**Exercise 11: Implementing Dependency Injection**

* **Create a New Java Project:** A new Java project named *DependencyInjectionExample* should be created. This serves as the container for all the classes and interfaces we'll define.
* **Define Repository Interface:** An interface named *CustomerRepository* should be created. This interface defines the contract for customer data operations, including a method called *findCustomerById().* This method is responsible for retrieving a customer based on their ID.
* **Implement Concrete Repository**: A class named *CustomerRepositoryImpl* should be created. This class implements the *CustomerRepository* interface. It provides the actual implementation of the *findCustomerById()* method, typically involving some form of data storage (like a database or, in our example, an in-memory collection).
* **Define Service Class:** A class named *CustomerService* should be created. This class depends on the *CustomerRepository* interface to perform its operations. It acts as a higher-level service that uses the repository to retrieve customer data.
* **Implement Dependency Injection:** Constructor injection should be used to inject the *CustomerRepository* into the *CustomerService*. This means the *CustomerService* constructor should take a *CustomerRepository* as a parameter, allowing different implementations of *CustomerRepository* to be injected.
* **Test the Dependency Injection Implementation:** A main class (in our case, *DependencyInjectionExample*) should be created to demonstrate the usage of Dependency Injection. It should show how to create a *CustomerService* with a *CustomerRepositoryImpl* and use it to find a customer.
* **Additional classes:**
* Customer: A simple class representing a customer with properties like id and name.

The implementation demonstrates the principle of Dependency Injection, where the *CustomerService* depends on an abstraction (*CustomerRepository* interface) rather than a concrete implementation. This allows for flexibility and easier testing, as different implementations of *CustomerRepository* can be injected into *CustomerService* without changing its code.

The main class showcases how to wire these components together, creating a *CustomerRepositoryImpl*, injecting it into a *CustomerService*, and then using the service to find customers based on user input.

This structure adheres to the Dependency Inversion Principle, one of the SOLID principles of object-oriented design, by depending on abstractions (interfaces) rather than concrete implementations.

Here is the output of the code -  
