ASPECT ORIENTED PROGRAMMING

Aspect Oriented programming (AOP) is often defined as a programming technique that promotes separation of system services (or concerns) from the business logic. The different system services (concerns) are logging, transaction management, exceptions, security, etc. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.
The DI helps us decouple our application objects from each other, AOP helps us decouple cross-cutting concerns from the objects that they effect.
AOP makes it possible to modularize these concerns and then apply them declaratively to the components.

The Advantages with AOP

1. The business components look very clean having only business logic without services.
2. All services are implemented in a common place, which simplifies code maintenance.
3. Promotes re-usability.
4. Clear demarcation (separation) among developers.

AOP Terminology

Aspect

Every system service is an aspect, which is written in one place, but can be integrated with components.

Advice

Advice is the actual **implementation of an aspect** i.e., it defines the job that an aspect will perform. Advice defines both **what** and **when** of an aspect.
Spring aspects can work with five kinds of advices:

Advice type	Interface	Description
Before	org.springframework.aop.MethodBeforeAdvice	Called before target method is invoked.
After	org.springframework.aop.AfterAdvice	Called after the target method returns regardless of the outcome.
After-returning	Org.springframework.aop.AfterReturningAdvice	Called after target method completes successfully.
Around	Org.aopalliance.intercept.MethodInterceptor	Both Before and After
After-throwing	org.springframework.aop.ThrowsAdvice	Called when target method throws an exception.

Joinpoint

A joinpoint is a point in the execution of the application where an aspect can be plugged in. Spring only supports method joinpoints.

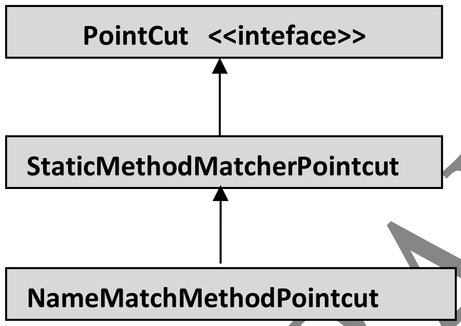
Pointcut

A pointcut defines at what joinpoints advice should be applied, instead of applying advice at all joinpoints i.e., pointcuts help narrow down the join points advised by an aspect. The pointcuts are configured in the spring configuration file.

There are two types of Pointcuts:

I. **Static Pointcut**

Static pointcuts define advice that is always executed.
The public interface org.springframework.aop.PointCut is the root interface for all spring built-in pointcuts. One of built-in static point cut is org.springframework.aop.support.NameMatchMethodPointcut, which is inherited from org.springframework.aop.support.StaticMethodMatcherPointcut.



II. **Dynamic Pointcut**

Dynamic pointcuts determine if advice should be executed by examining the runtime method arguments.

Introduction

An introduction advice allows us to add new methods or attributes to existing classes. For example, we would create an Auditable advice class that keeps the state when an object was last modified. This could be as simple as having one method, setLastModified(Date), and an instance variable to hold this state. This can then be introduced to existing classes without having to change them, giving them new behavior and state.

Target

The target class is a **pure business component** which is being advised. Without AOP, this class would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, the target class is free to focus on its primary logic.

Proxy

A proxy is the object created at runtime after applying advice to the target object.
The proxy <bean> element is defined using org.springframework.aop.framework.**ProxyFactoryBean**.

Proxy = target + advice(s)

Weaving

Weaving is the process of applying aspects to a target object to create a new proxied object.

Advisor

Combining Advice and Pointcut where the advice should be executed is called Advisor.

Advisor = Advice + Pointcut(s)

The public interface PointcutAdvisor the root interface for all built-in Advisors. Most of the spring's built-in pointcuts also have a corresponding PointcutAdvisor.

Built-it Pointcut	Matching Built-in Advisor
NameMatchMethodPointcut	**NameMatchMethodPointcutAdvisor**

Since introduction advice is applied only at class level, introductions have their own advisor: IntroductionAdvisor. Spring also provides default implementation called as DefaultIntroductionAdvisor.

Spring supports following AOPs:

1. Proxy based AOP
2. Declarative based AOP
3. Annotation based AOP

Proxy based AOP

It is a legacy approach hence needs following classes (or interfaces) from Spring API:

1. org.springframework.aop.MethodBeforeAdvice<<interface>>
2. org.springframework.aop.AfterAdvice<<interface>>
3. org.springframework.aop.AfterReturningAdvice<<interface>>
4. org.aopalliance.intercept.MethodInterceptor<<interface>>
5. org.springframework.aop.ThrowsAdvice<<interface>>
6. org.springframework.aop.support.NameMatchMethodPointcut
7. org.springframework.aop.support.NameMatchMethodPointcutAdvisor
8. org.springframework.aop.framework.ProxyFactoryBean

Application #1: Write a Proxy based AOP application using Spring API

```
package edu.aspire;
public interface Instrument {
        public void play();
}
package edu.aspire;
public class Guitar implements Instrument {
         public void play() {
                  System.out.println("Strum strum strum");
         }
}
package edu.aspire;
public interface Performer {
        public void perform() throws PerformanceException;
}
package edu.aspire;
public class Instrumentalist implements Performer {
        private Instrument instrument;
        public void setInstrument(Instrument instrument) { this.instrument = instrument; }
        public Instrument getInstrument() { return instrument; }
        public void perform() throws PerformanceException { instrument.play(); }
}
```

// **PerformanceException.java**

```
package edu.aspire;
public class PerformanceException extends Exception {
        public PerformanceException() { super(); }
        public PerformanceException(String message) { super(message); }
}
```

//**Audience.java**

```
package edu.aspire;
import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;
public class Audience implements MethodBeforeAdvice, AfterReturningAdvice, ThrowsAdvice, MethodInterceptor {
        @Override
        public void before(Method method, Object[] data, Object target) throws Throwable {
                System.out.println("The audience is taking their seats.");
        }

       /*@Override
        public void before(Method method, Object[] data, Object target) throws Throwable {
                System.out.println("The audience is turning off their cellphones");
        }*/

        @Override
        public void afterReturning(Object returnValue, Method method, Object[] data, Object target) throws Throwable{
                System.out.println("CLAP CLAP CLAP CLAP CLAP");
        }
        public void afterThrowing(Method method, Object[] data, Object target, PerformanceException e) {
                System.out.println("Boo! We want our money back!");
        }
        @Override
        public Object invoke(MethodInvocation joinpoint) throws Throwable {
                long start = System.currentTimeMillis();

                Object obj = joinpoint.proceed(); //automatically calls target method

                long end = System.currentTimeMillis();
                System.out.println("***The performance took***:" + (end - start)  + " milliseconds");
                return obj;
        }
}
```

# applicationContext_proxy_based_aop.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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-4.2.xsd">
        <!-- Target class -->
        <bean id="target" class="edu.aspire.Instrumentalist">
                <property name="instrument">
                        <!—inner bean -- >
                        <bean class="edu.aspire.Guitar"/>
                </property>
        </bean>
        <!—advice(s) -- >
        <bean id="audience" class="edu.aspire.Audience" />

        <!—advisor -- >
        <!--
        <bean id="advisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
                        <property name="advice">
                                <ref bean="audience"/>
                        </property>
                        <!-- static point cut -->
                        <property name="mappedNames">
                                <array>
                                        <value>perform</value>
                                </array>
                        </property>
        </bean> -- >
```

```
	<!—Proxy  class -- >
	<bean id="proxybean" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="target">
			<ref bean="target" />
		</property>
		<property name="interceptorNames">
			<array>
				<value>audience</value>
				<!-- <value>advisor </value> -- >
			</array>
		</property>
	</bean>
</beans>
```

```
//Client code
package edu.aspire.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import edu.aspire.Performer;
public class ClientTest {
	public static void main(String[] args) throws Exception {
		ApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext_proxy_based_aop.xml");
		Performer pRef = (Performer)context.getBean("proxybean");
		pRef.perform();
	}
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) **%SPRING_HOME%\libs\spring-aop-4.2.5.RELEASE.jar**
7) **%HOME%\Softwares\jars\aopalliance.jar**

Dis Advantages

Working with Proxy Based AOP using ProxyFactoryBean is overcomplicated when compared to Declarative or Annotation Based AOP.

Defining Pointcuts

In Spring AOP, pointcuts are defined by using AspectJ's pointcut Expression Language, which is used in both Declarative based AOP and Annotation based AOP. The following pointcut designators from AspectJ supported in Spring AOP:

AspectJ designator	Description
Args()	Limits joinpoint matches to the execution of methods whose arguments are instances of the given types.
@args()	Limits joinpoint matches to the execution of methods whose arguments are annotated with the given annotation types.
execution()	Matches joinpoints that are method executions
This()	Limits joinpoint matches to those where the bean reference of the AOP proxy is of a given type
Target()	Limits joinpoint matches to those where the target object is of given type.
@target()	Limits matching to joinpoints where the class of the executing object has an annotation of the given type.
Within()	Limits matching to joinpoints within certain types
@within()	Limits matching to joipints within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
@annotation	Limits joinpoint matches to those where the subject of the joipoint has the given annotation.

The **execution** designator is the only one that actually performs matches. The other designators are used to limit those matches. Hence, the execution is the primary designator we'll use in every pointcut definition we write.
The following pointcut expression is used to apply advice whenever Performer's perform() method is executed:

execution(* edu.aspire.Perfomer.perform(..))

We used the execution() designator to select the Perfomer's perform() method. The asterisk indicates that any return type is acceptable. Then we specify the fully qualified class name and the name of the method we want to select. For the method's parameter list, we use the double dot (..), indicates that the pointcut should select any perform() method irrespective of the argument list.
Let's suppose that we want to confine the reach of that pointcut to only the edu.aspire package. In that case, we could limit the match by taking on a **within()** designator as shown below:

execution * edu.aspire.Perfomer.perform(..) && within(edu.aspire.*)

The bean() designator identifies beans by their ID within a pointcut expression. The bean() designator takes a bean ID or name as an argument and limits the piontcut's effect to that specific bean.

execution (* edu.aspire.Perfomer.perform(..) and bean(eddie))

Declarative based AOP

Working with ProxyFactoryBean is difficult. The following elements are from spring's aop configuration namespace:

AOP configuration element	Purpose
<aop:before>	Defines AOP before advice
<aop:after>	Called after target method returns regardless of outcome.

<aop:after-returning>	Called after target method completes successfully.
<aop:around>	Both before and after.
<aop:after-throwing>	Called when target object method throws an exception.
<aop:pointcut>	Defines a pointcut.
<aop:advisor>	Defines a advisor
<aop:aspect>	Defines aspect
<aop:config>	The top-level AOP element.
<aop:aspectj-autoproxy/>	Enables annotation driven aspects using @AspectJ

We see that Audience contained all of the functionality needed for an audience, but none of the details to make it as aspect. That left us having to declare advice and pointcuts in XML.

Application #3: Declarative based AOP

The following files are identical as before.

1. Instrument.java
2. Guitar.java
3. Performer.java
4. Instrumentlist.java
5. PerformanceException.java

```
package edu.aspire;
import org.aspectj.lang.ProceedingJoinPoint;
public class Audience1 {
	public void takeSeats() {
		System.out.println("The audience is taking their seats.");
	}
	public void turnOffCellPhones() {
		System.out.println("The audience is turning off their cellphones");
	}
	public void applaud() {
		System.out.println("CLAP CLAP CLAP CLAP CLAP");
	}
	public void demandRefund() {
		System.out.println("Boo! We want our money back!");
	}
	public void watchPerformance(ProceedingJoinPoint joinpoint) {
		try {
			long start = System.currentTimeMillis();

			joinpoint.proceed();

			long end = System.currentTimeMillis();
			System.out.println("The performance duration is :" + (end - start)  + " milliseconds");
		} catch (Throwable e) {
			System.out.println("Boo! We want our money back!");
		}
	}
}
```

# **applicationContext_declarative_based_aop.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:aop="http://www.springframework.org/schema/aop"
	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-4.2.xsd
	http://www.springframework.org/schema/aop
	http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">
	<!-- target -->
	<bean id="eddie" class="edu.aspire.Instrumentalist1">
		<property name="instrument">
			<!-- inner bean -->
			<bean class="edu.aspire.Guitar1" />
		</property>
	</bean>

	<!—audience class -->
	<bean id="audience" class="edu.aspire.Audience1" />

	<aop:config>
		<aop:aspect ref="audience">
			<!-- Named pointcut to eliminate redundant pointcut definitions -->
			<aop:pointcut id="performance"
					expression="execution(* edu.aspire.Performer1.perform(..))" />
			<aop:before pointcut-ref="performance" method="takeSeats" />
			<aop:before pointcut-ref="performance" method="turnOffCellPhones" />
			<aop:after-returning pointcut-ref="performance" method="applaud" />
			<!--  <aop:around pointcut-ref="performance" method="watchPerformance" /> -->
			<!--  <aop:after-throwing pointcut-ref="performance" method="demandRefund" /> -->
		</aop:aspect>
	</aop:config>
</beans>
```

//JUnit Test case

```
package edu.aspire.test;
```

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.Performer1;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_declarative_based_aop.xml")
public class AspectTest1{
        @Autowired
        ApplicationContext context;

        @Test
        public void audienceShouldApplaud() throws Exception {
                Performer1 eddie = (Performer1) context.getBean("eddie");
                eddie.perform();
        }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) %SPRING_HOME%\libs\ spring-aop-4.2.5.RELEASE.jar
7) %HOME%\Softwares\jars\aopalliance.jar
8) **%SPRING_HOME%\libs\spring-test-4.2.5.RELEASE.jar**
9) **%HOME%\Softwares\Jars\junit-4.12.jar**
10) **%HOME%\Softwares\Jars\aspectjweaver-1.6.6.jar**

Note: Junit 4 and above additionally needs hamcrest-core.jar

The **@RunWith(SpringJUnit4ClassRunner.class)** tells JUnit to run using spring's test module.
The **@ContextConfiguration**("…xml") is used to specify which spring configuration file to load. This annotation automatically creates appropriate context based on argument.

Annotation based AOP

A key feature introduced in Aspectj 5 is the ability to use annotations to create aspects. The AspectJ's annotation-oriented model makes it simple to turn any class into an aspect by adding few annotations around. This new feature is commonly referred to as **@AspectJ**.
The following files are identical as before.

1. Instrument.java
2. Guitar.java
3. Performer.java
4. Instrumentlist.java
5. PerformanceException.java

But with @AspectJ annotations, we can rewrite Audience class and turn it into an aspect without the need for any additional classes or bean declarations in spring configuration file. The following shows the modified Audience class, now annotated to be an aspect:

Application #4: Example of Annotation based AOP

```
package edu.aspire;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class Audience2 {  //aspect class
	@Pointcut("execution(* edu.aspire.Performer2.perform(..))")
	public void performance() {}

	@Before("performance()")
	public void takeSeats() { //before adivce
		System.out.println("The audience is taking their seats.....");
	}

	@Before("performance()")
	public void turnOffCellPhones() { //before advice
		System.out.println("The audience is turning off their cellphones");
	}

	@AfterReturning("performance()")
	public void applaud() { //after-returning advice
		System.out.println("CLAP CLAP CLAP CLAP CLAP");
	}

	@AfterThrowing("performance()")
	public void demandRefund() {//after-throwing advice
		System.out.println("Boo! We want our money back!");
	}
```

```
        @Around("performance()")
        public void watchPerformance(ProceedingJoinPoint joinpoint) {//around advice
                try {
                        long start = System.currentTimeMillis();

                        joinpoint.proceed();

                        long end = System.currentTimeMillis();
                        System.out.println("***The performance took:" + (end - start) + " milliseconds***");
                } catch (Throwable e) {
                        System.out.println("***Boo! We want our money back***!");
                }
        }
}
```

Because the Audience class contains everything that's needed to define its own pointcuts and advice, there's no need to add pointcut and advice declarations in Spring configuration file.

We have to add **<aop:aspectj-autoproxy/>** to turn @Aspectj-annotated classes into proxy advice.

# **applicationContext_annotation_based_aop.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-4.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">
        <bean id="eddie" class="edu.aspire.Instrumentalist2">
                <property name="instrument">
                        <bean class="edu.aspire.Guitar2" />
                </property>
        </bean>
        <bean id="audience" class="edu.aspire.Audience2" />
        <aop:aspectj-autoproxy />
</beans>
```

//JUnit Test code

```
package edu.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.Performer2;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_annotation_based_aop.xml")
public class AspectTest2 {
	@Autowired
	ApplicationContext context;

	@Test
	public void audienceShouldApplaud() throws Exception {
		Performer2 eddie = (Performer2) context.getBean("eddie");
		eddie.perform();
	}
}
```

Note: Jar files are same as previous application.