Understanding Spring AOP

Written by Nam Ha Minh
Last Updated on 24 June 2019 | Print Email

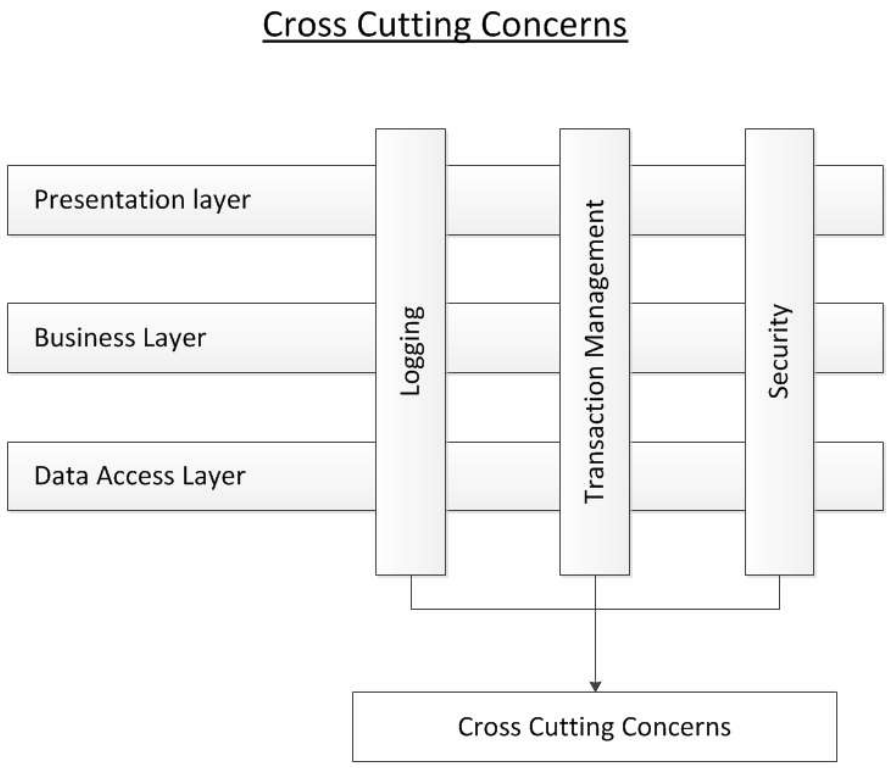
Table of Content

1. Overview

In this article we will learn about the *Aspect Oriented Programming (AOP)* and various terminologies associated with it. Then, we will look how the Spring framework provides the capability to implement various *cross cutting concerns* of the application through AOP. Lastly, we will then write a sample application with focus on Spring AOP.

2. Cross cutting concerns

In any enterprise application, there are a number of concerns which need to be taken care of in addition to the main business logic. These concerns are spread across the application and into multiple application layers. Such concerns are logging, transaction handling, performance monitoring, security etc. These concerns are known as *cross cutting concerns* of the application.



Cross Cutting Concerns

AOP help is implementing the cross cutting concerns of the application keeping them separate from the main business logic and thus resulting in *loosely coupled applications*.

3. Need of AOP

Before going in to details of AOP we must understand the need of AOP. Let us take an example of a java class `LibraryService` which has two methods `issueBook` and `returnBook`. The requirement is to log the request parameters and the response value. Here logging is our cross cutting concerns which we want to implement in our application. The code of `LibraryService` looks as below:

```
1  public class LibraryService {
2      public boolean issueBook(int memberID, int bookID) {
3          System.out.println("Executing method issueBook("+memberID+","+bookID-
4
5          boolean status = false;
6          // Business logic to issue a book from Library
7
8          System.out.println("Returning from method issueBook of LibraryServic
9          return status;
10     }
11     public boolean returnBook(int memberID, int bookID) {
12         System.out.println("Executing method returnBook("+memberID+","+bookID
13
14         boolean status = false;
15         // Business logic to return the issues book
16
17         System.out.println("Returning from method returnBook of LibraryServi
18         return status;
19     }
20 }
```

Now, the requirement comes to add a new book in the Library and thus we need another method `addBook` in the `LibraryService`. While writing the method `addBook`, we need to ensure that we log the request parameters and the response value. Thus the code of `addBook` method looks as below:

```
1  public boolean addBook(int bookID) {
2      System.out.println("Executing method addBook("+bookID+") of LibraryServic
3
4      boolean status = false;
5      // Business logic to return the issues book
6
7      System.out.println("Returning from method addBook of LibraryService : "+st
8      return status;
9  }
```

Drawbacks of the above conventional approach:

1. Similar logic of logging the request parameters and the response value is spread across multiple methods and thus creating redundant code. This makes maintenance very difficult.
2. Any change in the requirement of application logging will result in changing the code of multiple methods of `LibraryService` .
3. Adding any new method in the `LibraryService` will result in rewriting the logging code again in the newly added method. Thus we are not able to reuse the existing logging logic.
4. The main responsibility of `LibraryService` is to provide various operations of Library rather than logging. Keeping the code of logging in `LibraryService` is not a good idea.

AOP helps to implement the logging concern (and all other cross cutting concerns) of the application overcoming all of the above drawbacks. AOP keeps the cross cutting concerns separate from the main business logic of the application and weaves them appropriately in the various application object.

Spring AOP is used extensively in Spring's Transaction Management where the transaction handlers are injected or weaved around the method execution join points. AOP is also used in Spring's Security module to secure the method call for authenticated and authorised users only. Note that AOP is hidden behind the Spring's Security namespaces so that the users do not worry about weaving security handlers in application objects - all is done through namespaces.

4. AOP terminologies

Let us now discuss the various terminologies used in an aspect oriented programming. Note that these terminologies not only are specific to Spring AOP but also are used in general for any AOP framework.

Aspect

An *Aspect* is the concern (cross cutting concern) which you want to implement in the application such as logging, performance monitoring, transactional handing etc.

Advice

An *Advice* is the actual implementation of the aspect. Aspect is a concept and Advice is the concrete implementation of the concept.

Join Point

A *JoinPoint* is a point in the execution of the program where an aspect can be applied. It could be before/after executing the method, before throwing an exception, before/after modifying an instance variable etc. Keep in mind that it is not necessary and also not required to apply an aspect at all the available join points. *Spring AOP only supports method execution join points*.

Point cut

PointCuts tell on which join points the aspect will be applied. An advice is associated with a point cut expression and is applied to a join point which matches the point cut expression.

Target

Target is the application object on which the advice will be applied.

Proxy

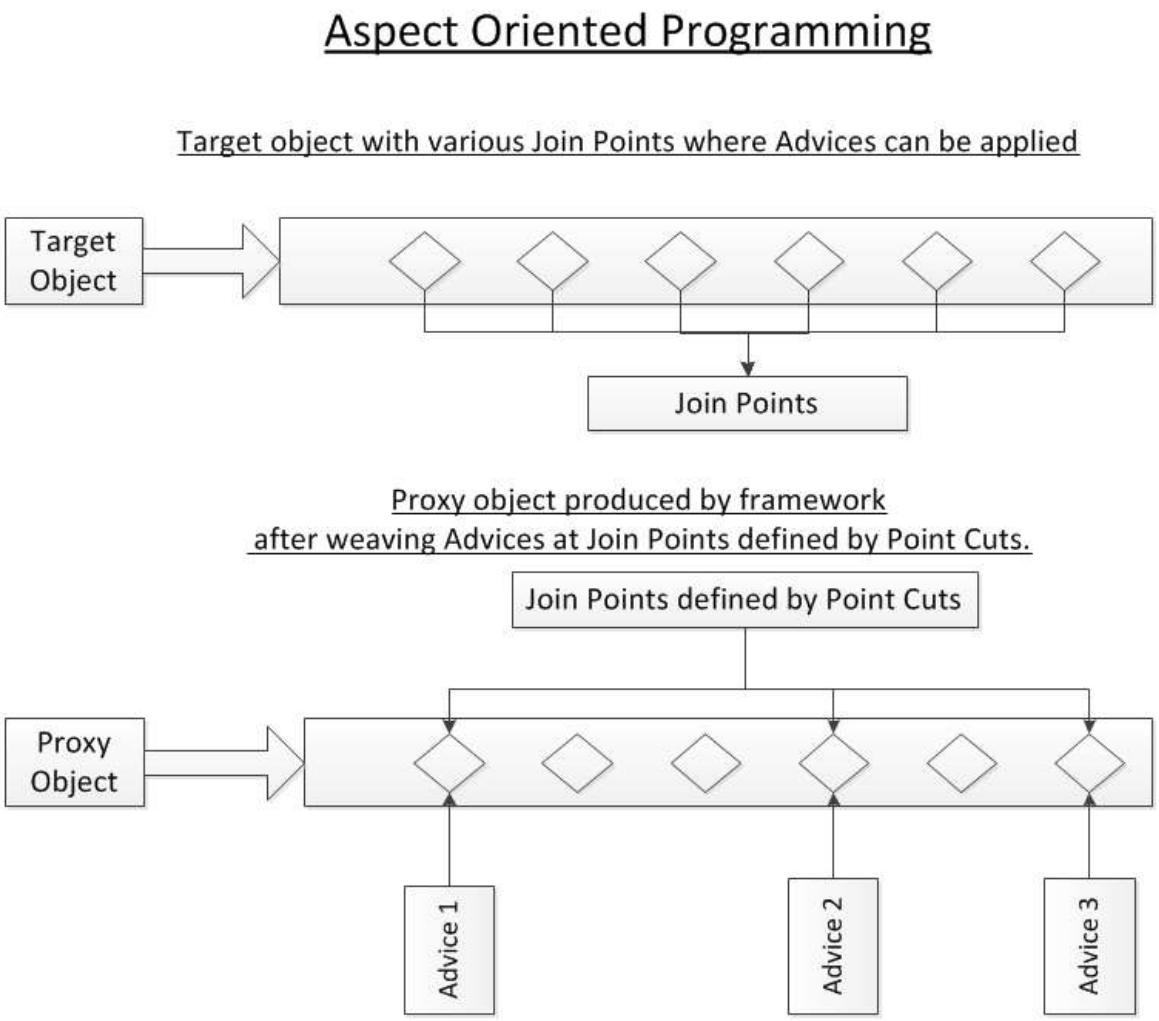
Proxy is the object which is created by the framework after applying the advice on the target object.

Weaving

Weaving is the process of applying the aspect on the target object to product the proxy object. Weaving can be done at compile time, class loading time or runtime. Spring AOP supports weaving at runtime.

Introduction

An *Introduction* enables to add new methods and instances to the target object.



Aspect Oriented Programming

5. Advices

As discussed earlier, Spring only supports method execution join points. The various method execution join points can be:

1. Before a method execution starts

2. After the method execution completes normally
3. After the method throws an exception
4. Around the method execution

Accordingly the various advices which can be applied in method execution join points are

Before Advice

The *Before Advice* gets executed before the actual method execution starts.

In our `Libraryervice`, we wanted to log the following details before the method execution starts.

1. Class name
2. Method name
3. Parameters value

This behaviour was required in all the methods of the `LibraryService` (and also may be in many more application objects across various layers). Thus we will create a Before advice and weave it in `LibraryService` target object at method execution join points.

To implement a Before advice, the java class should implement interface `org.springframework.aop.MethodBeforeAdvice`. Let us assume that Before advice name is `LogInput`. The java code of `LogInput` looks as below:

```
1  public class LogInput implements MethodBeforeAdvice {
2      @Override
3      public void before(Method method, Object[] args, Object target)
4              throws Throwable {
5          String className = target.getClass().getName();
6          String methodName = method.getName();
7
8          System.out.println("Executing method "+methodName+" of class "+classN
9          for(Object parameter: args){
10             System.out.println(parameter.getClass().getName() + " = "+paramet
11         }
12     }
13 }
```

After (returns) Advice

The *After Advice* gets executed after the method execution completes normally.

Again in the `LibraryService`, it is required to log the actual value of the return parameter along with the class name and method name. This was required as soon as the method execution completes normally. Thus we will create an After advice and weave it in `LibraryService` target at method execution join points.

To implement an After advice, the java class should implement interface `org.springframework.aop.AfterReturningAdvice`. Let us assume that the name of After advice is `LogOutput`. The java code of `LogOutput` looks as below:

```
public class LogOutput implements AfterReturningAdvice {
    @Override
    public void afterReturning(Object returnValue, Method method, Object[] arg
            Object target) throws Throwable {
        String className = target.getClass().getName();
        String methodName = method.getName();
        System.out.println("Returning from method "+methodName+" of class "+cl
    }
}
```

After (throws) Advice

This advice gets executed after the method completes abnormally by throwing an exception.

It is also required to log the exception details in case the method completes abnormally by throwing an exception. Thus we will create an after throws advice.

```
public class LogException implements ThrowsAdvice {
    public void afterThrowing(Method method, Object[] args, Object target, Exc
        String className = target.getClass().getName();
        String methodName = method.getName();

        System.out.println("Throwing exception from method "+methodName+" of c
        System.out.println("Exception message is "+ex.getMessage());
    }
}
```

After (finally)Advice

After (finally) advice gets executed on completion of the method execution. The method can get completed normally or abnormally (by throwing an exception). In both cases, the After advice finally gets executed.

Around advice

This advice gets executed around the method execution i.e. before the method execution starts and after the method execution completes. This is quite a powerful advice which can even decide whether to execute the actual method or not. It can also completely change the behaviour of the actual method by providing its own implementation for that method.

Let us assume that we need to find the time taken by the method to complete the execution. Thus we need to capture the time at the start of the method execution and the time at the end of the method execution. The total time can then be calculated as the difference in the two times. Thus we need something done before the method execution starts and something after method execution complete. Around advice is best to support this requirement. To implement an *Around Advice*, the java class must implement the interface `org.aopalliance.intercept.MethodInterceptor`.

```
public class PerformanceMonitoring implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable
        long startTime = System.currentTimeMillis();
        Object result = methodInvocation.proceed();
        long endTime = System.currentTimeMillis();
        System.out.println("Total time taken in ms : "+(endTime-startTime));
        return result;
    }
}
```

6. PointCuts and PointCutAdvisors

In the previous section we have created the various Advices which can be applied at the join points. The *PointCuts* will tell which advice to apply at which join point. So in this section we will learn how to define the point cuts. As we are aware that Spring only supports method execution join points so the point cuts will tell for which all methods the advice will be applied.

NameMatchMethodPointcut

This point cuts tell the names of the methods on which the advice will be applied. All the methods of the target which matches the methods names given in the point cuts are eligible for advice to be applied.

The following point cut tells that the advice will be applied only on method with name `issueBook` of the target object. Note that the point cut does not tell about on which target object the advice will be applied.

```
<bean id="myPointcut"
    class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedName" value="issueBook" />
</bean>
```

The following point cut tells that the advice will be applied only on methods with name `issueBook` and `returnBook` of the target object.

```
<bean id="myPointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
      <property name="mappedNames">
            <list>
                  <value>issueBook</value>
                  <value>returnBook</value>
            </list>
      </property>
</bean>
```

The following point cut tells that the advice will be applied to all the method of the target object which ends with Book. Thus for the `LibraryService` target the advice will be applied on methods `issueBook`, `returnBook` and `addBook`.

```
<bean id="myPointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
      <property name="mappedName">
                  <value>*Book</value>
            </list>
      </property>
</bean>
```

The point cut only tells about methods of the target object on which the advice is to be applied. It does not tell about which advice is to be applied. Here the Spring's point cut advisor comes into play. The *PointCutAdvisor* encapsulates the point cut and the advice into a single object.

NameMatchMethodPointcutAdvisor

This advisor encapsulates the name method point cut and the advice which is to be applied.

```
<bean id="myPointcutAdvisor" class="org.springframework.aop.support.NameMatchM
    <property name="advice" ref="performanceMonitoring"/>
    <property name="mappedName">
        <value>*Book</value>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor

This advisor helps to define a point cut using regular expression and simultaneously encapsulating it with the advice.

```
<bean id="myPointcutAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice" ref="performanceMonitoring"/>
    <property name="pattern" value=".*Book" />
</bean>
```

7. Proxy Objects

In the previous sections we have learned how to create advice and how to define point cut advisors. In this section we will learn how to weave the advices on the join points defined by point cut advisors on the target objects.

After weaving the advice in the target object we get the *Proxy* object. The proxy objects are generated via Spring's `ProxyFactoryBean`. It takes target object and the point cut advisors as input and produces proxy object.

```
<bean id="libraryServiceProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
            <ref local="libraryServiceTarget" />
      </property>
      <property name="interceptorNames">
            <list>
                  <value>myPointCutAdvisor</value>
            </list>
      </property>
</bean>
```

8. Sample Application

Overview

Finally putting all the bits and pieces together for our Sample application. In our sample application we have a `LibraryService` class which provides various functionality for the Library such as issuing books, returning books and adding new books to the library. For each of the functionality provided by the `LibraryService`, we have a corresponding method `issueBook`, `returnBook` and `addBook`. The aspects which we want to implement are the logging, performance monitoring and exception handling. Therefore, the requirement is:

1. Log the input parameters in each method of the `LibraryService` class along with class and method name.
2. Log the return value of each method of the `LibraryService` class along with class and method name.
3. In case of an exception, log the exception message.
4. Log the total time taken by each method. (To monitor the performance)

The various jar files which are required to build and run our sample application based on AOP are as below:

1. spring-aop-3.2.0.M2.jar
2. spring-aspects-3.2.0.M2.jar
3. spring-beans-3.2.0.M2.jar
4. spring-context-3.2.0.M2.jar
5. spring-context-support-3.2.0.M2.jar
6. spring-expression-3.2.0.M2.jar
7. spring-aop-3.2.0.M2.jar
8. spring-instrument-3.2.0.M2.jar

The above jars can be downloaded from following location:

https://repo.springsource.org/libs-milestone-local/org/springframework/spring/3.2.0.M2/spring-3.2.0.M2-dist.zip

Download the zip file from above location. Unzip it. Copy the relevant jars (as mentioned above) from spring-3.2.0.M2\libs to the lib folder of the sample java application.

We also need two following jars:

1. aopalliance.jar
2. commons-logging-1.1.1.jar

Coding the Target

Let us look at our `LibraryService` class. Note that the `LibraryService` does not implement any of the above requirements. We will weave the various aspects in this target class via AOP.

```
package net.codejava.frameworks.spring.aop.service;

public class LibraryService {
    public boolean issueBook(int memberID, int bookID) {
        boolean status = false;
        // Business logic to issue a book from Library
        return status;
    }

    public boolean returnBook(int memberID, int bookID) {
        boolean status = false;
        // Business logic to return the issues book
        return status;
    }

    public boolean addBook(int bookID) {
        boolean status = false;
        // Business logic to return the issues book
        return status;
    }
}
```

Declaring the target object in Spring's application context XML file.

```
<bean id="libraryServiceTarget"
    class="net.codejava.frameworks.spring.aop.service.LibraryService" />
```

Coding the Advices

Now, let us write the various advices which will implement the required aspects.

1. `LogInput` advice – This advice will be implemented as the *Before Advice* which will log the parameter values along with class and method name.
2. `LogOutput` advice – This advice will be implemented as the *After Advice* which will log the return value along with the class and method name.
3. `LogException` advice – This advice will be implemented as *After Throws Advice* which will log the exception message.
4. `PerfromanceMonitoring` advice – This advice will be implemented as the *Around Advice* which will log the time taken by the method to execute.

Let us have a quick look at the code of various advices.

```java
package net.codejava.frameworks.spring.aop.advice;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class LogInput implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        String className = target.getClass().getName();
        String methodName = method.getName();

        System.out.println("Executing method "+methodName+" of class "+classN
        for(Object parameter: args){
            System.out.println(parameter.getClass().getName() + " = "+paramet
        }
    }
}
```

```java
package net.codejava.frameworks.spring.aop.advice;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class LogOutput implements AfterReturningAdvice {
    @Override
    public void afterReturning(Object returnValue, Method method, Object[] ar
            Object target) throws Throwable {
        String className = target.getClass().getName();
        String methodName = method.getName();
        System.out.println("Returning from method "+methodName+" of class "+c
    }
}
```

```java
package net.codejava.frameworks.spring.aop.advice;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;

public class LogException implements ThrowsAdvice {
    public void afterThrowing(Method method, Object[] args, Object target, Ex
        String className = target.getClass().getName();
        String methodName = method.getName();

        System.out.println("Throwing exception from method "+methodName+" of
        System.out.println("Exception message is "+ex.getMessage());
    }
}
```

```
package net.codejava.frameworks.spring.aop.advice;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class PerformanceMonitoring implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable
        long startTime = System.currentTimeMillis();
        Object result = methodInvocation.proceed();
        long endTime = System.currentTimeMillis();
        System.out.println("Total time taken in ms : "+(endTime-startTime));
        return result;
    }
}
```

Declaring the four advices in Spring's application context XML file.

```
<bean id="logInputAdvice"
    class="net.codejava.frameworks.spring.aop.advice.LogInput" />

<bean id="logOutputAdvice"
    class="net.codejava.frameworks.spring.aop.advice.LogOutput" />

<bean id="logExceptionAdvice"
    class="net.codejava.frameworks.spring.aop.advice.LogException" />

<bean id="performanceMonitoringAdvice"
    class="net.codejava.frameworks.spring.aop.advice.PerformanceMonitoring"
```

Coding the PointCutAdvisors

Now, we need *PointCutAdvisors* which will tell for which all methods (join points) each of the advice will be applied to.

Thus we need four PointCutAdvisors for each of the four advices created in previous section. We will use the `NameMatchMethodPointcutAdvisor`.

Declaring the four PointCutAdvisors in Spring's application context XML file.

```
<bean id="logInputPointcutAdvisor"
    class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advice" ref="logInputAdvice" />
    <property name="mappedName">
        <value>*Book</value>
    </property>
</bean>

<bean id="logOutputPointcutAdvisor"
    class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advice" ref="logOutputAdvice" />
    <property name="mappedName">
        <value>*Book</value>
    </property>
</bean>

<bean id="logExceptionPointcutAdvisor"
    class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advice" ref="logExceptionAdvice" />
    <property name="mappedName">
        <value>*Book</value>
    </property>
</bean>

<bean id="performanceMonitoringPointcutAdvisor"
    class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="advice" ref="performanceMonitoringAdvice" />
    <property name="mappedName">
        <value>*Book</value>
    </property>
</bean>
```

Coding the Proxy

Finally we need to weave the advices in the target object to produce the proxy object. We will use the Spring's `ProxyFactoryBean` to generate the proxy object.

```
<bean id="libraryServiceProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref local="libraryServiceTarget" />
    </property>
    <property name="interceptorNames">
        <list>
            <value>logInputPointcutAdvisor</value>
            <value>logOutputPointcutAdvisor</value>
            <value>logExceptionPointcutAdvisor</value>
            <value>performanceMonitoringPointcutAdvisor</value>
        </list>
    </property>
</bean>
```

Spring's Application Context file

Finally our Spring's application context file (*myLibraryAppContext.xml*) looks as below:

```
<?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
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="libraryServiceTarget"
        class="net.codejava.frameworks.spring.aop.service.LibraryService" />
```

```
    <bean id="logInputAdvice"
        class="net.codejava.frameworks.spring.aop.advice.LogInput" />

    <bean id="logOutputAdvice"
        class="net.codejava.frameworks.spring.aop.advice.LogOutput" />

    <bean id="logExceptionAdvice"
        class="net.codejava.frameworks.spring.aop.advice.LogException" />

    <bean id="performanceMonitoringAdvice"
        class="net.codejava.frameworks.spring.aop.advice.PerformanceMonitorin

    <bean id="logInputPointcutAdvisor"
        class="org.springframework.aop.support.NameMatchMethodPointcutAdviso
        <property name="advice" ref="logInputAdvice" />
        <property name="mappedName">
            <value>*Book</value>
        </property>
    </bean>

    <bean id="logOutputPointcutAdvisor"
        class="org.springframework.aop.support.NameMatchMethodPointcutAdviso
        <property name="advice" ref="logOutputAdvice" />
        <property name="mappedName">
            <value>*Book</value>
        </property>
    </bean>

    <bean id="logExceptionPointcutAdvisor"
        class="org.springframework.aop.support.NameMatchMethodPointcutAdviso
        <property name="advice" ref="logExceptionAdvice" />
        <property name="mappedName">
            <value>*Book</value>
        </property>
    </bean>

    <bean id="performanceMonitoringPointcutAdvisor"
        class="org.springframework.aop.support.NameMatchMethodPointcutAdviso
        <property name="advice" ref="performanceMonitoringAdvice" />
        <property name="mappedName">
            <value>*Book</value>
        </property>
    </bean>

    <bean id="libraryServiceProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref local="libraryServiceTarget" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>logInputPointcutAdvisor</value>
                <value>logOutputPointcutAdvisor</value>
                <value>logExceptionPointcutAdvisor</value>
                <value>performanceMonitoringPointcutAdvisor</value>
            </list>
        </property>
    </bean>

</beans>
```

Coding the Client

Now time to test our sample application. The client code will obtain an object of the proxy (of `LibraryService` target) via Spring container and then calls the various methods of `LibraryService`.

```java
package net.codejava.frameworks.spring.aop.client;

import net.codejava.frameworks.spring.aop.service.LibraryService;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LibraryServiceClient {
    public static void main(String[] args){
        ApplicationContext applicationContext = new ClassPathXmlApplicationC
        LibraryService myLibraryService = (LibraryService) applicationContex
        myLibraryService.issueBook(1, 1);
        myLibraryService.returnBook(2, 2);
        myLibraryService.addBook(3);
    }
}
```

Output

Executing the above client code produces following result on the console. I have marked the output on console with different colours for illustration purpose only. The lines marked with blue colour are being produced by `LogInput` advice (Before Advice). The lines marked with red colour are produced by `PerformanceMonitoring` advice (Around Advice) and the lines marked with green colour are produced by `LogOutput` Advice (After Advice).

```
Executing method issueBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with following
parameters

java.lang.Integer = 1

java.lang.Integer = 1

Total time taken in ms : 26

Returning from method issueBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with false


Executing method returnBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with following
parameters

java.lang.Integer = 2

java.lang.Integer = 2

Total time taken in ms : 1

Returning from method returnBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with false
```

```
Executing method addBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with following
parameters

java.lang.Integer = 3

Total time taken in ms : 0

Returning from method addBook of class
net.codejava.frameworks.spring.aop.service.LibraryService with false
```

9. Conclusion

In this article we have learned about the aspect oriented programming, the need of it and the various AOP terminologies. And then, we have learned how AOP can be implemented using Spring framework, finally concluding with a sample application. The sample application is attached with this article. Download the sample application developed on Eclipse IDE version 4.2 (Juno) and try it yourself.

Other Spring Tutorials:

- Understand the core of Spring framework
- Understand Spring MVC
- Spring Dependency Injection Example with XML Configuration
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- 14 Tips for Writing Spring MVC Controller
- Spring Web MVC Security Basic Example (XML Configuration)
- Understand Spring Data JPA with Simple Example

About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on Facebook.

Attachments:

Library_SpringAOP.zip [Library Project using Spring AOP] 2626 kB

Add comment

Name E-mail