Tutorial 13 – Spring Core Framework

Description

Spring is well-known for being a web development framework, but it doesn't mean Spring is not useful for desktop or command-line applications. Using Spring Boot eliminates the need for most of the configuration but also makes it difficult to understand the core concepts in Spring such as Dependency Injection and Aspect-Oriented Programming.

In this tutorial, you will practice with these concepts (DI and AOP) in the core Spring framework by developing a simple console application.

Instructions

Part 1: Setting up project

See **Lecture 13** for instructions.

Part 2: Dependency Injection

(1) Create a configuration class. A configuration class should be marked with the `@Configuration` annotation. It should also be marked with the `@ComponentScan` annotation to specify where Spring should look for components. Example configuration class:

```
@Configuration
@ComponentScan("tutes.spring")
public class Config {}
```

(2) Create these component classes (mark them with the `@Component` annotation):

a. `Customer` : a POJO with two attributes: `name` and `phone`.

b. `CustomerManager` : a service class which stores customers in a `List<Customer>` and provides these methods to manipulate that list (just imagine this class being in charge of storing/retrieving customers in/from a database):

- `addCustomer(Customer c): boolean`
- `removeCustomer(Customer c): boolean`
- `findCustomerByName(String name): Customer`
  Returns `null` if not found.
- `getCustomers(): Customer[]`

b. `CustomerController` : a controller class which uses `CustomerManager` to add, remove and display a list of customers. This class should have the `CustomerManager` object auto-injected by Spring using the `@Autowired` annotation. It should have these methods:

- `add(String name, String phone): void`
  Adds this customer if he is not found by name in this application's data storage.
- `remove(String customerName): void`
  Removes a customer (if found) from the application's data storage.

- `index(): String`
  Returns a list of customers as a String.

(3) Create a `Main` class which acts as the application's entry point. This class should initialize the application's context and use the `CustomerController` class to add and remove some customers, as well as display a list of all customers. Here's an example of this entry point class:

```
public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(Config.class);
        CustomerController controller =
                context.getBean(CustomerController.class);
        controller.add("Quan", "0982496005");
        controller.add("Nam", "0123456789");
        System.out.println(controller.index());
        controller.remove("Nam");
        System.out.println(controller.index());
    }
}
```

Part 3: Aspect-Oriented Programming

(1) Also mark the configuration class with `@EnableAspectJAutoProxy` annotation to enable annotation-based AOP.

(2) Create a `Logger` class and mark it with both `@Component` and `@Aspect` annotations so that Spring recognizes it as an aspect. In this aspect class, we define the `Pointcut`(s) and also the `Advice`(s). In this application, you have to:

a. Create a `Pointcut` which captures the execution of the `addCustomer` method in the `CustomerManager` class.

b. Create an advice which executes *before* the above `Pointcut`. This advice should print a log message on the console screen (such as: "A new customer is going to be added…").

c. Create another advice which executes *after* the `getCustomers` method in `CustomerManager` but this advice doesn't need a Pointcut. It uses a Pointcut expression directly.