Spring AOP

# Spring AOP Advices

In Spring AOP, 4 type of advices are supported :

**Before advice** – Run before the method execution

**After returning advice** – Run after the method returns a result

**After throwing advice** – Run after the method throws an exception

**Around advice** – Run around the method execution, combine all three advices above. It combines all three advices above, and execute during method execution. Create a class which implements MethodInterceptor interface. You have to call the “methodInvocation.proceed();” to proceed on the original method execution, else the original method will not execute.

**Java code for AOP Services**

# Before advice

package com.ddlab.spring.aop.advice;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class BeforeMethodAOP implements **MethodBeforeAdvice**

{

@Override

public void before(Method method, Object[] args, Object target) throws Throwable {

System.out.println("HijackBeforeMethod : Before method hijacked!");

System.out.println("Before Method Execution ....");

System.out.println(target);//Refers to target object

}

}

# After returning advice

package com.ddlab.spring.aop.advice;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class AfterMethodAOP implements **AfterReturningAdvice**

{

@Override

public void afterReturning(Object returnValue, Method method,

Object[] args, Object target) throws Throwable {

System.out.println("HijackAfterMethod : After method hijacked!");

}

}

# After throwing advice

package com.ddlab.spring.aop.advice;

import org.springframework.aop.ThrowsAdvice;

public class AfterThrowingAOPException implements **ThrowsAdvice** {

public void afterThrowing(IllegalArgumentException e) throws Throwable {

System.out.println("HijackThrowException : Thrown exception hijacked!");

}

}

# Around Advice

package com.ddlab.spring.aop.advice;

import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;

import org.aopalliance.intercept.MethodInvocation;

public class AroundMethodAOP implements **MethodInterceptor** {

@Override

public Object invoke(MethodInvocation methodInvocation) throws Throwable {

System.out.println("Method name : "+ methodInvocation.getMethod().getName());

System.out.println("Method arguments : " + Arrays.toString(methodInvocation.getArguments()));

// same with MethodBeforeAdvice

System.out.println("HijackAroundMethod : Before method hijacked!");

try {

// proceed to original method call

Object result = methodInvocation.proceed();

// same with AfterReturningAdvice

System.out.println("HijackAroundMethod : Before after hijacked!");

return result;

} catch (IllegalArgumentException e) {

// same with ThrowsAdvice

System.out.println("HijackAroundMethod : Throw exception hijacked!");

throw e;

}

}

}

A simple service class is given below

package com.ddlab.spring.aop.advice;

public class CustomerService {

private String name;

private String url;

public void setName(String name) {

this.name = name;

}

public void setUrl(String url) {

this.url = url;

}

public void printName() {

System.out.println("Customer name : " + this.name);

}

public void printURL() {

System.out.println("Customer website : " + this.url);

}

public void printThrowException() {

throw new IllegalArgumentException();

}

}

The Test class is given below.

package com.ddlab.spring.aop.advice;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void executeBeforeMethodAOP() {

ApplicationContext appContext = new ClassPathXmlApplicationContext( new String[] { "beans.xml" });

CustomerService beforeCust = (CustomerService) appContext.getBean("beforecustomerServiceProxy");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

beforeCust.printName();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

beforeCust.printURL();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static void executeAfterMethodAOP() {

ApplicationContext appContext = new ClassPathXmlApplicationContext( new String[] { "beans.xml" });

CustomerService afterCust = (CustomerService) appContext.getBean("aftercustomerServiceProxy");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

afterCust.printName();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

afterCust.printURL();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static void executeAfterThrowingMethodAOP() {

ApplicationContext appContext = new ClassPathXmlApplicationContext( new String[] { "beans.xml" });

CustomerService afterThrowingCust = (CustomerService) appContext.getBean("throwingcustomerServiceProxy");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

afterThrowingCust.printName();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

afterThrowingCust.printURL();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

try {

afterThrowingCust.printThrowException();

} catch (Exception e) { }

}

public static void executeAroundServiceMethodAOP() {

ApplicationContext appContext = new ClassPathXmlApplicationContext( new String[] { "beans.xml" });

CustomerService aroundCust = (CustomerService) appContext.getBean("aroundcustomerServiceProxy");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

aroundCust.printName();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

aroundCust.printURL();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static void main(String[] args) {

executeBeforeMethodAOP();

executeAfterMethodAOP();

executeAfterThrowingMethodAOP();

executeAroundServiceMethodAOP();

}

}

Spring XML configuration is given below.

<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-2.5.xsd">

<bean id="customerService" class="com.ddlab.spring.aop.advice.CustomerService">

<property name="name" value="Yong Mook Kim" />

<property name="url" value="http://www.mkyong.com" />

</bean>

<!-- Before Advice -->

**<bean id="beforeMethodAOPBean" class="com.ddlab.spring.aop.advice.BeforeMethodAOP" />**

**<bean id="beforecustomerServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">**

**<property name="target" ref="customerService" />**

**<property name="interceptorNames">**

**<list>**

**<value>beforeMethodAOPBean</value>**

**</list>**

**</property>**

**</bean>**

<!-- After Advice -->

**<bean id="afterMethodAOPBean" class="com.ddlab.spring.aop.advice.AfterMethodAOP" />**

**<bean id="aftercustomerServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">**

**<property name="target" ref="customerService" />**

**<property name="interceptorNames">**

**<list>**

**<value>afterMethodAOPBean</value>**

**</list>**

**</property>**

**</bean>**

<!-- After throwing Advice -->

**<bean id="afterThrowExceptionBean" class="com.ddlab.spring.aop.advice.AfterThrowingAOPException" />**

**<bean id="throwingcustomerServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">**

**<property name="target" ref="customerService" />**

**<property name="interceptorNames">**

**<list>**

**<value>afterThrowExceptionBean</value>**

**</list>**

**</property>**

**</bean>**

<!-- Around Advice -->

**<bean id="hijackAroundMethodBean" class="com.ddlab.spring.aop.advice.AroundMethodAOP" />**

**<bean id="aroundcustomerServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">**

**<property name="target" ref="customerService" />**

**<property name="interceptorNames">**

**<list>**

**<value>hijackAroundMethodBean</value>**

**</list>**

**</property>**

**</bean>**

</beans>

# POINT CUT Example XML Based

package com.kruders.spring.aop;

public interface BusinessService {

void doSomeThing();

}

package com.kruders.spring.aop;

public class BusinessImpl implements BusinessService {

public void doSomeThing() {

System.out.println("Do Something Here");

}

}

package com.kruders.spring.aop;

import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;

import org.aopalliance.intercept.MethodInvocation;

public class BusinessAdvice implements **MethodInterceptor** {

@Override

public Object invoke(MethodInvocation method) throws Throwable {

System.out.println("Method name : " + method.getMethod().getName());

System.out.println("Method arguments : " + Arrays.toString(method.getArguments()));

System.out.println("Around method is called");

System.out.println("Around before is running");

method.proceed();

System.out.println("Around after is running");

return null;

}

}

package com.kruders.spring.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.kruders.spring.aop.BusinessService;

public class Main {

public static void main(String args[]) {

ApplicationContext appContext = new ClassPathXmlApplicationContext("beans2.xml");

BusinessService businessService = (BusinessService)appContext.getBean("businessServiceProxy");

try {

businessService.doSomeThing();

} catch(ArithmeticException ae) {

}

}

}

<?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.xsd">

**<bean id="businessService" class="com.kruders.spring.aop.BusinessImpl" />**

**<!-- Advice -->**

**<bean id="businessAdvice" class="com.kruders.spring.aop.BusinessAdvice" />**

**<bean id="customerPointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">**

**<property name="mappedName" value="doSomeThing" />**

**</bean>**

**<bean id="customerAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">**

**<property name="pointcut" ref="customerPointcut" />**

**<property name="advice" ref="businessAdvice" />**

**</bean>**

**<bean id="businessServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">**

**<property name="target" ref="businessService" />**

**<property name="interceptorNames">**

**<list>**

**<value>customerAdvisor</value>**

**</list>**

**</property>**

**</bean>**

**</beans>**

# AOP Annotation Based

package com.pretech;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class Observer {

/\*\*

\* This method will execute before executing any method in side com.pretech package

\*/

@Before("execution(\* com.pretech.\*.\*(..))")

public void beforeAdvice(**JoinPoint jp**){

System.out.println("Creating object"+jp.getTarget().getClass().getName());

}

/\*\*

\* This method will execute after executing any method

\*/

@After("execution(\* com.pretech.\*.\*(..))")

public void afterAdvice(**JoinPoint jp**){

System.out.println("Created object"+jp.getTarget().getClass().getName());

}

}

package com.pretech;

public class Employee {

public Employee() {

System.out.println("Employee constructor executed");

}

private String name;

private String company;

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public String getCompany() {

return company;

}

public void setCompany(String company) {

this.company = company;

}

}

package com.pretech;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AOPTest {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans3.xml");

Employee employee = (Employee) context.getBean("employee");

employee.getName();

employee.getCompany();

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop" xmlns="http://www.springframework.org/schema/beans"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

<bean id="employee" class="com.pretech.Employee">

<property name="name" value="Raghavan" />

<property name="company" value="wipro" />

</bean>

<bean id="observer" class="com.pretech.Observer" />

</beans>

# AOP Concepts in Spring (Breaking business logic into parts called Concerns)

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, and caching etc.

**Aspect** : A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.

**Join point** : This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.

**Advice** : This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.

**before Run advice before the a method execution.**

**after Run advice after the a method execution regardless of its outcome.**

**after-returning Run advice after the a method execution only if method completes successfully.**

**after-throwing Run advice after the a method execution only if method exits by throwing an exception.**

**around Run advice before and after the advised method is invoked.**

**Pointcut** : This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. Pointcut indicates which method should be intercepted, by method name or regular expression pattern.

**Introduction** : An introduction allows you to add new methods or attributes to existing classes.

**Target object** :The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object.

**Weaving** : Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

**Aspect**- A module which has a set of APIs providing cross-cutting requirements.

**Join point**- This represents a point in your application where you can plug-in AOP aspect

**Advice**- This is the actual action to be taken either before or after the method execution.

**Point cut** - This is a set of one or more join points where an advice should be executed

**Join Points**

While creating the business logic of the method the additional services are needed to be injected (which we saw already) at different places or points, we call such points as join points. At a join point a new services will be added into the normal flow of a business method. While executing the business method, the services are required at the following 3 places (generally), we call them as JoinPoints.

**Before business logic of the method starts**

**After business logic of the method got completed**

**If business logic throws an exception at run time**

**At the join point, an Aspect is injected, nothing but the implementation of Aspect i mean Advice will be injected**

**Difference between Join Point and Point Cuts**

To understand the difference between a join point and pointcut, think of pointcuts as specifying the weaving rules and join points as situations satisfying those rules. In below example,

**@Pointcut("execution(\* \* getName()")**

Pointcut defines rules saying, advice should be applied on getName() method present in any class in any package and joinpoints will be a list of all getName() method present in classes so that advice can be applied on these methods.

A point cut can be considered as a collection of multiple joint points. Joint point specify the particular location where the advice could be implemented, where as pointcut reflects the list of all joint points.

**JoinPoints**: These are basically places in the actual business logic where you wish to insert some miscellaneous functionality that is necessary but not being part of the actual business logic. Some examples of JoinPints are: method call, method returning normally, method throwing an exception, instantiating an object, referring an object, etc...

**Pointcuts**: Pointcuts are something like regular expressions which are used to identify joinpoints. Pontcuts are expressed using "pointcut expression language". Pointcuts are points of execution flow where the cross-cutting concern needs to be applied. There is a difference between **Joinpoint** and **Pointcut**; **Joinpoints are more general and represents any control flow where we 'may choose to' introduce a cross-cutting concern while pointcuts identifies such joinpoints where 'we want to' introduce a cross-cutting concern.**

AOP in Spring

# Basics of AOP in Spring

## Maven Dependencies

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version4.0.2.RELEASE </version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version4.0.2.RELEASE </version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version1.7.5</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version1.0.13</version>

<scope>runtime</scope>

</dependency>

<!-- AspectJ dependencies -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version1.7.4</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjtools</artifactId>

<version1.7.4</version>

</dependency>

</dependencies>

## Spring IOC Configuration – beans.xml

<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-2.5.xsd"*>

<bean id=*"employee"* class=*"com.ddlab.rnd.spring.aop.Employee"*>

<property name=*"name"* value=*"Deb"* />

<property name=*"salary"* value=*"1234"* />

</bean>

<bean id=*"aopBeforeAdvice"* class=*"com.ddlab.rnd.spring.aop.AopBeforeAdvice"*/>

<bean id=*"aopAfterAdvice"* class=*"com.ddlab.rnd.spring.aop.AopAfterAdvice"*/>

<bean id=*"aopAroundAdvice"* class=*"com.ddlab.rnd.spring.aop.AopAroundAdvice"*/>

<bean id=*"aopAfterThrowing"* class=*"com.ddlab.rnd.spring.aop.AopAfterThrowing"*/>

<bean id=*"employeeProxy"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*>

<property name=*"target"* ref=*"employee"*/>

<property name=*"interceptorNames"*>

<list>

<value>aopAfterThrowing</value>

</list>

</property>

</bean>

</beans>

# Java Code

## Employee.java

**package** com.ddlab.rnd.spring.aop;

**public** **class** Employee {

**private** String name;

**private** **int** salary;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** **void** invoke(String name,**int** age,String desgn) {

System.***out***.println("Name----->"+name);

System.***out***.println("Age :::"+age);

System.***out***.println("Designation :::"+desgn);

}

**public** String returnSomeValue(String str) {

**return** str+" : New Value";

}

**public** String getSomeValue(String ss) **throws** Exception {

System.***out***.println("Some Value---->"+ss);

String temp = ss.substring(2);

System.***out***.println("After some value---->"+temp);

**return** temp;

}

}

## AopAfterAdvice.java

package com.ddlab.rnd.spring.aop;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class AopAfterAdvice implements AfterReturningAdvice {

public void afterReturning**(Object returnValue, Method method, Object[] methodArguments,**

**Object targetObject**) throws Throwable {

System.out.println("----------------After Advice-----------------");

System.out.println("returnValue-------->"+returnValue);

System.out.println("Target Object ----->"+targetObject);

System.out.println("Method Name :::"+method.getName());

System.out.println("--------Method Arguments--------------");

for( Object object : methodArguments) {

System.out.println("---->"+object);

}

}

}

## AopAfterThrowing.java

package com.ddlab.rnd.spring.aop;

import java.lang.reflect.Method;

import java.util.Arrays;

import org.springframework.aop.ThrowsAdvice;

public class AopAfterThrowing implements ThrowsAdvice {

public String afterThrowing(**Method method, Object[] args, Object target, Exception ex**) {

String finalVal = null;

System.out.println("----------------After Throwing Advice------------------");

System.out.println("Method name : " + method.getName());

System.out.println("Method arguments : " + Arrays.toString(args));

System.out.println("Target object name : " + target);

return finalVal;

}

}

## AopAroundAdvice

package com.ddlab.rnd.spring.aop;

import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;

import org.aopalliance.intercept.MethodInvocation;

public class AopAroundAdvice implements MethodInterceptor {

public Object invoke(**MethodInvocation methodInvocation**) throws Throwable {

System.out.println("----------------Around Advice------------------");

System.out.println("Method name : " + methodInvocation.getMethod().getName());

System.out.println("Method arguments : " + Arrays.toString(methodInvocation.getArguments()));

Object returnResult = methodInvocation.proceed();

System.out.println("Return result :::"+returnResult);

return returnResult;

}

}

## AopBeforeAdvice.java

package com.ddlab.rnd.spring.aop;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class AopBeforeAdvice implements MethodBeforeAdvice {

public void before(**Method method, Object[] methodArguments, Object targetObject**)

throws Throwable {

System.out.println("----------------Before Advice-----------------");

System.out.println("Method Name :::"+method.getName());

System.out.println("--------Method Arguments--------------");

for( Object object : methodArguments) {

System.out.println("---->"+object);

}

System.out.println("Target Obbject Name :::"+targetObject);

}

}

## App.java

package com.ddlab.rnd.spring.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App

{

public static void main( String[] args )

{

ApplicationContext appContext = new ClassPathXmlApplicationContext(

new String[] { "beans.xml" });

Employee emp = (Employee) appContext.getBean("employeeProxy");

emp.invoke("John", 40, "Software Engineer");

String result = emp.returnSomeValue("abcd");

emp.setName("John");

System.out.println("Result :::"+result);

System.out.println("Emp Name :::"+emp.getName());

System.out.println("Emp Salary :::"+emp.getSalary());

try {

String s = emp.getSomeValue(null);

System.out.println("Final Some Value :::"+s);

} catch (Exception e) {

e.printStackTrace();

}

}

}

[AspectJ](http://en.wikipedia.org/wiki/AspectJ) Pointcut Expressions in [Spring](http://www.springsource.org/) Applications

In [Aspect Oriented Programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming), a pointcut is a set of joinpoints. A joinpoint is a point in program execution where you can add additional behavior. Spring applications only support method based joinpoints. So, you can use AspectJ pointcut expressions to define method pointcuts.

**Method Signature Patterns**

The most typical pointcut expressions are used to match a number of methods by their signatures. A common method based pointcut expression is something like

**expression(<method scope> <return type> <fully qualified class name>.\*(parametes))**

1. method scope: Advice will be applied to all the methods having this scope. For e.g., public, private, etc. Please note that Spring AOP only supports advising public methods.
2. return type: Advice will be applied to all the methods having this return type.
3. fully qualified class name: Advice will be applied to all the methods of this type. If the class and advice are in the same package then package name is not required
4. parameters: You can also filter the method names based on the types. Two dots(..) means any number and type of parameters.

**Examples**

* execution(\* com.aspects.pointcut.DemoClass.\*(..)) : This advice will be applied to all the methods of DemoClass.
* execution(\* DemoClass.\*(..)): You can omit the package if the DemoClass and the advice is in the same package.
* execution(public \* DemoClass.\*(..)): This advice will be applied to the public methods of DemoClass.
* execution(public int DemoClass.\*(..)): This advice will be applied to the public methods of DemoClass and returning an int.
* execution(public int DemoClass.\*(int, ..)): This advice will be applied to the public methods of DemoClass and returning an int and having first parameter as int.
* execution(public int DemoClass.\*(int, int)): This advice will be applied to the public methods of DemoClass and returning an int and having both parameters as int.

### **Type Signature Patterns**

These pointcut expressions are applied to all joinpoint of certain types. A common type signature patterns looks like

**within(type name) :** Here type name is either the package name or the class name.

**Examples**

* within(com.aspects.blog.package.\*) : This will match all the methods in all classes of com.aspects.blog.package.
* within(com.aspects.blog.package..\*) : This will match all the methods in all classes of com.aspects.blog.package and its sub packages. The only difference is the extra dot(.) after package.
* within(com.aspects.blog.package.DemoClass) : This will match all the methods in the DemoClass.
* within(DemoClass) : Again, if the target class is located in the same package as this aspect, the package name can be omitted.
* within(DemoInterface+) : This will match all the methods which are in classes which implement DemoInterface.

<http://docs.spring.io/spring/docs/2.0.x/reference/aop.html#aop-pointcuts>

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

**execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)**

**throws-pattern?)**

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use \* as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the \* wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more). The pattern (\*) matches a method taking one parameter of any type, (\*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the [Language Semantics](http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(public \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

*'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.*

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

*'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

*'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.*

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

*'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

*'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

*'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

*'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.*

# Spring AOP AspectJ Annotation Example

Spring AspectJ AOP implementation provides many annotations:

1. **@Aspect** declares the class as aspect.
2. **@Pointcut** declares the pointcut expression.

The annotations used to create advices are given below:

1. **@Before** declares the before advice. It is applied before calling the actual method.
2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
4. **@Around** declares the around advice. It is applied before and after calling the actual method.
5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.
6. @Pointcut("execution(\* Operation.\*(..))")
7. **private** **void** doSomething() {}

The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

#### Understanding Pointcut Expressions

Let's try the understand the pointcut expressions by the examples given below:

1. @Pointcut("execution(public \* \*(..))")

It will be applied on all the public methods.

1. @Pointcut("execution(public Operation.\*(..))")

It will be applied on all the public methods of Operation class.

1. @Pointcut("execution(\* Operation.\*(..))")

It will be applied on all the methods of Operation class.

1. @Pointcut("execution(public Employee.set\*(..))")

It will be applied on all the public setter methods of Employee class.

1. @Pointcut("execution(int Operation.\*(..))")

It will be applied on all the methods of Operation class that returns int value.

AOP with Annotations

# Maven Dependencies

<dependencies>

<!-- Spring and Transactions -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.0.2.RELEASE</version>

</dependency>

<!-- Logging with SLF4J & LogBack -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.5</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.0.13</version>

<scope>runtime</scope>

</dependency>

<!-- AspectJ dependencies -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.7.4</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjtools</artifactId>

<version>1.7.4</version>

</dependency>

</dependencies>

# Spring IOC config-beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<aop:aspectj-autoproxy />

<bean id=*"account"* class=*"com.ddlab.rnd.aop.AccountImpl"* />

<!-- Before Aspect -->

<bean id=*"beforeAspect"* class=*"com.ddlab.rnd.aop.BeforeAspect"* />

<bean id=*"beforeAspect1"* class=*"com.ddlab.rnd.aop.BeforeAspect1"* />

<!-- After Aspect -->

<bean id=*"afterAspect"* class=*"com.ddlab.rnd.aop.AfterAspect"* />

<bean id=*"afterAspect1"* class=*"com.ddlab.rnd.aop.AfterAspect1"* />

<!-- Around Aspect -->

<bean id=*"aroundAspect1"* class=*"com.ddlab.rnd.aop.AroundAspect1"* />

<!-- After Return Aspect -->

<bean id=*"afterReturnAspect"* class=*"com.ddlab.rnd.aop.AfterReturnAspect"* />

<bean id=*"afterReturnAspect1"* class=*"com.ddlab.rnd.aop.AfterReturnAspect1"* />

<!-- After Throwing Aspect -->

<bean id=*"afterThrowingAspect"* class=*"com.ddlab.rnd.aop.AfterThrowingAspect"* />

<bean id=*"afterThrowingAspect1"* class=*"com.ddlab.rnd.aop.AfterThrowingAspect1"* />

</beans>

# Java Code

## Account.java

**package** com.ddlab.rnd.aop;

**public** **interface** Account {

**public** String openAccount(String name, String ssn, **int** amount);

**public** String closeAccount(String ssn);

**public** **int** withdrawAmount(String acctNo,**int** amount);

**public** String applyLoan(String ssn,**int** loanAmount);

**public** **void** applyTax(String ssn,String panNo) **throws** Exception ;

}

## AccountImpl.java

**package** com.ddlab.rnd.aop;

**public** **class** AccountImpl **implements** Account {

**public** String openAccount(String name, String ssn, **int** amount) {

System.***out***.println("Going to open a new account for "+name);

System.***out***.println("Your SSN : "+ssn);

**return** "123456";

}

**public** String closeAccount(String ssn) {

System.***out***.println("Savings account will be closed for : "+ssn);

**return** "account closed successfully";

}

**public** **int** withdrawAmount(String acctNo, **int** amount) {

**int** availableBalance = 50000;//dummy value

**return** amount;

}

**public** String applyLoan(String ssn, **int** loanAmount) {

System.***out***.println("Your SSN : "+ssn);

System.***out***.println("Your loan amount : "+loanAmount);

**return** "You are eligible for loan";

}

**public** **void** applyTax(String ssn,String panNo) **throws** Exception {

**if**(panNo == **null**) **throw** **new** NullPointerException("Pan No is not available ...");

}

}

## AfterAspect.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AfterAspect {

// **@After("execution(public String com.ddlab.rnd.aop.AccountImpl.closeAccount(..))")**

**@After("execution(\* com.ddlab.rnd.aop.AccountImpl.closeAccount(..))")**

public void doAfter() {

System.out.println("I am doing something afte the actual method execution");

}

}

## AfterReturnAspect.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class AfterReturnAspect {

**@AfterReturning(pointcut = "execution(\* com.ddlab.rnd.aop.AccountImpl.applyLoan(..))", returning= "result")**

public void doAfterReturn() {

System.out.println("I am doing something afte the actual method returns something");

}

}

## AfterAspect1.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AfterAspect1 {

// @After("execution(public String com.ddlab.rnd.aop.AccountImpl.closeAccount(..))")

**@After("execution(\* com.ddlab.rnd.aop.AccountImpl.closeAccount(..))")**

public void doAfter(JoinPoint jp) {

System.out.println("--------------- Let me play with the internals of the methods ---------");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

}

}

## AfterReturnAspect1.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AfterReturnAspect1 {

**@AfterReturning(pointcut = "execution(\* com.ddlab.rnd.aop.AccountImpl.applyLoan(..))", returning= "result")**

public void doAfterReturn(JoinPoint jp, Object result) {

System.out.println("---Let me do something when method returns something---");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

System.out.println("Final return result :::"+result);

}

}

## AfterThrowingAspect.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AfterThrowingAspect {

**@AfterThrowing( pointcut = "execution(\* com.ddlab.rnd.aop.AccountImpl.applyTax(..))", throwing= "error")**

public void doAfterThrowing() {

System.out.println("I am doing something after the method throws exception");

}

}

## AfterThrowingAspect1.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AfterThrowingAspect1 {

**@AfterThrowing( pointcut = "execution(\* com.ddlab.rnd.aop.AccountImpl.applyTax(..))", throwing= "error")**

public void doAfterThrowing(JoinPoint jp,Throwable error) {

System.out.println("---Let me do something when the method throws Exception---");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

System.out.println("Exception Type :::"+error);

}

}

## AroundAspect1.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

**@Aspect**

public class AroundAspect1 {

**@Around("execution(\* com.ddlab.rnd.aop.AccountImpl.withdrawAmount(..))")**

public Object doAround(ProceedingJoinPoint pjp) {

System.out.println("--------------- Let me play with the internals of the methods ---------");

System.out.println("Around before is running!");

Object returnObj = null;

Object[] objects = pjp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = pjp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

try {

returnObj = pjp.proceed();

System.out.println("-----Completed-----");

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("Around after is running!");

return returnObj;

}

}

## BeforeAspect.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

**@Aspect**

public class BeforeAspect {

// **@Before("execution(public String com.ddlab.rnd.aop.AccountImpl.openAccount(..))")**

**@Before("execution(\* com.ddlab.rnd.aop.AccountImpl.openAccount(..))")**

public void doBefore() {

System.out.println("I am doing something before the actual method gets executed");

}

}

## BeforeAspect1.java

package com.ddlab.rnd.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

**@Aspect**

public class BeforeAspect1 {

// **@Before("execution(public String com.ddlab.rnd.aop.AccountImpl.openAccount(..))")**

**@Before("execution(\* com.ddlab.rnd.aop.AccountImpl.openAccount(..))")**

public void doBefore(JoinPoint jp) {

System.out.println("--------------- Let me play with the internals of the methods ---------");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

}

}

## Test.java

package com.ddlab.rnd.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main( String[] args ) {

ApplicationContext appContext = new ClassPathXmlApplicationContext(

new String[] { "beans.xml" });

Account account = (Account) appContext.getBean("account");

// account.openAccount("Deb", "1111", 500);

// account.closeAccount("111");

// int amount = account.withdrawAmount("111", 5000);

// System.out.println(amount);

// account.applyLoan("111", 12345);

try {

account.applyTax("111", null);

} catch (Exception e) {

e.printStackTrace();

}

}

}

AOP PointCut Example

# Spring IOC Config-beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<aop:aspectj-autoproxy />

<!-- <bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/> -->

<bean id=*"account"* class=*"com.ddlab.rnd.spring.aop.AccountImpl"* />

<!-- Before Aspect -->

<bean id=*"beforeAspect"* class=*"com.ddlab.rnd.spring.aop.PointCutBeforeAspect"* />

<!-- After Aspect -->

<bean id=*"afterAspect"* class=*"com.ddlab.rnd.spring.aop.PointCutAfterAspect"* />

<!-- Around Aspect -->

<bean id=*"aroundAspect"* class=*"com.ddlab.rnd.spring.aop.PointCutAroundAspect"* />

</beans>

# Java Code

## Account.java

**package** com.ddlab.rnd.spring.aop;

**public** **interface** Account {

**public** String doItBefore(String ssn);

**public** String doItAfter(String ssn);

**public** String doItAround(String ssn);

**public** String doItAfterThrowing(String ssn) **throws** Exception;

}

## AccountImpl.java

**package** com.ddlab.rnd.spring.aop;

**public** **class** AccountImpl **implements** Account {

**public** String doItBefore(String ssn) {

System.***out***.println("----BEFORE----");

**return** "before : "+ssn;

}

**public** String doItAfter(String ssn) {

System.***out***.println("----AFTER----");

**return** "after : "+ssn;

}

**public** String doItAround(String ssn) {

System.***out***.println("----AROUND----");

**return** "around : "+ssn;

}

**public** String doItAfterThrowing(String ssn) **throws** Exception {

System.***out***.println("----AROUND----");

**if**( ssn == **null** ) **throw** **new** NullPointerException("SSN is null");

**return** "AfterThrowing : "+ssn;

}

}

## PointCutAfterAspect.java

package com.ddlab.rnd.spring.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

**@Aspect**

public class PointCutAfterAspect {

**@Pointcut("execution(\* AccountImpl.doItAfter(..))")**

public void doItAfter(){}//pointcut name

**@After("doItAfter()")//applying pointcut on before advice**

public void doAfter(JoinPoint jp)//it is advice (before advice)

{

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*doItAfter\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("--------------- Let me play with the internals of the methods ---------");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

}

}

## PointCutAroundAspect.java

package com.ddlab.rnd.spring.aop;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class PointCutAroundAspect {

**@Pointcut("execution(\* com.ddlab.rnd.spring.aop.AccountImpl.doItAround(..))")**

public void doItAround(){} //pointcut name

**@Around("doItAround()") //applying pointcut**

public void doAround(ProceedingJoinPoint pjp)//it is advice (before advice) {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*doItAround\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("--------------- Let me play with the internals of the methods ---------");

Object[] objects = pjp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

try {

Object result = pjp.proceed();

System.out.println("Final result :::"+result);

} catch (Throwable e) {

e.printStackTrace();

}

Object targetObject = pjp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

}

}

## PointCutBeforeAspect.java

package com.ddlab.rnd.spring.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

**@Aspect**

public class PointCutBeforeAspect {

// **@Pointcut("execution(\* AccountImpl.\*(..))")**

**@Pointcut("execution(\* com.ddlab.rnd.spring.aop.AccountImpl.doItBefore(..))")**

public void doItBefore(){}//pointcut name

**@Before("doItBefore()")//applying pointcut on before advice**

public void doBefore(JoinPoint jp)//it is advice (before advice)

{

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*doItBefore\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("--------------- Let me play with the internals of the methods ---------");

Object[] objects = jp.getArgs();

for( Object obj : objects ) {

System.out.println("Methd Parameter Values ::: "+obj);

}

Object targetObject = jp.getTarget();

System.out.println("Target Object Name :::"+targetObject);

}

}

## Test.java

package com.ddlab.rnd.spring.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test

{

public static void main( String[] args )

{

ApplicationContext appContext = new ClassPathXmlApplicationContext(

new String[] { "beans.xml" });

Account account = (Account) appContext.getBean("account");

// account.doItBefore("111");

// account.doItAfter("111");

account.doItAround("111");

}

}

**Before**

**MethodBeforeAdvice before(Method m , Object[] obj , Object target)**

**After**

**AfterReturningAdvice afterReturning(Object returnVal,Method m , Object[] obj , Object target)**

**After Throwing**

**ThrowsAdvice afterThrowing(Exception ex)**

**ThrowsAdvice afterThrowing(Method m, Object[] args, Object target, Exception ex)**

**Around Advice**

**MethodInterceptor invoke(MethodInvocation mi)**

How AOP works – JDK Dynamic Proxy and CG lib Proxy

# Maven dependencies

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.1</version>

</dependency>

</dependencies>

# JDK Dynamic Proxy

## Calculator.java

**package** com.ddlab.rnd.dynamic.proxy;

**public** **interface** Calculator {

**public** **int** calculate( **int** a , **int** b);

}

## CalculatorImpl.java

**package** com.ddlab.rnd.dynamic.proxy;

**public** **class** CalculatorImpl **implements** Calculator {

@Override

**public** **int** calculate(**int** a, **int** b) {

**return** a/b;

}

}

## BeforeHandler.java

package com.ddlab.rnd.dynamic.proxy;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

public class BeforeHandler implements InvocationHandler {

private Object targetObject;

public BeforeHandler(Object targetObject) {

this.targetObject = targetObject;

}

@Override

public Object invoke(Object proxy, Method method, Object[] params) throws Throwable {

int param1 = (Integer)params[0];

int param2 = (Integer)params[1];

if( param2 == 0 ) {

//Set the default to 1

param2 = 1;

params[1] = param2;

}

Object result = method.invoke(targetObject ,params);

return result;

}

}

## Test.java

**package** com.ddlab.rnd.dynamic.proxy;

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Proxy;

**public** **class** Test {

**public** **static** Object getProxy(Class interfaceClazz, Object implObj, InvocationHandler handler) {

**return** Proxy.*newProxyInstance*(interfaceClazz.getClassLoader(),

implObj.getClass().getInterfaces(), handler);

}

**public** **static** **void** main(String[] args) {

CalculatorImpl calcImpl = **new** CalculatorImpl();

Calculator proxied = (Calculator) *getProxy* (Calculator.**class**, calcImpl,

**new** BeforeHandler(calcImpl));

**int** result = proxied.calculate(20, 10);

System.***out***.println("FInal Result :::" + result);

}

}

# CG lib Proxy

## Account.java

**package** com.ddlab.rnd.cglib1;

**public** **class** Account {

**private** **int** availBalance = 50000;

**public** **int** withdraw(String acctNo, **int** amount) {

//We will do something here

**return** availBalance - amount;

}

}

# MyAccountInterceptor.java

package com.ddlab.rnd.cglib1;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class MyAccountInterceptor implements MethodInterceptor {

private Object realObj;

public MyAccountInterceptor(Object obj) {

this.realObj = obj;

}

@Override

public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable

{

System.out.println("Before");

Object res = method.invoke(realObj, objects);

System.out.println("After");

return res;

}

}

## Test.java

package com.ddlab.rnd.cglib1;

import net.sf.cglib.proxy.Enhancer;

import com.ddlab.rnd.cglib.MyInterceptor;

public class Test {

public static <T> T createProxy(T obj) {

Enhancer e = new Enhancer();

e.setSuperclass(obj.getClass());

e.setCallback(new MyInterceptor(obj));

T proxifiedObj = (T) e.create();

return proxifiedObj;

}

public static void main(String[] args) {

Account acct = new Account();

Account proxifiedAccount = createProxy(acct);

int amount = proxifiedAccount.withdraw("123456", 1234);

System.out.println("Amount drawn :::"+amount);

}

}

**AOP Example**

**public** **class** Account {

**public** **int** withdrwaMoney(String accountNo , **int** amount ) {

System.***out***.println("Going to withdraw money .....");

//Get total available balance from account

**int** totalBalance = 5000;

**if**( (totalBalance - amount) >= 0 )

**return** amount;

**else** **return** 0;

}

}

**import** java.lang.reflect.Method;

**import** org.springframework.aop.AfterReturningAdvice;

**public** **class** ValidateAfterWithdraw **implements** AfterReturningAdvice {

@Override

**public** **void** afterReturning(Object returnObj, Method method, Object[] methodParams,

Object target) **throws** Throwable {

**int** amount = (**int**) returnObj;

System.***out***.println("Method Name :::"+method.getName());

**if**( method.getName().equals("withdrwaMoney")) {

String actNo = (String) methodParams[0];

//validate acctNo

**boolean** flag = validate(actNo);

**if**( !flag ) amount = 0;

}

}

**private** **boolean** validate(String actNo) {

**return** **false**;

}

}

**import** java.util.Arrays;

**import** org.aopalliance.intercept.MethodInterceptor;

**import** org.aopalliance.intercept.MethodInvocation;

**public** **class** ValidateAroundWithdraw **implements** MethodInterceptor {

@Override

**public** Object invoke(MethodInvocation methodInvocation) **throws** Throwable {

System.***out***.println("Method name : "+ methodInvocation.getMethod().getName());

System.***out***.println("Method arguments : " + Arrays.*toString*(methodInvocation.getArguments()));

Object[] methodParams = methodInvocation.getArguments();

String actNo = (String) methodParams[0];

**int** withdrawAmount = (**int**) methodParams[1];

System.***out***.println("HijackAroundMethod : Before method hijacked!");

**try** {

// proceed to original method call

Object result = methodInvocation.proceed();

System.***out***.println("Expected Result before validation --------->"+result);

//Now validate the account No

**boolean** flag = validate(actNo);

**if**( !flag ) withdrawAmount = 0;

result = withdrawAmount;

**return** result;

} **catch** (IllegalArgumentException e) {

// same with ThrowsAdvice

System.***out***.println("HijackAroundMethod : Throw exception hijacked!");

**throw** e;

}

}

**private** **boolean** validate(String actNo) {

**return** **false**;

}

}

**import** java.lang.reflect.Method;

**import** org.springframework.aop.MethodBeforeAdvice;

**public** **class** ValidateBeforeWithdraw **implements** MethodBeforeAdvice {

@Override

**public** **void** before(Method method, Object[] methodParams, Object target)

**throws** Throwable {

System.***out***.println("Method Name :::"+method.getName());

**if**( method.getName().equals("withdrwaMoney")) {

//validate the account no

String actNo = (String) methodParams[0];

//validate acctNo

**boolean** flag = validate(actNo);

**if**( !flag) **throw** **new** UnsupportedOperationException();

}

}

**private** **boolean** validate(String actNo) {

**return** **false**;

}

}

<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-2.5.xsd"*>

<bean id=*"account"* class=*"com.ddlab.spring.aop.type1.Account"*/>

<bean id=*"validateBeforeWithdraw"* class=*"com.ddlab.spring.aop.type1.ValidateBeforeWithdraw"* />

<bean id=*"validateAfterWithdraw"* class=*"com.ddlab.spring.aop.type1.ValidateAfterWithdraw"* />

<bean id=*"validateAroundWithdraw"* class=*"com.ddlab.spring.aop.type1.ValidateAroundWithdraw"* />

<bean id=*"accountProxy"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*>

<property name=*"target"* ref=*"account"* />

<property name=*"interceptorNames"*>

<list>

<value>validateAroundWithdraw</value>

</list>

</property>

</bean>

</beans>

**import** org.springframework.context.ApplicationContext;

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

**public** **class** TestAOP1 {

**public** **static** **void** main(String[] args) {

ApplicationContext appContext = **new** ClassPathXmlApplicationContext( **new** String[] { "beans.xml" });

Account act = (Account) appContext.getBean("accountProxy");

**int** amount = act.withdrwaMoney("123456", 200);

System.***out***.println("I got the money : "+amount);

}

}

AOP concepts

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html>

* ***Aspect***: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the [schema-based approach](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-schema)) or regular classes annotated with the @Aspectannotation (the [@AspectJ style](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-ataspectj)).
* ***Join point***: a point during the execution of a program, such as the execution of a method or the handling of an exception. **In Spring AOP, a join point *always* represents a method execution**.
* ***Advice*:** action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model **an advice as an *interceptor***, maintaining a chain of interceptors *around* the join point.
* ***Pointcut*:** **a predicate that matches join points**. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
* *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
* *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
* *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
* *Weaving*: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

* *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
* *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
* *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
* *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
* *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

**After (finally) advice**

After (finally) advice runs however a matched method execution exits. It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.After;

*@Aspect*

public class AfterFinallyExample {

*@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")*

public void doReleaseLock() {

*// ...*

}

}

#### Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

* *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
* *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
* *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
* *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
* *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
* *@target* - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
* *@args* - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
* *@within* - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
* *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Examples

*@Pointcut("execution(public \* \*(..))")*

**private** **void** anyPublicOperation() {}

*@Pointcut("within(com.xyz.someapp.trading..\*)")*

**private** **void** inTrading() {}

*@Pointcut("anyPublicOperation() && inTrading()")*

**private** **void** tradingOperation() {}

**How to use “execution”**

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

**execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)**

**throws-pattern?)**

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(**public** \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

**this**(com.xyz.service.AccountService)

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classifiedannotation:

@args(com.xyz.security.Classified)

**Due to the proxy-based nature of Spring’s AOP framework, protected methods are by definition *not* intercepted, neither for JDK proxies (where this isn’t applicable) nor for CGLIB proxies (where this is technically possible but not recommendable for AOP purposes). As a consequence, any given pointcut will be matched against *public methods only*!**

**If your interception needs include protected/private methods or even constructors, consider the use of Spring-driven**[**native AspectJ weaving**](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-aj-ltw)**instead of Spring’s proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision.**

Application level Instrumentation logging example

**LoggerAspect.java**

**package** com.ddlab.rnd.aop;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Pointcut;

@Aspect

**public** **class** LoggerAspect {

//Hook all the public methods

@Pointcut("execution(\* \*(..))")

**public** **void** allMethodsPointcuts(){

}

@Pointcut("@within(com.ddlab.rnd.aop.AuditLogger)")

// @Pointcut("@target(com.ddlab.rnd.aop.AuditLogger)")//It is also ok.

**public** **void** allAnnotatedClasses() {

}

//~~ Actual Implementation

@Around("allAnnotatedClasses()")

**public** Object logAnnotatedClasses( ProceedingJoinPoint pjp ) {

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Object result = **null**;

System.***out***.println("Target Object :::"+pjp.getTarget());

System.***out***.println("Signature :::"+pjp.getSignature());

**for**(Object obj : pjp.getArgs() )

System.***out***.println("Parameter Values : "+obj);

**try** {

result = pjp.proceed();

} **catch** (Throwable e) {

e.printStackTrace();

}

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

**return** result;

}

@Around("allMethodsPointcuts()")

**public** Object logMethods( ProceedingJoinPoint pjp ) {

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Object result = **null**;

System.***out***.println("Target Object :::"+pjp.getTarget());

System.***out***.println("Signature :::"+pjp.getSignature());

**for**(Object obj : pjp.getArgs() )

System.***out***.println("Parameter Values : "+obj);

**try** {

result = pjp.proceed();

} **catch** (Throwable e) {

e.printStackTrace();

}

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

**return** result;

}

}

**AuditLogger.java**

**package** com.ddlab.rnd.aop;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** AuditLogger {

}

**TargetBean1.java**

**package** com.ddlab.rnd.aop;

**public** **class** TargetBean1 {

**public** String achieveTarget( String s1 , String s2 , String s3) {

**return** getValue();

}

**private** String getValue() {

**return** "successfull";

}

}

**TargetBean2.java**

**package** com.ddlab.rnd.aop;

@AuditLogger

**public** **class** TargetBean2 {

**public** String executeTarget( String s1 ) {

**return** "some value";

}

}

**TestAOP.java**

**package** com.ddlab.rnd.aop;

**import** org.springframework.context.ApplicationContext;

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

**public** **class** TestAOP {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

TargetBean1 bean1 = (TargetBean1) context.getBean("targetBean1");

String result = bean1.achieveTarget("asdf", **null**, "pqrs");

System.***out***.println("Result ::: "+result);

TargetBean2 bean2 = (TargetBean2) context.getBean("targetBean2");

bean2.executeTarget("Hi");

}

}

**Spring IOC Configuration File(beans.xml)**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>

<aop:aspectj-autoproxy/>

<bean id=*"targetBean1"* class=*"com.ddlab.rnd.aop.TargetBean1"*/>

<bean id=*"targetBean2"* class=*"com.ddlab.rnd.aop.TargetBean2"*/>

<bean id=*"loggerBean"* class=*"com.ddlab.rnd.aop.LoggerAspect"*/>

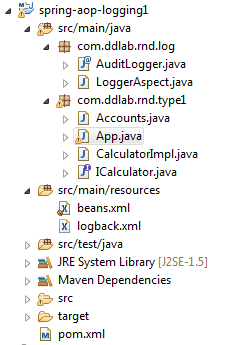
</beans>

Logging in application using Spring AOP

You can log all information from the methods about the method parameters, return type, result and others without explicitly mentioning log.debug statements in the method. You can use Spring AspectJ AOP concept to track all information. It also can be used as instrumentation logging also. You can create your own annotation to hook all the classes where you want to print the log statements.

The code project is given below.

# Project Structure



# Spring XML Configuration(beans.xml)

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd*

*http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd"*

xmlns:c=*"http://www.springframework.org/schema/c"* xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:p=*"http://www.springframework.org/schema/p"*>

<context:component-scan base-package=*"com.ddlab.rnd.spring.aop"* />

<aop:aspectj-autoproxy />

<bean class=*"com.ddlab.rnd.log.LoggerAspect"*></bean>

<bean id=*"calculator"* class=*"com.ddlab.rnd.type1.CalculatorImpl"*/>

<bean id=*"accounts"* class=*"com.ddlab.rnd.type1.Accounts"*/>

</beans>

# SLF4J Configuration(logback.xml)

<configuration>

<appender name=*"STDOUT"* class=*"ch.qos.logback.core.ConsoleAppender"*>

<!-- encoders are assigned the type

ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger - %msg%n</pattern>

</encoder>

</appender>

<root level=*"debug"*>

<appender-ref ref=*"STDOUT"* />

</root>

<logger name=*"org.springframework"* level=*"OFF"*/>

<logger name=*"org.apache"* level=*"OFF"*/>

</configuration>

# Java Code

# LoggerAspect.java

**package** com.ddlab.rnd.log;

**import** java.util.Date;

**import** org.apache.commons.beanutils.BeanUtils;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Pointcut;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

@Aspect

**public** **class** LoggerAspect {

**protected** **static** Logger *log* = LoggerFactory.*getLogger*(LoggerAspect.**class**);

@Pointcut("within(@org.springframework.stereotype.Controller \*)")

**public** **void** allControllerBean() {

}

@Pointcut("within(@org.springframework.stereotype.Service \*)")

**public** **void** allServiceBean() {

}

@Pointcut("within(@org.springframework.stereotype.Repository \*)")

**public** **void** allRepositoryBean() {

}

@Pointcut("within(@com.ddlab.rnd.log.AuditLogger \*)")

**public** **void** otherBean() {

}

@Pointcut("execution(\* \*(..))")

**public** **void** methodPointcut() {

}

@Around("otherBean() && methodPointcut()")

**public** Object aroundCheckMethod(ProceedingJoinPoint joinPoint) **throws** Throwable {

*log*.debug("======================AOP Logging=========================");

*log*.debug("Class Name : "+joinPoint.getTarget());

*log*.debug("Method Signature : {}",joinPoint.getSignature());

Object[] methodArguments = joinPoint.getArgs();

**for**( Object obj : methodArguments )

*log*.debug("Method argument : {}",obj);

Date start = **new** Date();

Object result = joinPoint.proceed();

*log*.debug("Method return type : {}",BeanUtils.*describe*(result));

*log*.debug("Method output if exists : "+result);

Date end = **new** Date();

*log*.debug(">>>>>>>> end of {} took {} millisec",

joinPoint.getSignature(), end.getTime() - start.getTime());

*log*.debug("======================AOP Logging=========================");

**return** result;

}

//For others

@Around("allControllerBean() && methodPointcut()")

**public** Object aroundControllerMethod(ProceedingJoinPoint joinPoint)

**throws** Throwable {

*log*.debug(">>>>>>>> invoking {}", joinPoint.getSignature());

Date start = **new** Date();

Object result = joinPoint.proceed();

Date end = **new** Date();

*log*.debug(">>>>>>>> return of {} with {}", joinPoint.getSignature(),

BeanUtils.*describe*(result));

*log*.debug(">>>>>>>> end of {} take {} millisec",

joinPoint.getSignature(), end.getTime() - start.getTime());

**return** result;

}

@Around("allServiceBean() && methodPointcut()")

**public** Object aroundServiceMethod(ProceedingJoinPoint joinPoint)

**throws** Throwable {

*log*.debug(">>>>>>>> invoking {}", joinPoint.getSignature());

Date start = **new** Date();

Object result = joinPoint.proceed();

Date end = **new** Date();

*log*.debug(">>>>>>>> return of {} with {}", joinPoint.getSignature(),

BeanUtils.*describe*(result));

*log*.debug(">>>>>>> method arguments {} , {}", joinPoint.getArgs());

*log*.debug(">>>>>>>> end of {} take {} millisec",

joinPoint.getSignature(), end.getTime() - start.getTime());

**return** result;

}

@Around("allRepositoryBean() && methodPointcut()")

**public** Object aroundRepositoryMethod(ProceedingJoinPoint joinPoint)

**throws** Throwable {

*log*.debug(">>>>>>>> invoking {}", joinPoint.getSignature());

Date start = **new** Date();

Object result = joinPoint.proceed();

Date end = **new** Date();

*log*.debug(">>>>>>>> return of {} with {}", joinPoint.getSignature(),

BeanUtils.*describe*(result));

*log*.debug(">>>>>>>> end of {} take {} millisec",

joinPoint.getSignature(), end.getTime() - start.getTime());

**return** result;

}

}

# AuditLogger.java(Annotation)

**package** com.ddlab.rnd.log;

**public** **@interface** AuditLogger {

}

# Core Business Logic

# ICalculator.java

**package** com.ddlab.rnd.type1;

**public** **interface** ICalculator {

**int** add( **int** a , **int** b);

}

# CalculatorImpl.java

**package** com.ddlab.rnd.type1;

**import** com.ddlab.rnd.log.AuditLogger;

@AuditLogger

**public** **class** CalculatorImpl **implements** ICalculator {

**public** **int** add(**int** a, **int** b) {

**return** a+b;

}

}

# Accounts.java

**package** com.ddlab.rnd.type1;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** com.ddlab.rnd.log.AuditLogger;

@AuditLogger

**public** **class** Accounts {

@Autowired

**private** ICalculator calculator;

**public** **void** setCalculator(ICalculator calculator) {

**this**.calculator = calculator;

}

**public** **int** getCalculation( **int** a , **int** b ) {

**return** calculator.add(a, b);

}

}

# App.java(Test)

**package** com.ddlab.rnd.type1;

**import** org.springframework.context.ApplicationContext;

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

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

Accounts accounts = (Accounts) context.getBean("accounts");

**int** result = accounts.getCalculation(10, 15);

System.***out***.println("Result :::" + result);

}

}

# Output

22:46:35.280 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - ======================AOP Logging=========================

22:46:35.282 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Class Name : com.ddlab.rnd.type1.Accounts@47bd5719

22:46:35.283 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method Signature : int com.ddlab.rnd.type1.Accounts.getCalculation(int,int)

22:46:35.283 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method argument : 10

22:46:35.283 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method argument : 15

22:46:35.305 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - ======================AOP Logging=========================

22:46:35.305 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Class Name : com.ddlab.rnd.type1.CalculatorImpl@956ff4d

22:46:35.305 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method Signature : int com.ddlab.rnd.type1.ICalculator.add(int,int)

22:46:35.305 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method argument : 10

22:46:35.305 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method argument : 15

22:46:35.335 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method return type : {class=class java.lang.Integer}

22:46:35.335 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method output if exists : 25

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - >>>>>>>> end of int com.ddlab.rnd.type1.ICalculator.add(int,int) took 31 millisec

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - ======================AOP Logging=========================

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method return type : {class=class java.lang.Integer}

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - Method output if exists : 25

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - >>>>>>>> end of int com.ddlab.rnd.type1.Accounts.getCalculation(int,int) took 53 millisec

22:46:35.336 [main] DEBUG com.ddlab.rnd.log.LoggerAspect - ======================AOP Logging=========================

Result :::25

# Maven Configuration(pom.xml)

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>spring-aop-logging1</groupId>

<artifactId>spring-aop-logging1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>spring-aop-logging1</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>3.2.1.RELEASE</spring.version>

<spring.security.version>3.1.3.RELEASE</spring.security.version>

<spring.security.oauth.version>1.0.2.RELEASE</spring.security.oauth.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-web</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-core</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-taglibs</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security.oauth</groupId>

<artifactId>spring-security-oauth2</artifactId>

<version>${spring.security.oauth.version}</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.6.11</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.6.11</version>

</dependency>

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>2.2.2</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>1.6.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jul-to-slf4j</artifactId>

<version>1.6.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.6.6</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.0.9</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<!-- Hibernate Validator -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.2.0.Final</version>

</dependency>

<dependency>

<groupId>commons-beanutils</groupId>

<artifactId>commons-beanutils</artifactId>

<version>1.8.3</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.21</version>

</dependency>

</dependencies>

</project>

**HOW AOP works internally in Spring**

http://stackoverflow.com/questions/8224465/spring-use-of-proxies-in-spring-aop

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxies for your target objects.

According to Spring documentation, in case your target implements at least one interface, a JDK dynamic proxy will be used. However if your target object does not implement any interfaces then a CGLIB proxy will be created.

This is how you can force creation of the CGLIB proxies (set proxy-target-class="**true**"):

<aop:config proxy-target-class="true">

<!-- other beans defined here... -->

</aop:config>

When using AspectJ and its autopoxy support you can also force CGLIB proxies. This is where the<aop:aspectj-autoproxy> is used and also here the "proxy-target-class" must be set to **true**:

<aop:aspectj-autoproxy proxy-target-class="true"/>

Spring AOP makes extensive use of proxies as a mechanism to implement cross-cutting concerns (a.k.a aspects) in a non-intrusive way, the idea basically is use the proxies as wrappers that enrich the original behaviour, i.e. add transactional capabilities.

To achieve this there are two options, depending of whether the original object implements an interface or not.

In the first case (the original object implements at least one interface) the dynamic proxy capabilities of the reflection API are used to create a proxy object that **IMPLEMENTS** the same interfaces that the original object and therefore the proxy can be used instead.

In the second case (the original object does **NOT** implement any interface), so a more elaborated trick must be used, and this is when CGLIB appears. According to the project page "CGLIB is used to extend Java classes and implements interfaces at runtime". So in this case the trick consists on create a proxy that **EXTENDS** the original object and therefore can be used instead.

<https://today.java.net/pub/a/today/2005/11/01/implement-proxy-based-aop.html>

Aspect-oriented programming (AOP) is well-suited to managing application crosscutting concerns, such as logging, security, and  
transaction management. AOP provides a complement to object-oriented programming (OOP), which is still the most common and  
powerful methodology to address core business concerns. AOP can reduce code scattering, tangling, and duplication in applications.  
Based on their implementation approaches, AOP frameworks can be classified into two categories:

1. **Class-weaving-based**, such as ["http://eclipse.org/aspectj/">AspectJ](https://today.java.net/pub/a/today/2005/11/01/%3Cbr) and ["http://www.jboss.org/products/aop">JBoss AOP](https://today.java.net/pub/a/today/2005/11/01/%3Cbr). Core and  
   crosscutting concerns are implemented independently. Class weaving is the process of integrating the concern implementations to form  
   the final system. Weaving can be performed at compile, load, and run time. Both AspectJ and JBoss AOP are very powerful AOP  
   implementations. They provide field interception, caller side interception, and constructor interception.
2. **Proxy-based**, such as ["http://www.springframework.org/">Spring AOP](https://today.java.net/pub/a/today/2005/11/01/%3Cbr), ["http://nanning.codehaus.org/">Nanning](https://today.java.net/pub/a/today/2005/11/01/%3Cbr), and["https://dynaop.dev.java.net/">dynaop](https://today.java.net/pub/a/today/2005/11/01/%3Cbr). With proxies, method invocations on an object can be intercepted to inject custom code.  
   The aforementioned AOP frameworks use ["http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html">JDK  
   dynamic proxy](https://today.java.net/pub/a/today/2005/11/01/%3Cbr), ["http://cglib.sourceforge.net/">CGLIB](https://today.java.net/pub/a/today/2005/11/01/%3Cbr) proxy, or both. Unlike the class-weaving-based ones, proxy-based AOP frameworks are simpler and often focus on method interception. Most of the time, Java developers use method interception only. Some proxy-based AOP implementations, such as Spring AOP, provide close integration with AspectJ to take advantage of its capabilities.

JDK dynamic proxy has been available since JDK 1.3. The proxy class, which implements a list of interfaces specified at runtime,  
is dynamically created by the JVM. Method invocations on the proxy class are delegated to the underlying proxied object. JDK dynamic  
proxy is simple to use, but, like all reflective code, it is somewhat slower. For most situations, the overhead is not critical.  
Another limitation is that it can only implement interfaces.

What if you want to proxy legacy classes that do not have interfaces? You can use CGLIB. CGLIB is a powerful, high-performance code generation library. Under the cover, it uses [ASM](http://asm.objectweb.org/), a small but fast bytecode manipulation framework, to transform existing byte code to  
generate new classes. CGLIB is faster than the JDK dynamic proxy approach. Essentially, it dynamically generates a subclass to  
override the non-final methods of the proxied class and wires up hooks that call back to the user-defined interceptors.

To help you understand and demystify AOP, this article shows you how to create a simple AOP framework using both JDK dynamic proxy  
and CGLIB. This framework supports declarative transaction management. This article uses Java 5 features, including annotations and generics. Since JDK dynamic proxy is simpler, this article starts with dynamic proxy.

## Creating a Proxy Factory

A proxy factory is the central place to create proxies for the requested target classes. Clients of proxies do not know how the  
proxies are created.

public interface DynamicProxyFactory{

    &lt;T&gt; T createProxy(Class&lt;T&gt; clazz,

        T target,

        AOPInterceptor interceptor);

}

To create a dynamic proxy, you need a list of proxy interfaces and a target object. There is a set of rules about the proxy  
interfaces. You can look at the

["http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html">  
java.lang.reflect.Proxy](https://today.java.net/pub/a/today/2005/11/01/%3Cbr)

documentation for details. For simplicity, this article uses a single interface only. You can ignore the interceptor argument for now; it will be discussed in the next section.

public &lt;T&gt; T createProxy(Class&lt;T&gt; clazz,

    T target, AOPInterceptor interceptor) {

    InvocationHandler handler =

        new DynamicProxyInvocationHandler(target,

            interceptor);

    return (T)Proxy.newProxyInstance(

        Thread.currentThread().getContextClassLoader(),

        new Class&lt;?&gt;[] {clazz},

        handler);

}

The implementation of the proxy factory is simple. First, it creates an instance of InvocationHandler, which is one of the two key dynamic proxy APIs. Then, it uses the static method Proxy.newProxyInstance to create a proxy that implements the interface passed in as its second parameter. Note that the second argument of the newProxyInstance method is an array of Class<?> instead of Class<T>.  
Arrays cannot be created if the element type is generic, but an unbounded wildcard can be used. All method invocations on the generated proxy class are forwarded to InvocationHandler's single method:

public Object invoke(Object proxy,

        Method method, Object[] args)

Let's see how this method is implemented in  
*DynamicProxyInvocationHandler.java*:

public class DynamicProxyInvocationHandler

        implements InvocationHandler {

    private Object target;

    private AOPInterceptor interceptor;

    public DynamicProxyInvocationHandler(Object target,

        AOPInterceptor interceptor)  {

        this.target = target;

        this.interceptor = interceptor;

    }

    public Object invoke(Object proxy, Method method,

        Object[] args) throws Throwable{

        try {

            interceptor.before(method, args);

            Object returnValue = method.invoke(target, args);

            interceptor.after(method, args);

            return returnValue;

        } catch(Throwable t) {

            interceptor.afterThrowing(method, args, t);

            throw t;

        } finally {

            interceptor.afterFinally(method, args);

        }

    }

}

If you ignore the interceptor-related code, the implementation of the invoke method is straightforward. It uses reflective invocation  
on the Method object to delegate to the target object.

## Adding an Interceptor

As discussed in the preceding section, all method invocations on the proxy class are forwarded to the invoke method of InvocationHandler. The invoke method delegates calls to the target object. Since all method calls have to go through the single invoke method, you can apply the decorator pattern on that method, or even immediately return without further delegating to the target object. If you decorate that method before delegating to the target object, you are essentially applying AOP before advice. If you add custom code after delegating to the target object, you are essentially applying AOP after advice. If, instead of delegating to the target object, you route method calls to a different path, you are applying "around" advice. Now it should be easy for you to figure out what  
afterThrowing and afterReturn advices mean. For details on each kind of advice, you can refer to the book [AspectJ in  
Action](http://www.manning.com/books/laddad).

In this article, some method advices are grouped into the AOPInterceptor. As its name implies, it intercepts method invocations on the target object through decorating the invoke method of InvocationHandler, as shown in the DynamicProxyInvocationHandler class. The method advices should be decoupled in the real world. Advices for finally blocks are not common, but are added here for demonstration purposes.

public interface AOPInterceptor {

    void before(Method method, Object[] args);

    void after(Method method, Object[] args);

    void afterThrowing(Method method, Object[] args, Throwable throwable);

    void afterFinally(Method method, Object[] args);

}

## AOP Framework in Action

Typically, you do not want to intercept all of the method calls. That is, advices are applied only to the methods or classes you are  
interested in. You can set the classes and methods through regular expressions in XML files, annotations, or other mechanisms. At  
run time, your framework should be able to decide whether or not to apply any advices by matching the current class and method with  
those specified in your configuration files or annotations. Even the runtime argument values for a method can be used to determine  
which advice should be applied. The TransactionAnnotation is a simple annotation, as shown below:

[prettify]@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface TransactionAnnotation {

    String value();

}

[/prettify]

The @Retention meta-annotation is set to RetentionPolicy.RUNTIME so that it can be accessed via reflection. You can attach annotations to methods only since the @Target meta annotation is set to  
ElementType.METHOD.

Assume you have a business service, called  
PersistenceService. It requires a new transaction in  
its save method, but transactions are not supported in  
its load method.

public interface PersistenceService {

    @TransactionAnnotation("REQUIRES\_NEW")

    void save(long id, String data);

    @TransactionAnnotation("NOT\_SUPPORTED")

    String load(long id);

}

Now you need a transaction. Assume you have a transaction API like this:

public interface Transaction{

    void open();

    void rollBack();

    void commit();

    void closeIfStillOpen();

}

The transaction starts by calling the open method and must be closed after use. Here is the transaction interceptor  
that performs declarative transaction management for PersistenceService:

public class TransactionInterceptor

        implements AOPInterceptor {

    private Transaction transaction;

    public void before(Method method, Object[] args) {

        if (isRequiresNew(method)) {

            transaction = new TransactionAdapter();

            transaction.open();

        }

    }

    public void after(Method method, Object[] args) {

        if (transaction != null) {

            transaction.commit();

        }

    }

    public void afterThrowing(Method method,

        Object[] args, Throwable t) {

        if (transaction != null) {

            transaction.rollBack();

        }

    }

    public void afterFinally(Method method, Object[] args) {

        if (transaction != null) {

            transaction.closeIfStillOpen();

        transaction = null;

        }

    }

    protected boolean isRequiresNew(Method method) {

        TransactionAnnotation transactionAnnotation =

            method.getAnnotation(TransactionAnnotation.class);

        if (transactionAnnotation != null) {

            if ("REQUIRES\_NEW".equals(

                transactionAnnotation.value())){

            return true;

        }

        }

        return false;

    }

}

Now you can plug in the transaction interceptor into a proxy when the proxy is created.

DynamicProxyFactory proxyFactory = new DynamicProxyFactoryImpl();

AOPInterceptor interceptor = new TransactionInterceptor();

PersistenceService proxy =

    proxyFactory.createProxy(PersistenceService.class,

        new PersistenceServiceImpl(),

        interceptor);

proxy.save(1, "Jason Zhicheng Li");

String data = proxy.load(1);

You can run the manual test from the attached source code to see the results. As annotated in the PersistenceService interface, the save method is executed in a new transaction context, but there is no transaction for the load method. Without much coding, you can externalize the proxy creation through dependency injection. If you have experience in dependency injection frameworks, like ["http://www.springframework.org/">Spring](https://today.java.net/pub/a/today/2005/11/01/%3Cbr), it should be familiar to you. All you need to do is to configure interface type, target,  
and interceptor in your proxy factory. You can even add a layer of abstraction for the target by passing a target holder instance into  
the proxy factory. The target holder has a reference to the real target and it can be instantiated without a real target. Then you  
can implement advanced features such as hot swapping or pooling of real targets and virtual proxies.

## Alternative Implementation Using CGLIB

Similar to InvocationHandler and Proxy in dynamic proxy, there are two key APIs in CGLIB proxy, MethodInterceptor and Enhancer. The  
MethodInterceptor is the general callback interface used by Enhancer, which dynamically generates subclasses to override the non-final methods of the superclass. MethodInterceptor is responsible for intercepting all method calls in the generated proxy. You can invoke custom code before and after the invocation of the super methods, and even skip invocation of the super methods. Typically, a single callback is used per enhanced class, but you can use CallbackFilter to control which callback to use for a method.

Let's first create a CGLIB MethodInterceptor.

public class CGLIBMethodInterceptor

        implements MethodInterceptor {

    private AOPInterceptor interceptor;

    public CGLIBMethodInterceptor(AOPInterceptor interceptor) {

        this.interceptor = interceptor;

    }

    public Object intercept(Object object, Method method,

        Object[] args, MethodProxy methodProxy )

            throws Throwable {

        try {

            interceptor.before(method, args);

            Object returnValue =

                methodProxy.invokeSuper(object, args);

            interceptor.after(method, args);

            return returnValue;

        } catch(Throwable t) {

            interceptor.afterThrowing(method, args, t);

            throw t;

       } finally {

            interceptor.afterFinally(method, args);

       }

}

The implementation is very similar to DynamicProxyInvocationHandler in dynamic proxy, but note that there is no target object and the type T is the concrete class type, not the interface type as in DynamicProxyFactory. The real method is invoked by using MethodProxy, which is faster, instead of the Method object. Now let's create the proxy factory:

public class CGLIBProxyFactoryImpl

        implements CGLIBProxyFactory {

    public &lt;T&gt; T createProxy(Class&lt;T&gt; clazz,

            AOPInterceptor interceptor) {

        MethodInterceptor methodInterceptor =

            new CGLIBMethodInterceptor(interceptor);

        Enhancer enhancer = new Enhancer();

        enhancer.setSuperclass(clazz);

        enhancer.setCallback(methodInterceptor);

        return (T)enhancer.create();

    }

}

After you set the superclass type and method interceptor, you simply call the create() method on the Enhancer object to create a proxy. Optionally, you can configure CallbackFilter to map a method to a callback by calling the setCallbackFilter(CallbackFilter) method.  
In addition, you can specify the proxy class to implement a set of interfaces. In this CGLIB implementation, since no interface is  
specified, the transaction attributes must be declared in the PersistenceService implementation instead of the interface. Similarly, you can implement interceptors to address logging, validation, auditing, caching, and security, which are orthogonal to core business concerns. As shown above, both dynamic proxy and CGLIB implementation are simple to implement, but you must be aware that important issues such as performance, exception handling, and threading are not covered here.

## Conclusion

The AOP implementation in this article is simplified for clarity, but it shows you the essentials of proxy-based AOP frameworks. AOP decouples crosscutting concerns, such as the transaction management demonstrated in this article, from application core concerns. With aspect-oriented design and programming, you can significantly simplify your design and implementation. In some cases, however, third-party AOP frameworks cannot be used due to non-technical reasons, such as corporate policies and license issues. As shown in this article, you can  
implement your own AOP framework that is tailored to meet your needs. JDK dynamic-proxy-based implementation is simpler, since it  
uses standard Java. That means there are no third-party libraries or build-time bytecode instrumentation. Alternatively, you can choose CGLIB to proxy legacy classes and have better performance, but you need to introduce multiple third-party libraries into your system. At that moment, you should ask yourself if you need to pick an available AOP framework, which is often more complete and sophisticated than your roll-your-own AOP implementation.

<http://markbramnik.blogspot.in/2010/04/cglib-introduction.html>

### CGLIB introduction

Today I would like to briefly discuss the bytecode generation framework, CGLIB.  
  
There are a lot of these frameworks, each one works at the different level of abstraction.  
Recently I was looking for a high-level framework that would let me to dynamically change my classes providing its proxy and substituting the functionality of some methods.  
  
While the most obvious jdk proxies can do the job (java.lang.reflect.Proxy), I've figured out, that when I don't have both an interface and implementation of my to-be-proxified class, it just doesn't work. So I've found another solution, a library called [CGLIB](http://cglib.sourceforge.net/)  
  
The only significant drawback for me was a **lack of comprehensive documentation**, in fact I've found only one decent tutorial [here](http://jnb.ociweb.com/jnb/jnbNov2005.html) (It could be great if someone could point me on more tutorials about this tool).  
Anyway, I think that a beginner's level introduction can't harm so I fill the gap and share the experience :)  
  
So CGLIB is a bytecode generation library, that relies on low-level  
[ASM](http://forge.ow2.org/projects/asm) framework.  
So in order to create a working example we'll need to open a regular java project and add two jars as a dependency (the latest versions available at the moment):  
  
**- cglib-2.2.jar  
- asm-all-3.2.jar**  
We will 'proxify' the mock Algorithm class which is supposed to implement some long-running algorithm. We would like to measure its execution time.  
  
So we create our algorithm like this:

|  |  |  |
| --- | --- | --- |
| 01 | public class Algorithm { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public void runAlgorithm() { |
| 04 | System.out.println("running the algorithm"); | |

|  |  |
| --- | --- |
| 05 | try { |
| 06 | // do something  here - | |

|  |  |  |
| --- | --- | --- |
| 07 | // simulate some real time-consuming operation here | |
| 08 | Thread.sleep(500); |

|  |  |  |
| --- | --- | --- |
| 09 | } catch (InterruptedException e) { | |
| 10 | e.printStackTrace(); |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 | } |

|  |  |
| --- | --- |
| 13 | } |

Now the most interesting part of the program:  
We'll create a class that adds the 'measurements'. This class will be used by CGLIB to proxify our algorithm, so it should implement *net.sf.cglib.proxy.MethodInterceptor*  
  
The class looks like this:

|  |  |  |
| --- | --- | --- |
| 01 | import net.sf.cglib.proxy.MethodInterceptor; | |
| 02 | import net.sf.cglib.proxy.MethodProxy; |

|  |  |
| --- | --- |
| 03 |  |
| 04 | import java.lang.reflect.Method; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | public class MyInterceptor implements MethodInterceptor { | |

|  |  |  |
| --- | --- | --- |
| 07 | // the real object | |
| 08 | private Object realObj; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | // constructor - the supplied parameter is an | |

|  |  |  |
| --- | --- | --- |
| 11 | // object whose proxy we would like to create | |
| 12 | public MyInterceptor(Object obj) { |

|  |  |  |
| --- | --- | --- |
| 13 | this.realObj = obj; | |
| 14 | } |

|  |  |
| --- | --- |
| 15 |  |
| 16 | // this method will be called each time | |

|  |  |  |
| --- | --- | --- |
| 17 | // when the object proxy calls any of its methods | |
| 18 | public Object intercept(Object o, |

|  |  |
| --- | --- |
| 19 | Method method, |
| 20 | Object[] objects, | |

|  |  |  |
| --- | --- | --- |
| 21 | MethodProxy methodProxy) throws Throwable { | |
| 22 | // just print that we're about to execute the method |

|  |  |  |
| --- | --- | --- |
| 23 | System.out.println("Before"); | |
| 24 | // measure the current time |

|  |  |
| --- | --- |
| 25 | long time1 = System.currentTimeMillis(); |
| 26 | // invoke the method on the real object with the given params | |

|  |  |  |
| --- | --- | --- |
| 27 | Object res = method.invoke(realObj, objects); | |
| 28 | // print that the method is finished |

|  |  |
| --- | --- |
| 29 | System.out.println("After"); |
| 30 | // print how long it took to execute the method on the proxified object | |

|  |  |  |
| --- | --- | --- |
| 31 | System.out.println("Took: " + (System.currentTimeMillis() - time1) + " ms"); | |
| 32 | // return the result |

|  |  |  |
| --- | --- | --- |
| 33 | return res; | |
| 34 | } |

|  |  |
| --- | --- |
| 35 | } |

The last class is the main class. Here we will actually create the proxy so here we'll see some CGLIB related code:

|  |  |  |
| --- | --- | --- |
| 01 | import net.sf.cglib.proxy.Enhancer; | |
| 02 | public class Main { |

|  |  |  |
| --- | --- | --- |
| 03 | public static void main(String[] args) { | |
| 04 | // 1. create the 'real' object |

|  |  |  |
| --- | --- | --- |
| 05 | Algorithm alg = new Algorithm(); | |
| 06 | // 2. create the proxy |

|  |  |
| --- | --- |
| 07 | Algorithm  proxifiedAlgorithm = createProxy(alg); |
| 08 | // 3. execute the proxy - as we see it has the same API as the real object | |

|  |  |  |
| --- | --- | --- |
| 09 | proxifiedAlgorithm.runAlgorithm(); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 | // given the obj, creates its proxy |
| 12 | // the method is generified - just to avoid downcasting... | |

|  |  |
| --- | --- |
| 13 | public static <T> T createProxy(T obj) { |
| 14 | // this is the main cglib api entry-point | |

|  |  |  |
| --- | --- | --- |
| 15 | // this object will 'enhance' (in terms of CGLIB) with new capabilities | |
| 16 | // one can treat this class as a 'Builder' for the dynamic proxy |

|  |  |  |
| --- | --- | --- |
| 17 | Enhancer e = new Enhancer(); | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | // the class will extend from the real class | |
| 20 | e.setSuperclass(obj.getClass()); |

|  |  |
| --- | --- |
| 21 | // we have to declare the interceptor  - the class whose 'intercept' |
| 22 | // will be called when any method of the proxified object is called. |

|  |  |
| --- | --- |
| 23 | e.setCallback(new MyInterceptor(obj)); |
| 24 | // now the enhancer is configured and we'll create the proxified object | |

|  |  |
| --- | --- |
| 25 | T proxifiedObj = (T) e.create(); |
| 26 | // the object is ready to be used - return it | |

|  |  |  |
| --- | --- | --- |
| 27 | return proxifiedObj; | |
| 28 | } |

|  |  |
| --- | --- |
| 29 | } |

The output of this program is predictable :)  
  
*Before*  
*running the algorithm*  
*After*  
*Took: 500 ms*

[AspectJ](http://en.wikipedia.org/wiki/AspectJ) Pointcut Expressions in [Spring](http://www.springsource.org/) Applications

In [Aspect Oriented Programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming), a pointcut is a set of joinpoints. A joinpoint is a point in program execution where you can add additional behavior. Spring applications only support method based joinpoints. So, you can use AspectJ pointcut expressions to define method pointcuts.

**Method Signature Patterns**

The most typical pointcut expressions are used to match a number of methods by their signatures. A common method based pointcut expression is something like

**expression(<method scope> <return type> <fully qualified class name>.\*(parametes))**

1. method scope: Advice will be applied to all the methods having this scope. For e.g., public, private, etc. Please note that Spring AOP only supports advising public methods.
2. return type: Advice will be applied to all the methods having this return type.
3. fully qualified class name: Advice will be applied to all the methods of this type. If the class and advice are in the same package then package name is not required
4. parameters: You can also filter the method names based on the types. Two dots(..) means any number and type of parameters.

**Examples**

* execution(\* com.aspects.pointcut.DemoClass.\*(..)) : This advice will be applied to all the methods of DemoClass.
* execution(\* DemoClass.\*(..)): You can omit the package if the DemoClass and the advice is in the same package.
* execution(public \* DemoClass.\*(..)): This advice will be applied to the public methods of DemoClass.
* execution(public int DemoClass.\*(..)): This advice will be applied to the public methods of DemoClass and returning an int.
* execution(public int DemoClass.\*(int, ..)): This advice will be applied to the public methods of DemoClass and returning an int and having first parameter as int.
* execution(public int DemoClass.\*(int, int)): This advice will be applied to the public methods of DemoClass and returning an int and having both parameters as int.

### **Type Signature Patterns**

These pointcut expressions are applied to all joinpoint of certain types. A common type signature patterns looks like

**within(type name)**

Here type name is either the package name or the class name.

**Examples**

* within(com.aspects.blog.package.\*) : This will match all the methods in all classes of com.aspects.blog.package.
* within(com.aspects.blog.package..\*) : This will match all the methods in all classes of com.aspects.blog.package and its sub packages. The only difference is the extra dot(.) after package.
* within(com.aspects.blog.package.DemoClass) : This will match all the methods in the DemoClass.
* within(DemoClass) : Again, if the target class is located in the same package as this aspect, the package name can be omitted.
* within(DemoInterface+) : This will match all the methods which are in classes which implement DemoInterface.

<http://docs.spring.io/spring/docs/2.0.x/reference/aop.html#aop-pointcuts>

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

**execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)**

**throws-pattern?)**

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use \* as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the \* wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more). The pattern (\*) matches a method taking one parameter of any type, (\*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the [Language Semantics](http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(public \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

*'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.*

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

*'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

*'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.*

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

*'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

*'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

*'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

*'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.*

<http://insufficientinformation.blogspot.in/2007/12/spring-dynamic-proxies-vs-cglib-proxies.html>

[Spring's AOP](http://static.springframework.org/spring/docs/2.5.x/reference/aop.html) is proxy-based. Spring provides [two different options](http://static.springframework.org/spring/docs/2.5.x/reference/aop.html#aop-proxying) to create the proxies. One is based on JDK dynamic proxies and works with interfaces, the other one utilizes CGLib and is based on classes. (That's why the property is called proxyTargetClass respectively proxy-target-class.) For the moment I just want to provide a quick summary on the pros and cons of both options:  
  
**JDK dynamic proxies:**

* The class ***has*** to implement interfaces. Otherwise you will get ClassCastExceptions saying that $Proxy0 can not be casted to the particular class.
* Eventually dynamic proxies force you to program to interfaces since you can not cast the proxy to the class - a feature I ***really*** like about them.

**CGLib proxies:**

* The proxies are created by sub-classing the actual class. This means wherever an instance of the class is used it is also possible to use the CGLib proxy.
* The class needs to provide a default constructor, i.e. without any arguments. Otherwise you'll get an IllegalArgumentException: "Superclass has no null constructors but no arguments were given." This makes constructor injection impossible.

The proxying does not work with final methods since the proxy sub class can not override the class' implementation.

The CGLib proxy is final, so proxying a proxy does not work. You will get an IllegalArgumentException saying "Cannot subclass final class $Proxy0". But this feature is usually not needed anyway. (This [issue](http://opensource.atlassian.com/projects/spring/browse/SPR-3642) might be solved in the future.)

Since two objects are created (the instance of the class and the proxy as instance of a sub class) the constructor is called twice. In general this should not matter. I consider changing the class' state based on constructor calls a code smell anyway.

You have CGLib as additional dependency.

Both options suffer from some *issues* (not really issues, but you have to be aware of them):

Most important [proxy-based AOP](http://static.springframework.org/spring/docs/2.5.x/reference/aop.html#aop-understanding-aop-proxies) only works from "outside". Internal method calls are never intercepted.

Second, the object has to be managed by the Spring container. Instantiating it yourself using newdoes not work.

The proxies are not Serializable.

Regarding performance of the one or the other method I have read different things. I remember having read a blog post about CGLib proxies being better, one of the comments says dynamic proxies are. Actually the Spring reference has a [paragraph](http://static.springframework.org/spring/docs/2.5.x/reference/aop-api.html#aop-api-proxying-class) on this itself:

There's little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. **Performance should not be a decisive consideration in this case.**

I emphasized the last sentence by intention.  
Especially for enforcing programming to interfaces and allowing constructor injection I strongly prefer the JDK dynamic proxies.

AOP using Decorator Pattern

# Introduction

You can also achieve AOP in core java using Decorator Design Pattern. The example is given below.

# AbstractAccount.java

**package** com.ddlab.rnd.aop1;  
  
**public abstract class** AbstractAccount {  
  
 **public abstract void** withdraw( String actNo , **int** amount);  
}

# RetailAccount.java

**package** com.ddlab.rnd.aop1;  
  
**public class** RetailAccount **extends** AbstractAccount {  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
  
 System.***out***.println(**"Amount "** + amount + **" is debited from the account no "** + actNo);  
 }  
}

# BeforeDecorator.java

**package** com.ddlab.rnd.aop1;  
  
**public class** BeforeDecorator **extends** AbstractAccount {  
  
 **private** RetailAccount **retailAccount**;  
  
 **public** BeforeDecorator(RetailAccount retailAccount) {  
 **this**.**retailAccount** = retailAccount;  
 }  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
 System.***out***.println(**"Doing validation before debiting from acount..."**);  
 **retailAccount**.withdraw(actNo,amount);  
 }  
}

# AroundDecorator.java

**package** com.ddlab.rnd.aop1;  
**public class** AroundDecorator **extends** AbstractAccount {  
  
 **private** RetailAccount **retailAccount**;  
  
 **public** AroundDecorator(RetailAccount retailAccount) {  
 **this**.**retailAccount** = retailAccount;  
 }  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
  
 System.***out***.println(**"Doing before withdrawing money ........."**);  
 **retailAccount**.withdraw(actNo, amount);  
 System.***out***.println(**"Doing after withdrawing money ........."**);  
  
 }  
}

# TestAOPDecorator.java

**package** com.ddlab.rnd.aop1;  
**public class** TestAOPDecorator {  
  
 **public static void** main(String[] args) {  
  
 RetailAccount account = **new** RetailAccount();  
*// BeforeDecorator beforeDecorator = new BeforeDecorator(account);* AroundDecorator aroundDecorator = **new** AroundDecorator(account);  
 aroundDecorator.withdraw(**"1122334455"**,500);  
 }  
}