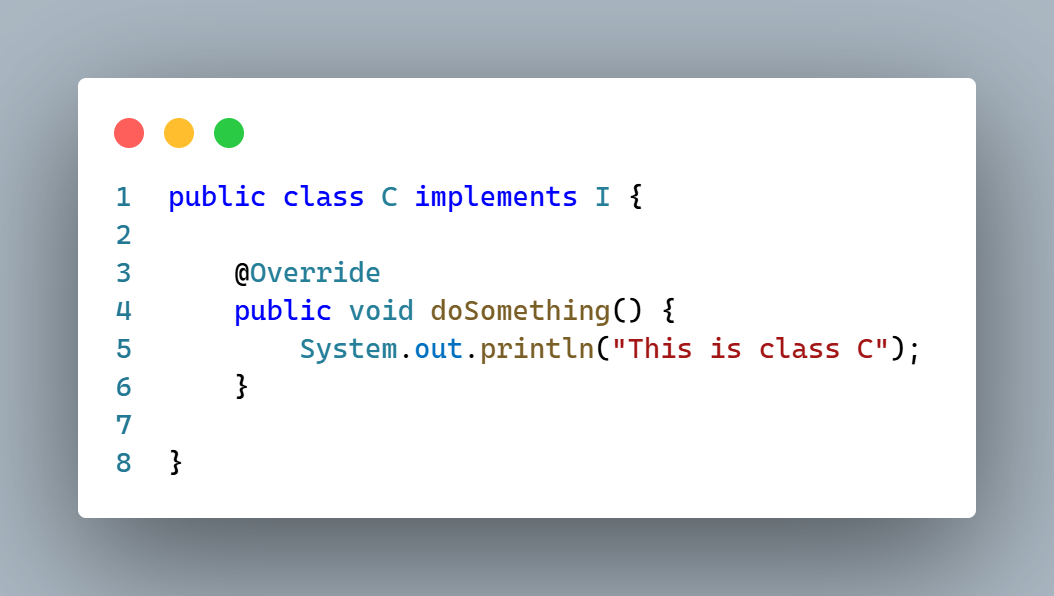
Tương tự đối với việc truyền phụ thuộc thông qua **setter**:

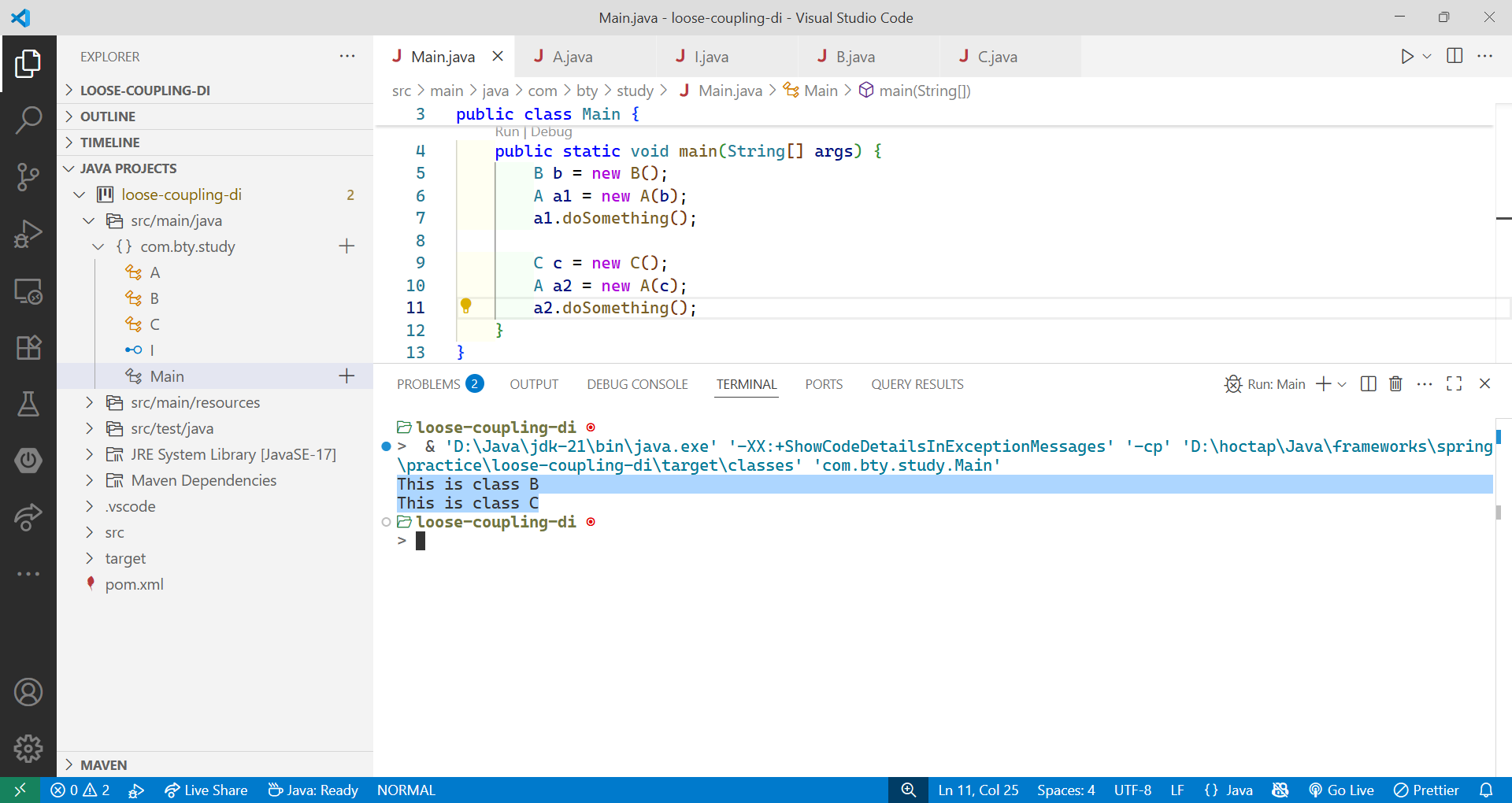


Kết quả:



Lúc này nếu class A nếu muốn “chơi” với class khác với class B, thì ta chỉ cần tạo một class mới, sau đó implements interface I rồi “tiêm” vào qua constructor hoặc setter là được mà không cần phải thay đổi code vì class A chỉ phụ thuộc vào interface I mà thôi:





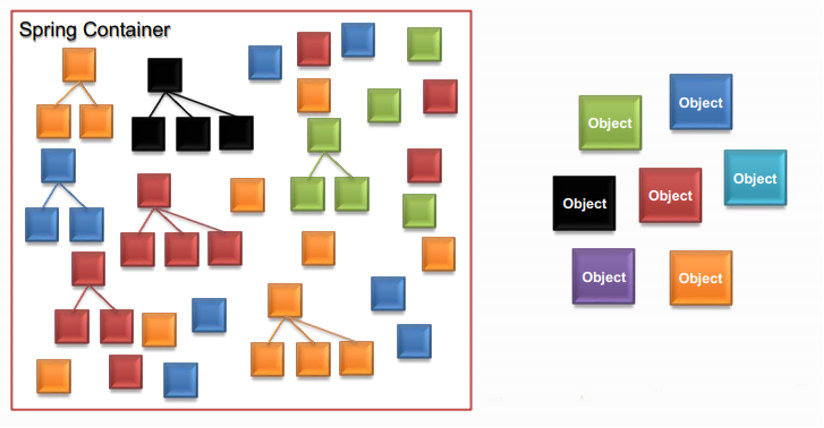
**=> Loose Coupling (Liên kết lỏng lẻo)**

**Vậy ta đã đạt được Loose Coupling thông qua Dependency Injection (qua constructor hoặc settor).**

### Spring Container - Spring IoC Container

Trong Spring, **Spring Container** (**IoC Container**) sẽ tạo các đối tượng, lắp ráp chúng lại với nhau, cấu hình các đối tượng và quản lý vòng đời của chúng từ lúc tạo ra cho đến lúc bị hủy.

Spring Container sử dụng DI để quản lý các **thành phần, đối tượng** để tạo nên 1 ứng dụng. Các thành phần đối tượng này gọi là Spring **Bean**.



Có 2 loại Spring Container:

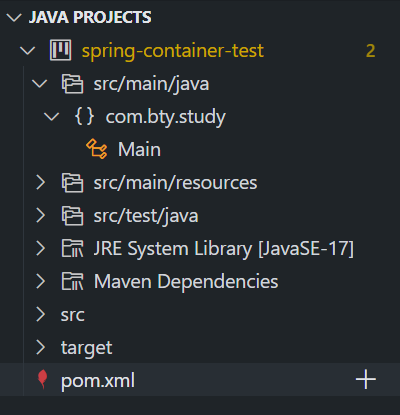
* **BeanFactory Container**: Container đơn giản nhất, cung cấp các chức năng cơ bản nhất của IoC.
* **ApplicationContext Container**: Là bản mở rộng của BeanFactory, cung cấp nhiều tính năng mạnh mẽ hơn như tích hợp AOP, xử lý message resources, publish event, web-aware capabilities, v.v. Trong hầu hết các ứng dụng hiện đại, ApplicationContext là lựa chọn phổ biến.

Bảng so sánh 2 loại Container: BeanFactory vs ApplicationContext

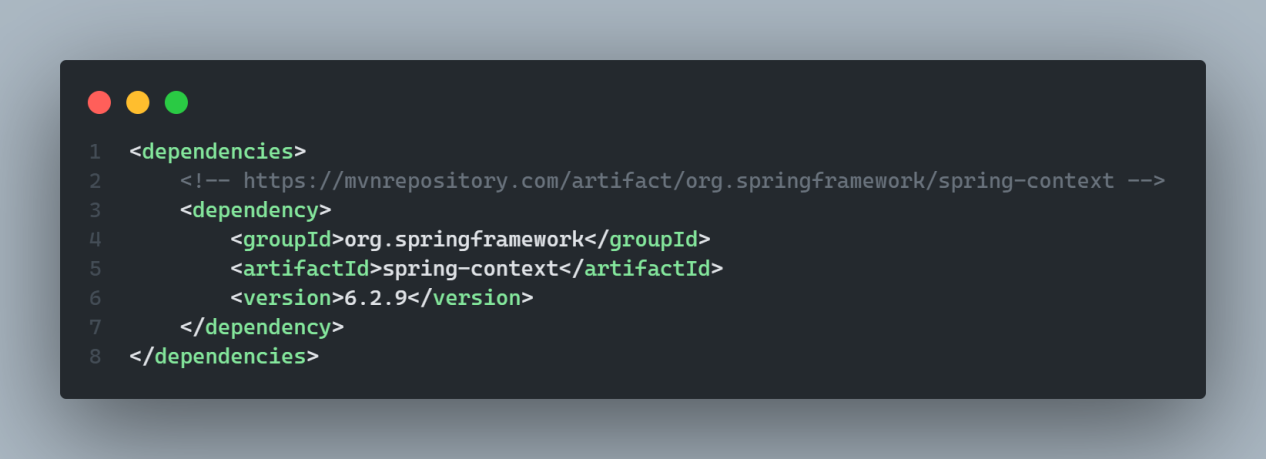
|  |  |  |
| --- | --- | --- |
|  | **BeanFactory** | **ApplicationContext** |
| **Definition** | Container cơ bản cung cấp chức năng cơ bản cho quản lý các beans. | Container cao cấp, được mở rộng từ BeanFactory với nhiều tính năng được thêm vào. |
| **Usage** | Phù hợp cho xây dựng các standalone applications. | Phù hợp cho xây dựng ứng dụng web, tích hợp với AOP modules, ORM, và distributed applications. |
| **Bean Scopes Supported** | Chỉ hỗ trợ Singleton và Prototype bean scopes. | Hỗ trợ tất cả các loại bean scope, bao gồm Singleton, Prototype, Request, Session,… |
| **Annotation Support** | Không hỗ trợ annotation; yêu cầu cấu hình trong XML file. | Hỗ trợ cấu hình dựa trên annotation cho bean autowiring. |
| **Initialization** | Tạo những bean objects khi yêu cầu sử dụng lazy initialization. | Load tất cả beans và tạo những objects khi khởi động sử dụng eager initialization. |

VD: ApplicationContext Container

Tạo **maven project**



Thêm phụ thuộc **Spring Context** vào file **pom.xml**



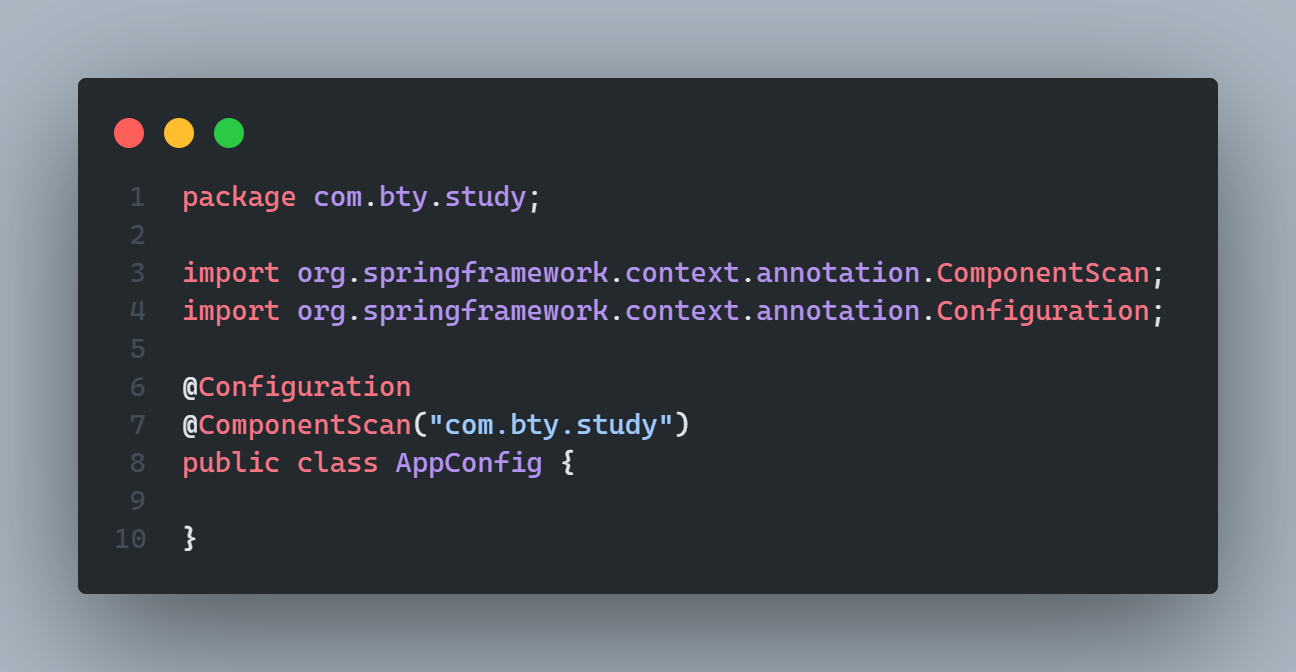
Tạo class **ImageService** trong package **com.bty.study** với annotation **@Component**:



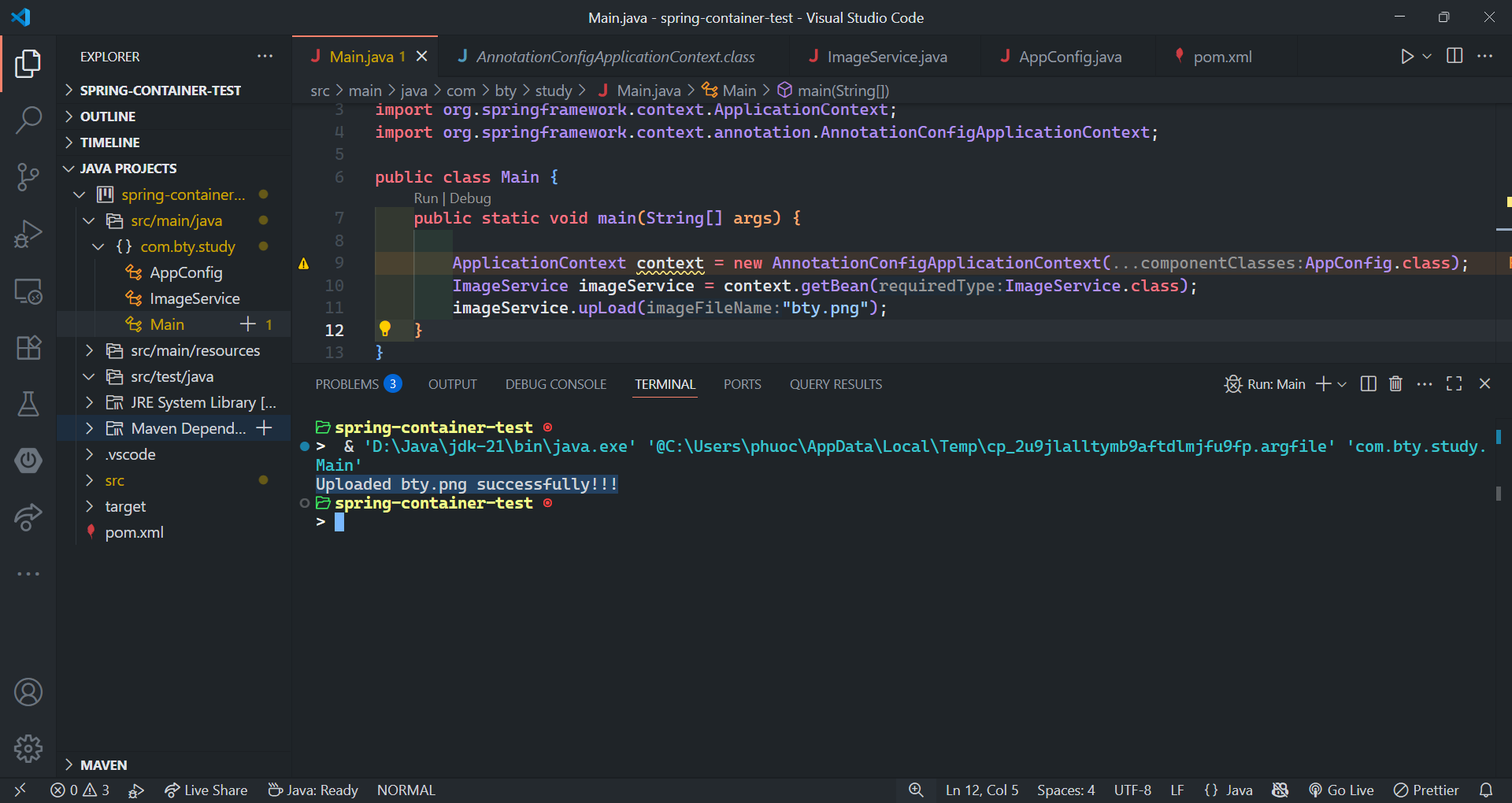
Tạo class **AppConfig** trong package **com.bty.study** với 2 annotation:

**@Configuration**

**@ComponentScan(“com.bty.study”)**



Khởi tạo **ApplicationContext Container** và chạy chương trình:



Có thể thấy **ApplicationContext Container** đã tạo đối tượng của class **ImageService** vì ta đã khởi tạo và truyền vào **AppConfig** class cho Container.

Vì class AppConfig có 2 annotation là:

**@Configuration**

**@ComponentScan(“com.bty.study”)**

Với annotation **@ComponentScan(“com.bty.study”)**, Container sẽ quét các class nằm trong package **com.bty.study** và có chứa:

**@Component**

**@Service**

**@Repository**

**@Controller**

…

Để khởi tạo **bean**

Mà **ImageService** class nằm trong **com.bty.study** và được đánh dấu là **@Componet** nên Container **sẽ khởi tạo đối tượng của class này và quản lý vòng đời của nó**. Vì vậy ta có thể lấy được bean của class ImageService thông qua **context.getBean(tên class);** và sử dụng bean.

### Bean

* Những **Objects** tạo nên **xương sống** cho ứng dụng của chúng ta và được quản lý bởi **Spring IoC Container** thì được gọi là **Bean**.
* Bất kỳ **đối tượng** nào được **khởi tạo**, **lắp ráp** và **quản lý** bởi **Spring IoC Container** đều được gọi là **Bean**

#### Bean Scopes

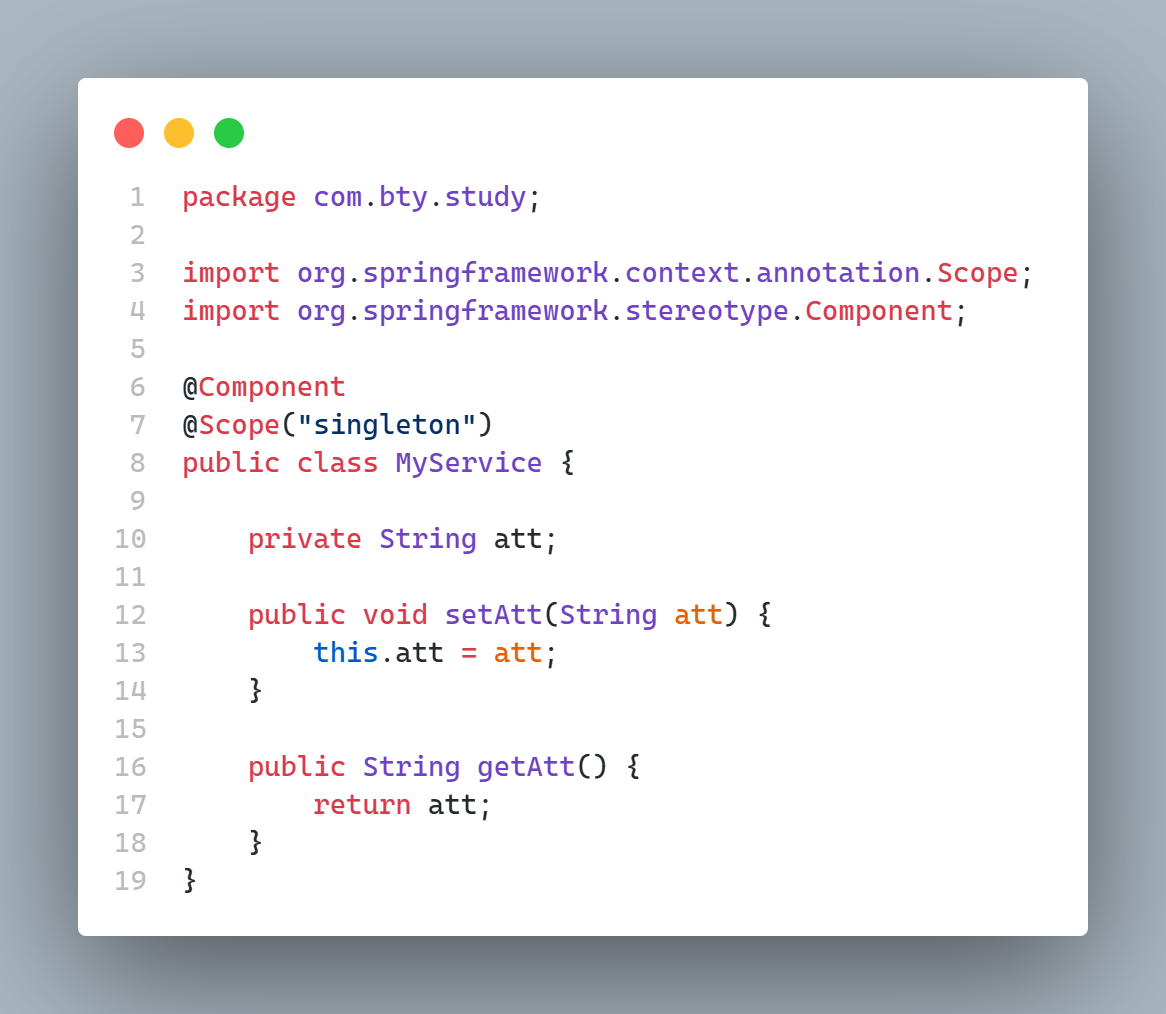
##### Singleton Scope (mặc định)

Một Bean được định nghĩa với **singleton** scope sẽ khiến **IoC container** khởi tạo **duy nhất một** instance cho bean đó và nó được sử dụng trong tất cả các yêu cầu đến bean này.

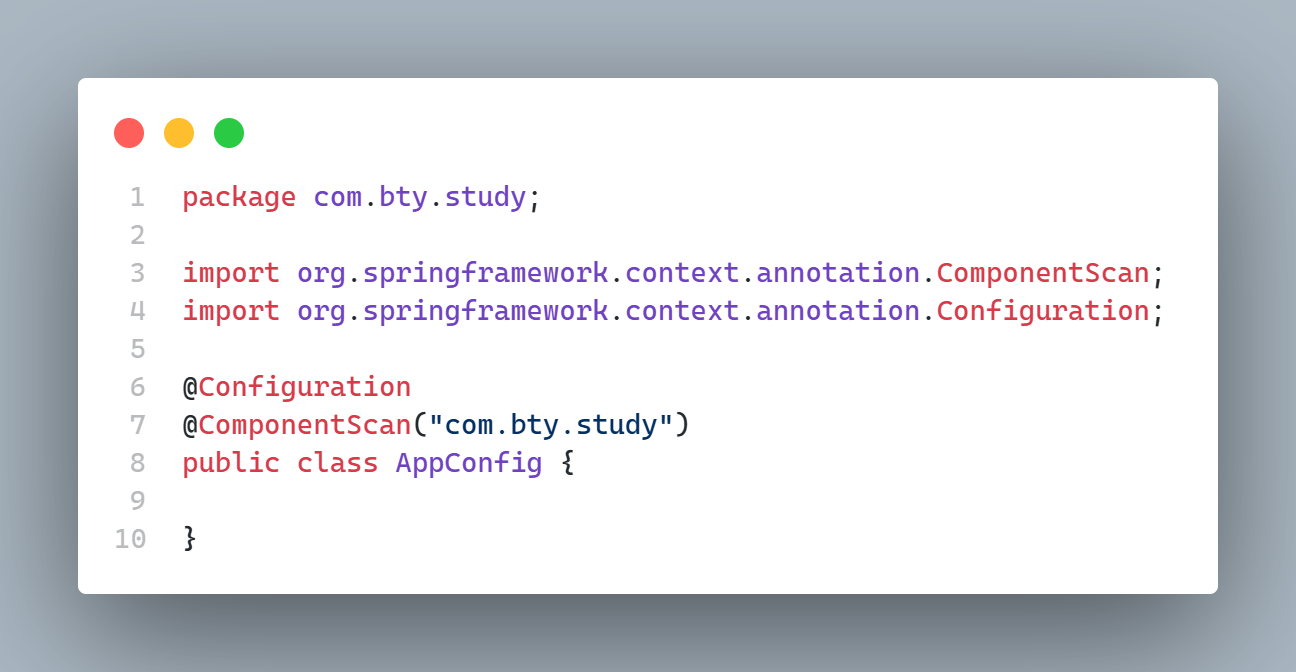
Chỉ duy nhất một thể hiện của bean sẽ được tạo cho **mỗi container**

VD:

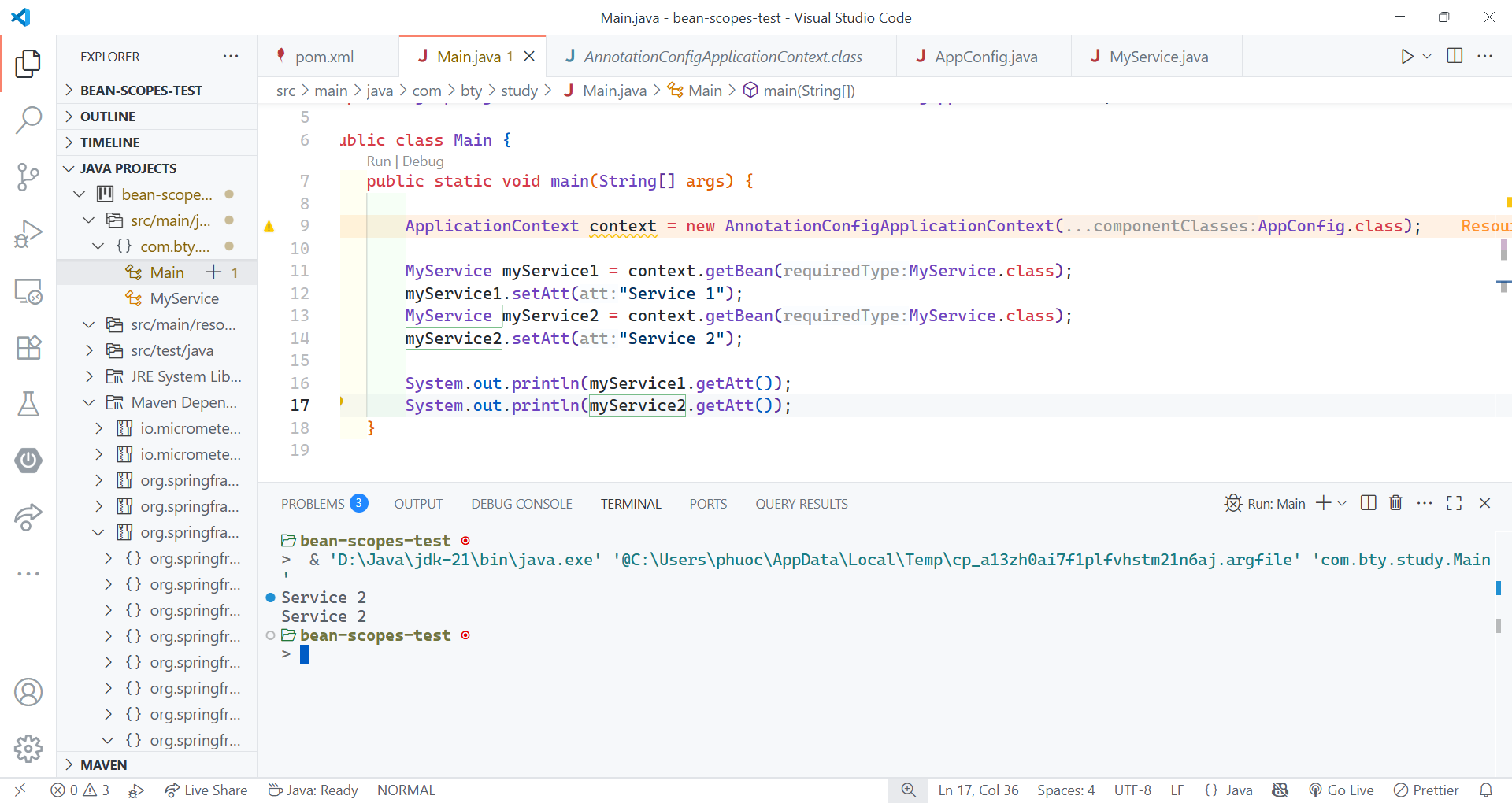
Tạo class **MyService** được đánh dấu là @Component tức là một **bean** và có bean scope là **singleton**:



Tạo class **AppConfig** với **@Configuration** và **@ComponentScan(“com.bty.study”)** để Container có thể quét và tạo ra **bean** của class **MyService**:



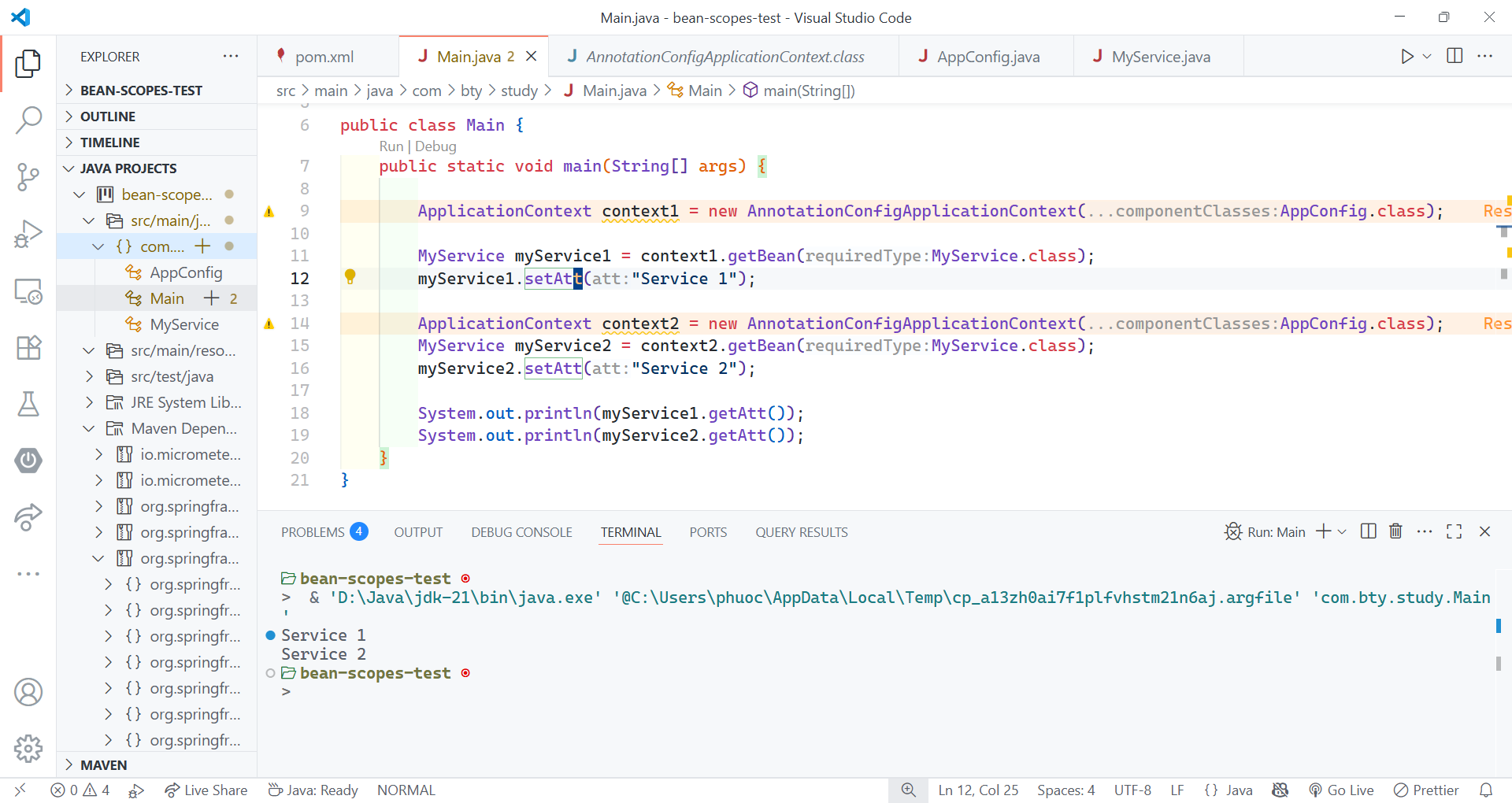
Tạo hàm main() và chạy chương trình:



Ta có thể thấy **myService1.getAtt()** và **myService2.getAtt()** có kết quả **giống nhau** vì cả myService1 và myService2 cùng trỏ đến **1 đối tượng** (**bean của class MyService**) trong context Container.

Nói cách khác vì class **MyService** được đánh dấu là **@Component** và **@Scope("singleton")** nên container chỉ khởi tạo **duy nhất một đối tượng**.

Nếu ta có 2 container thì kết quả sẽ như sau:



Vì class **MyService** được đánh dấu là **@Component** và **@Scope("singleton")** nên chỉ có duy nhất **một đối tượng** cho **mỗi Container**.

Nên **myService1** và **myService2** tham chiếu đến **2 bean khác nhau** của class MyService. Nên kết quả in ra **khác nhau**.

##### Prototype Scope

Một bean với prototype scope sẽ trả về **các instance khác nhau** mỗi khi có một yêu mới sử dụng chúng đến IoC container.

##### Ngoài ra còn có các scope khác như: Request, Session, Global-Session,…

#### Cách định nghĩa bean

Có 3 cách định nghĩa class là một bean:

* Khai báo trong file XML
* Dùng annotation trên class (**@Component**, **@Service**, **@Repository**, **@Controller**)

VD:

@Service // Đánh dấu lớp này là một Spring Bean

public class MyService {

private final MyRepository myRepository;

@Autowired // Yêu cầu Spring tiêm phụ thuộc MyRepository

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

// Các phương thức khác...

}

* Dùng **@Configuration** và **@Bean**

VD:

@Configuration

public class AppConfig {

@Bean

public Car carBean() {

return new Car();

}

@Bean

public PasswordEncoder passwordEncoderBean() {

return new BCryptPasswordEncoder();

}

// Có thể định nghĩa nhiều bean khác với @Bean

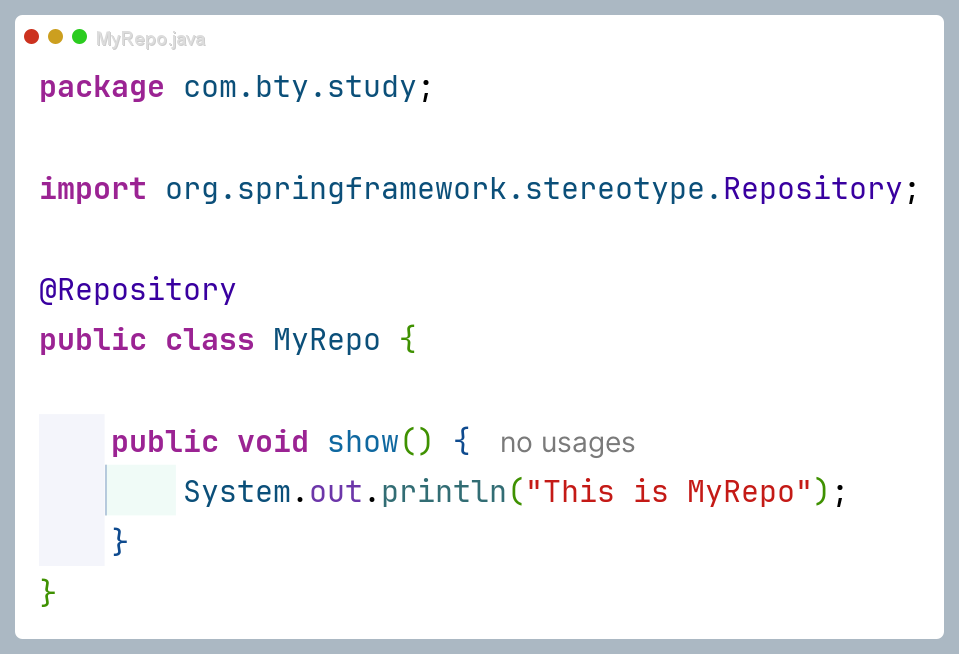
}

## @Autowired annotation trong Spring

**@Autowired** được sử dụng để **tiêm (inject) phụ thuộc (dependency hay bean)** vào trong một bean khác.

### Field Injection

Tạo class **MyRepo** được đánh dấu là **@Repository**



Tạo class **MyService** được đánh dấu là **@Service**

Sử dụng **@Autowired** trên thuộc tính **myRepo** để Spring Container tự động tiên **bean phù hợp** vào đó



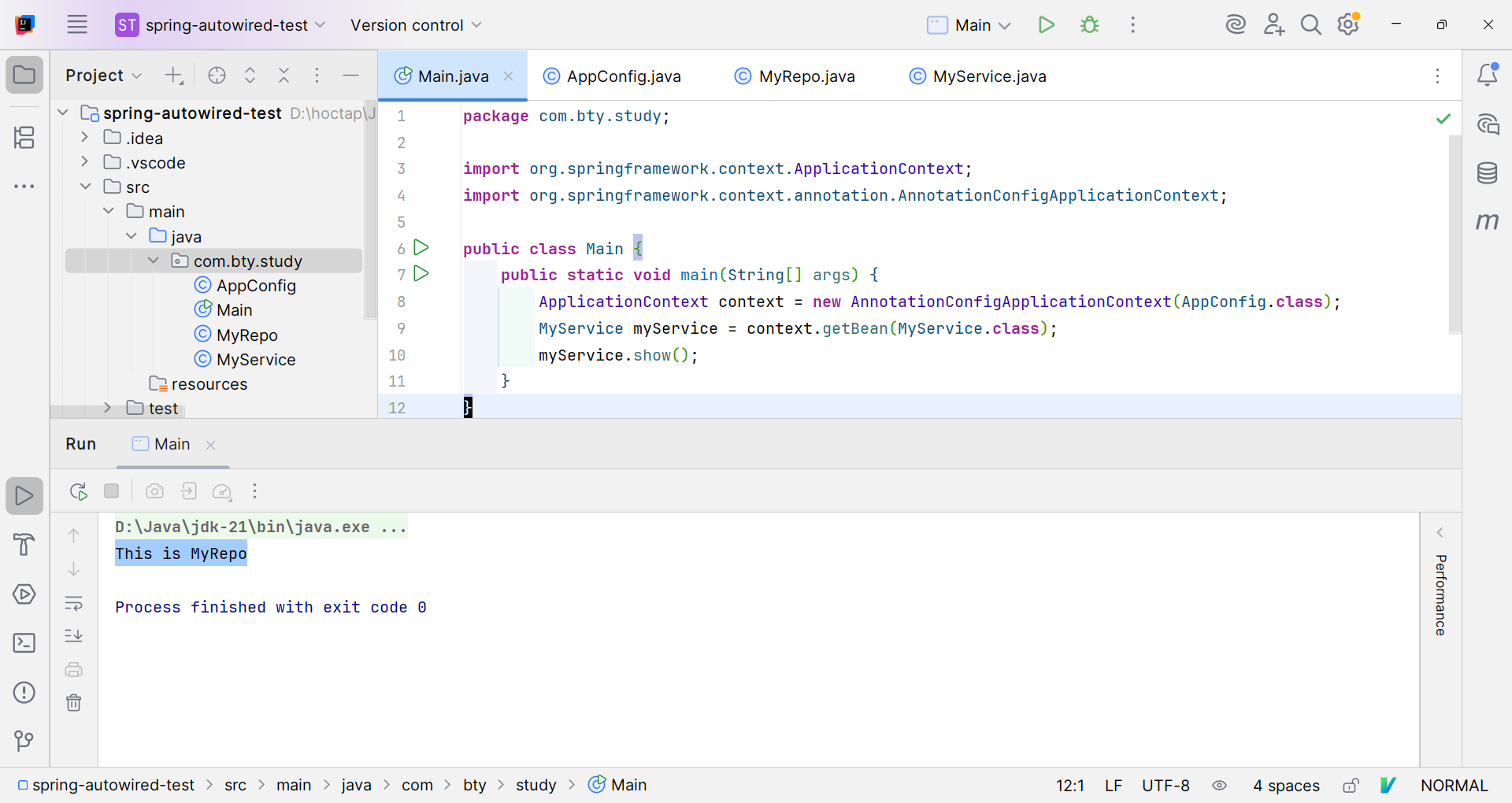
Tạo class **AppConfig** với **@Configuration** và **@ComponentScan(“com.bty.study”)** để Container có thể tìm thấy và tạo bean của class MyService và MyRepo.



Tạo hàm main() và chạy chương trình



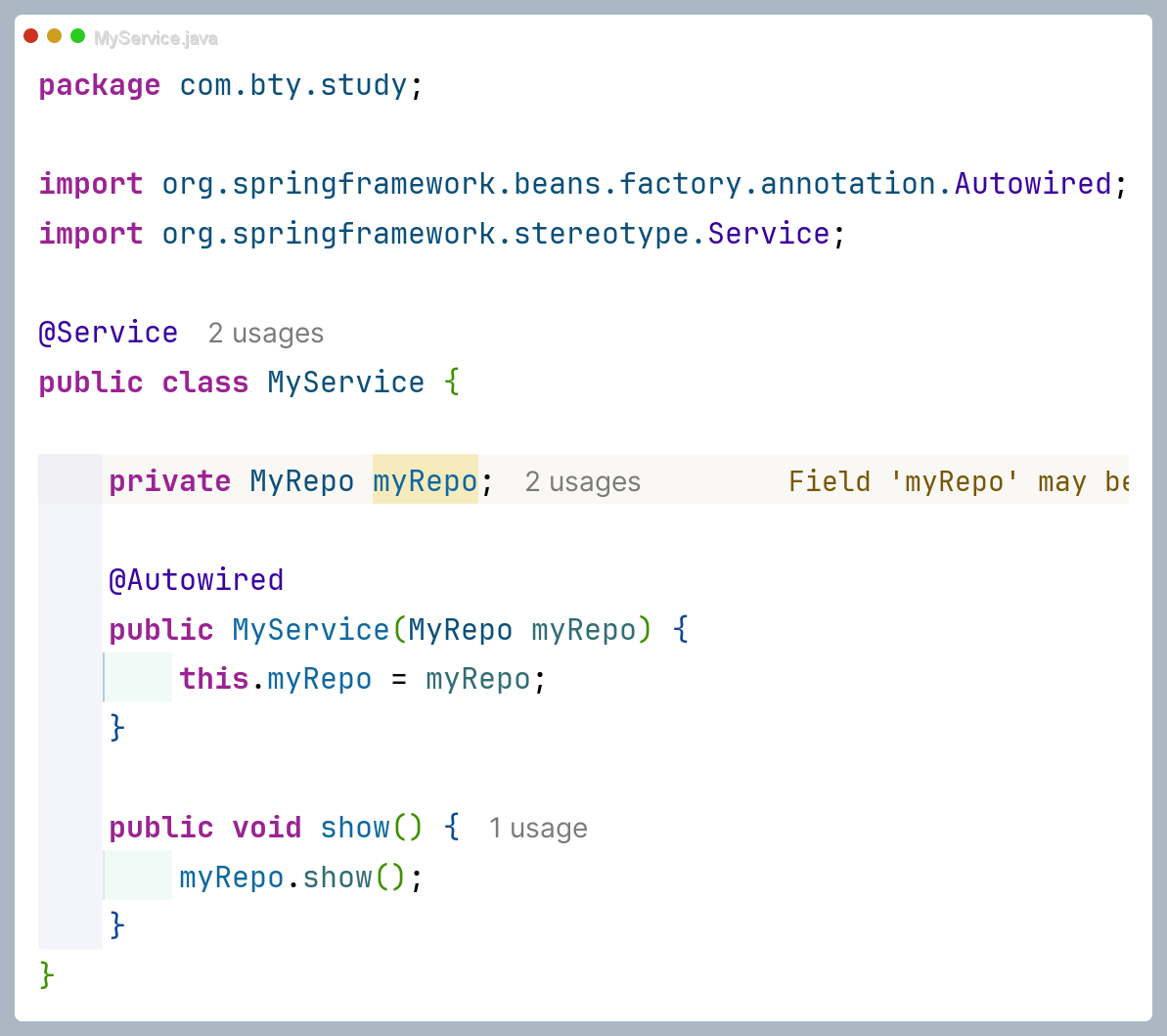
Kết quả:

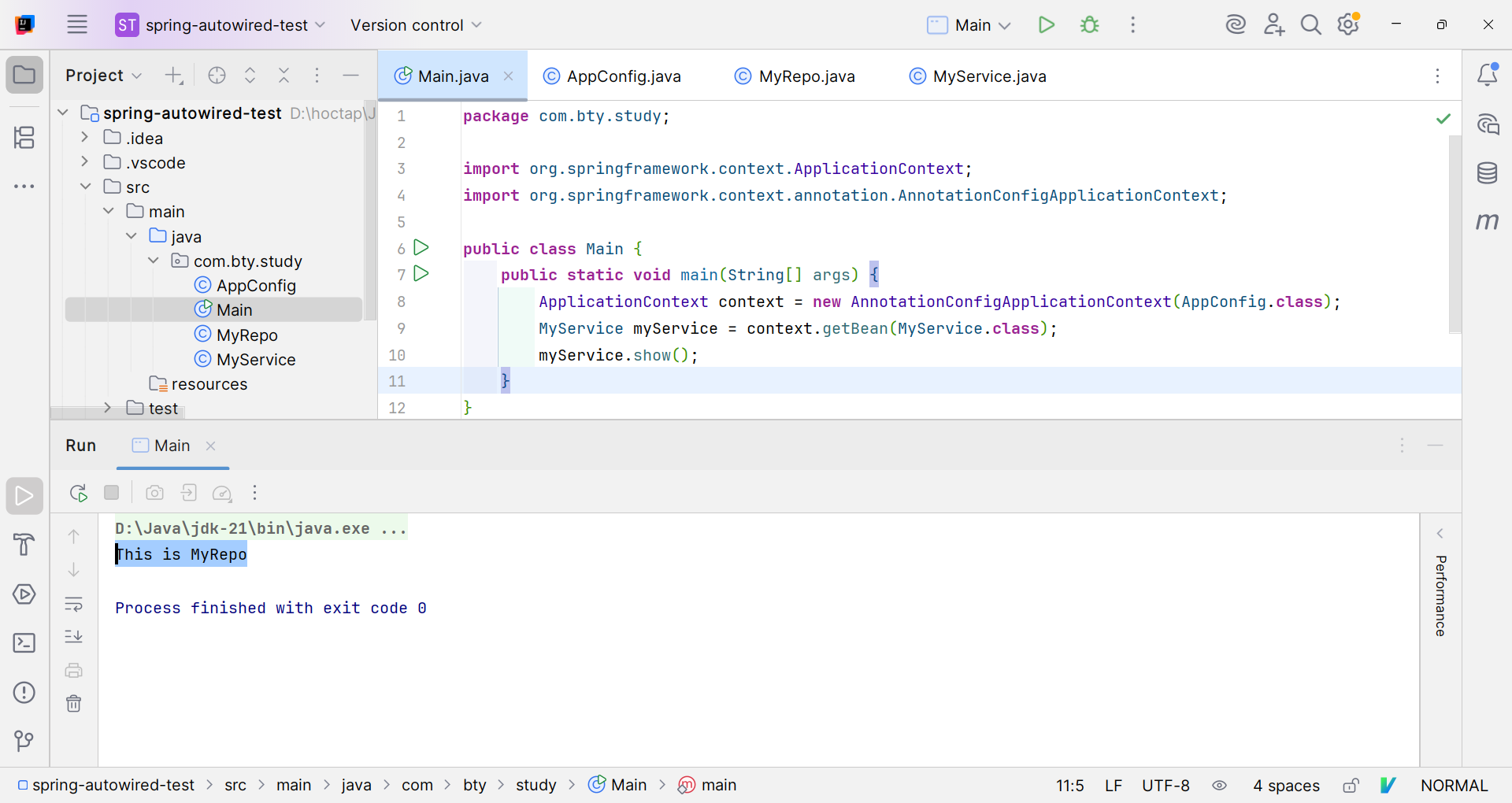


Ta có thể thấy sau khi Container tạo bean của class MyRepo và MyService thì đã tiêm trực tiếp bean của class MyRepo vào thuộc tính myRepo của bean class MyServic. Nên myService.show() có thể gọi được myRepo.show() và in ra “This is MyRepo”.

### Constructor Injection

Ta lấy lại vị dụ ở trên (Field Injection) nhưng lần này ta sẽ tiêm phụ thuộc thông qua **constructor**:



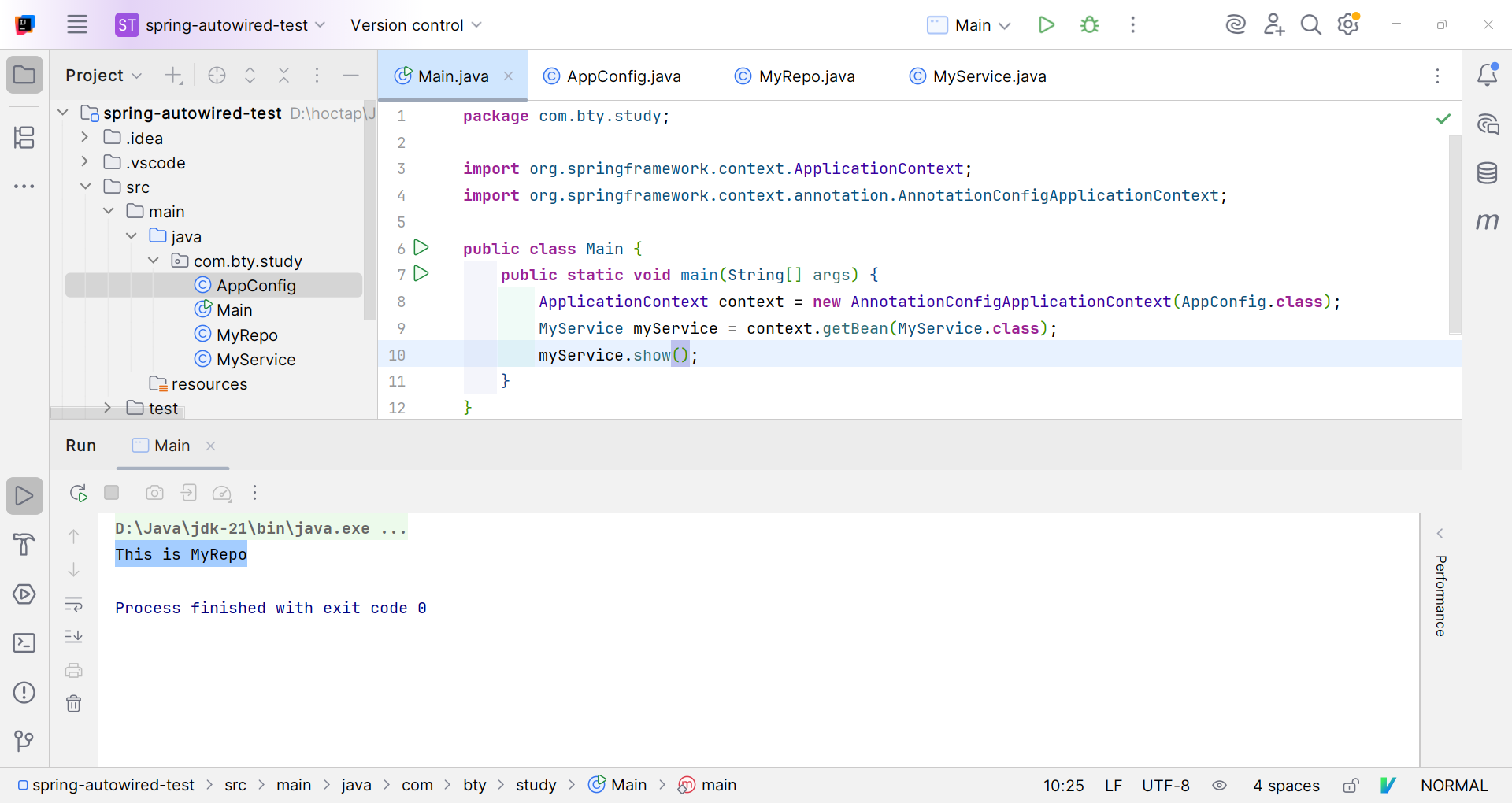


Có thể thấy ta vẫn nhận được kết quả tương tự.

### Setter Injection

Vẫn ví dụ ơ trên nhưng lần này ta tiêm phụ thuộc thông qua **setter**:





Ta có thể thấy kết quả vẫn tương tự như field injection và constructor injection.