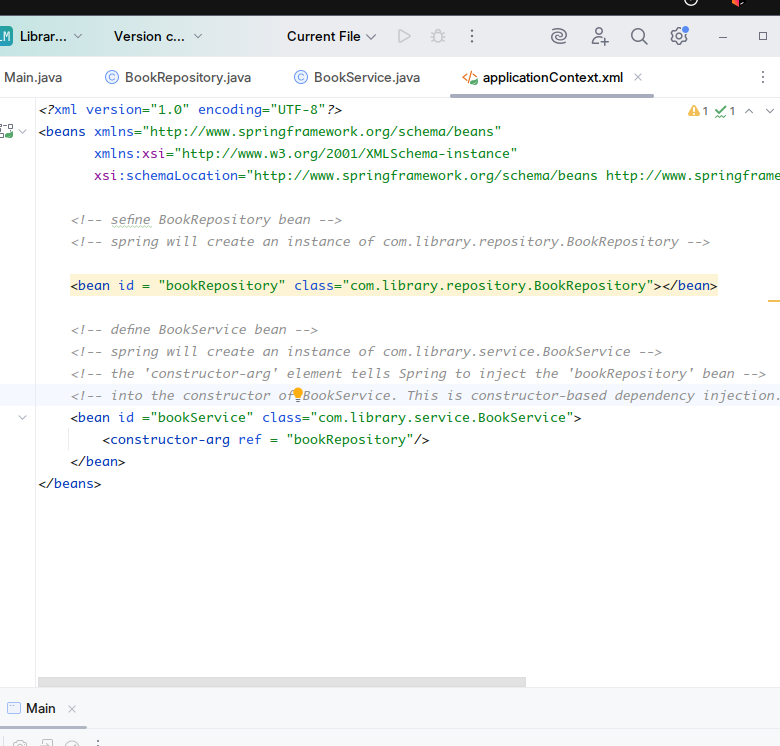
|  |
| --- |
| *Exercise 2: Implementing Dependency Injection (****Solution****)* |

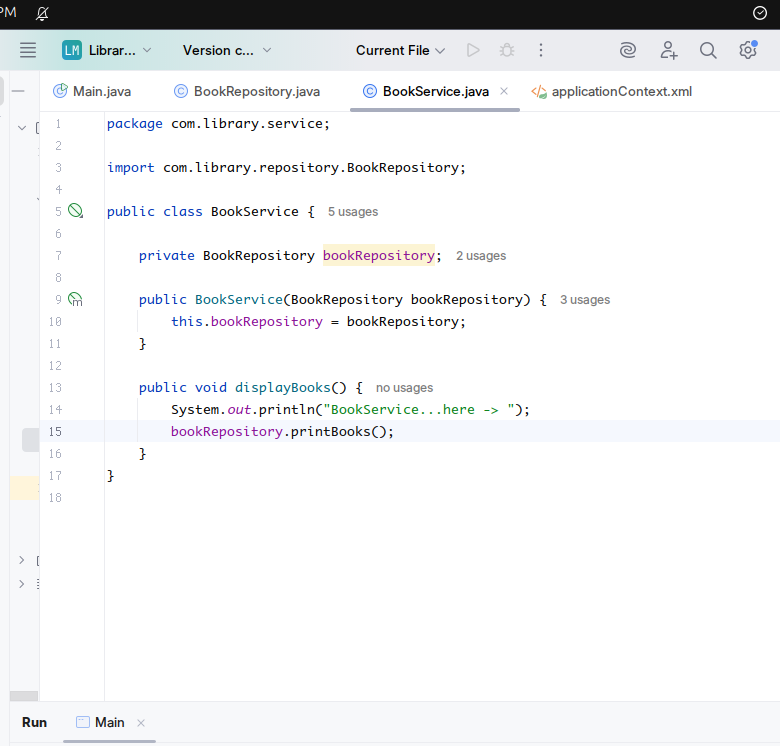
1. **Modify the XML Configuration:**

**In our case, we have previously modified applicationcontext.xml, which was already configured to use constructor based DI.**

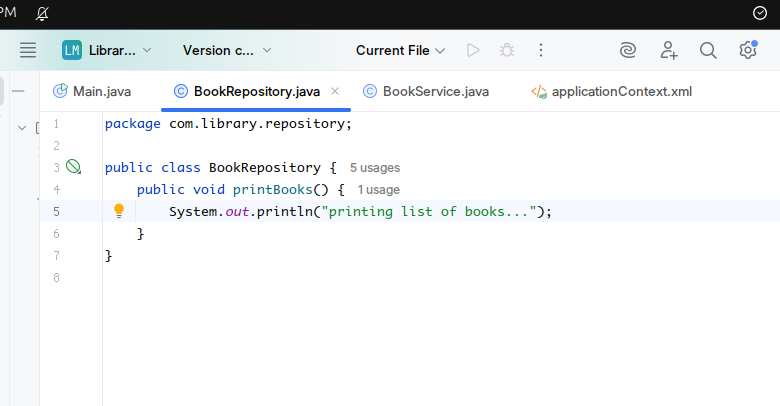
****

**2. Update the BookService Class:**

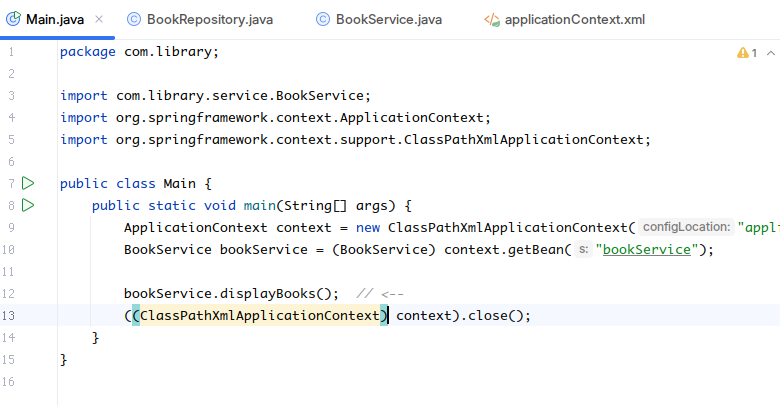
**We update the BookService class to instantiate wired repository object. Then use it to implement business logic.**

****

**Here the printBooks(...) method is already from the repository class.**

****

**3. Test the Configuration:**



**Creating the Spring Container** ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
 This line initializes the Spring IoC container by loading the XML configuration file applicationContext.xml from the classpath. It reads the bean definitions and instantiates all the beans configured there. This is where dependency injection (DI) actually happens—Spring sees the constructor argument for BookService and injects an instance of BookRepository.

**Retrieving a Bean** BookService bookService = (BookService) context.getBean("bookService");  
 This retrieves the BookService bean that Spring created and injected earlier. Since the bean was configured using constructor-arg, the injected BookRepository is already wired in when this object is returned.

**Using the Injected Bean** bookService.displayBooks();  
 Here, you call a method (displayBooks) on the BookService instance. Inside that method, it internally calls a method (like printBooks) on the BookRepository, demonstrating that the injection worked as

expected.



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*