cnfsad-2 script for lab1

Lab Program - 1

Demonstrate Dependency Injection using annotation based using Spring boot

Step 1: Setting Up the Project on Spring Initializer:

Navigate to 'start.spring.io', choose Java, Spring Boot 3.2.2, and configure the project details. Hit generate and download the project zip. Import it into Eclipse.

```
# Example configuration:
Project: Maven
Language: Java
Spring Boot: 3.2.2 (SNAPSHOT)
Group: com.lab1
Artifact: my_lab1
Name: my_lab1
Description: Lab Program 1
Package name: com.lab1.my_lab1
Packaging: Jar
Java: 17
```

Step 2: Creating Customer and Ticket Classes:

In Eclipse, create two classes - `Customer` and `Ticket`.

```
// Customer.java
package com.lab1.my_lab1;

public class Customer {
    String name, address;
    Ticket ticket_instance;

    // Getters and setters...
}

// Ticket.java
package com.lab1.my_lab1;

public class Ticket {
```

```
    int ticket_number, seat_number, price;
    String ticket_type;

    // Getters and setters...
}
```

Step 3: Writing XML Configuration for Spring Container:

Create `testBoot.xml` to define Spring beans for `Customer` and `Ticket` classes.

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="customer" class="com.lab1.my_lab1.Customer" scope="prototype">
        <property name="ticket_instance" ref="ticket" />
    </bean>

    <bean id="ticket" class="com.lab1.my_lab1.Ticket" scope="prototype">
    </bean>

</beans>
```

Glossary:

- **Bean:** An object that is managed by the Spring IoC (Inversion of Control) container.
- **ID:** A unique identifier assigned to a bean within the Spring IoC container, allowing for bean retrieval.
- **Class:** The fully qualified name of the Java class that the Spring IoC container will instantiate to create a bean.
- **Scope:** Defines the lifecycle and visibility of a bean. Common scopes include "singleton" (default, one instance per container), and "prototype" (a new instance for each request).
- **Property:** A characteristic or attribute of a bean that is set during its instantiation, often specified in the bean configuration.
- **Name:** An alternative to ID, used as a symbolic name for a bean within the Spring IoC container.
- **Ref:** Stands for reference. It's used in bean configuration to refer to another bean by its ID or name. Used in property injection to establish dependencies between beans.

Property Injection and Scope Definition in XML Configuration:

Utilize the `<property>` element within each bean definition to inject properties. Set the scope of both beans to "prototype."

```
<bean id="customer" class="com.lab1.my_lab1.Customer" scope="prototype">
    <property name="ticket_instance" ref="ticket" />
</bean>

<bean id="ticket" class="com.lab1.my_lab1.Ticket" scope="prototype">
</bean>
```

Ensure that the property names specified in `<property>` match the actual property names in `Customer.java`. The scope definition is crucial for Spring to manage dependency injection correctly.

Property Injection:

Utilize the `<property>` element within each bean definition to inject properties. In this case, `Customer` has a property called `ticket_instance`, which is set to reference the `ticket` bean.

Scope Definition:

Consider the scope of your beans. In this example, both `customer` and `ticket` have a scope of "prototype," meaning a new instance is created each time they are requested.

Ensure that:

- The id in the `<bean>` definition for `ticket` should match the `ref` attribute in the `<property>` definition inside the `customer` bean.
- The `name` attribute in `<property>` should match the variable name in the `Customer.java` class.

Here, `ticket_instance` is the property name that we are referring to in the XML configuration and is crucial for Spring to correctly manage the dependency injection between beans.

Step 4: Using Dependency Injection in the Main Class:

Initialize the Spring application context using `ClassPathXmlApplicationContext` and specify the name of the XML file.

```
// MyLab1Application.java
package com.lab1.my_lab1;

import java.util.Scanner;
import java.util.InputMismatchException;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

@SpringBootApplication
```

```
public class MyLab1Application {

    public static void main(String[] args) {
        SpringApplication.run(MyLab1Application.class, args);

        Scanner scan = new Scanner(System.in);

        ApplicationContext ac = new
ClassPathXmlApplicationContext("testBoot.xml");
        Customer c = (Customer) ac.getBean("customer");
        Ticket t = (Ticket) c.getTicket_instance();

        while (true) {

System.out.println("\n****************************************");
            System.out.println("Ticket Management System");
            System.out.println("1. Insert\n2. Display \n3. Exit");
            System.out.println("****************************************");
            System.out.print("Enter your choice: ");
            int choice = scan.nextInt();

            switch (choice) {
                case 1:
                    System.out.println("\nInsert Customer Detials");
                    System.out.print(" - Enter name: ");
                    c.setName(scan.next());
                    System.out.print(" - Enter Address: ");
                    c.setAddress(scan.next());
                    System.out.println("\nInsert Ticket Detials");
                    t.setTicket_number(scan.nextInt(" - Enter Ticket Number:
"));
                    System.out.print(" - Enter Ticket
Type(economical/business): ");
                    t.setTicket_type(scan.next());
                    t.setSeat_number(scan.nextInt(" - Enter Seat Number:
"));
                    t.setPrice(scan.nextInt(" - Enter Ticket Price: "));
                    System.out.print("\nDetails inserted successfully");
                    System.out.println();
                    break;

                case 2:
                    System.out.println("\nCustomer Detials");
```

```
                    System.out.println(" - Name: " + c.getName());
                    System.out.println(" - Address: " + c.getAddress());
                    System.out.println("\nTicket Detials");
                    System.out.println(" - Ticket Number: " +
t.getTicket_number());
                    System.out.println(" - Ticket Type: " +
t.getTicket_type());
                    System.out.println(" - Seat Number: " +
t.getSeat_number());
                    System.out.println(" - Ticket Price: " + t.getPrice());
                    break;

                case 3:
                    System.out.println("\nExiting...");
                    System.exit(0);

                default:
                    System.out.println("\nInvalid Choice");
                    break;
            }
        }
    }
}
```

Bean Retrieval:

Retrieve beans from the application context using their IDs.

```
Customer c = (Customer) ac.getBean("customer");
Ticket t = (Ticket) ac.getBean("ticket");
```

The Spring application context is used to get instances of the `Customer` and `Ticket` beans.

The injected `Customer` bean has a reference to the `Ticket` bean, establishing the dependency.

Conclusion:

The provided program demonstrates Dependency Injection using annotation-based Spring Boot. It involves setting up the project, creating classes, configuring a Spring Container using XML, and utilizing Dependency Injection in the main class. Ensure consistency in IDs, package names, and class attributes for successful execution. Happy coding!
