[Aspect Oriented Programming (AOP) 2](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308742)

[Introduction 2](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308743)

[AOP terminology 3](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308744)

[Add AspectJ dependencies 3](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308745)

[Example1: Apply Logging aspect before a method call 4](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308746)

[Pointcuts & Wildcard Expressions 4](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308747)

[JoinPoint & Advice Arguments 5](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308748)

[After Advice 5](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308749)

[Around Advice 6](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308750)

[Naming Conventions 6](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308751)

[Custom Annotation 6](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308752)

[AspectJ XML Configuration Example 6](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308753)

[Question – Why we apply advice on any method of Final class? 6](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308754)

[How Spring AOP works OR Proxies Design Pattern? 7](file:///C:\Users\nitik\Downloads\Spring%20Framework%20-%20AOP.doc#_Toc421308755)

# Aspect Oriented Programming (AOP)

### Introduction

Spring AOP (**Aspect-oriented programming**) framework is used to modularize cross-cutting concerns in aspects. Put it simple, it’s just an interceptor to intercept some processes, for example, when a method is execute, Spring AOP can hijack the executing method, and add extra functionality before or after the method execution.

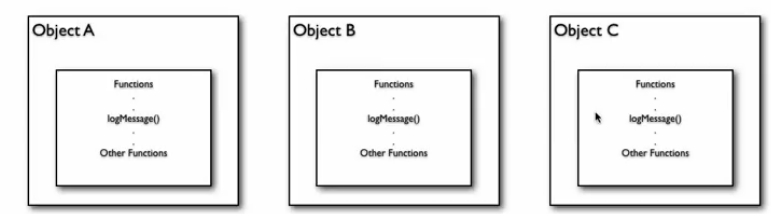
A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

**Question**- How AOP is different from OOPs & why we need it?

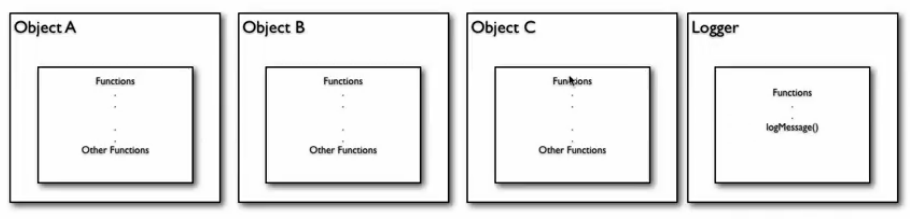
*Aspect-Oriented Programming* (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects.

For example –

**Requirement**: Introduce a new logMessage() inside every object.



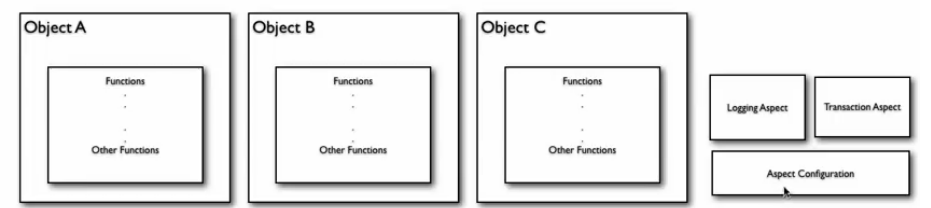
In OOP-

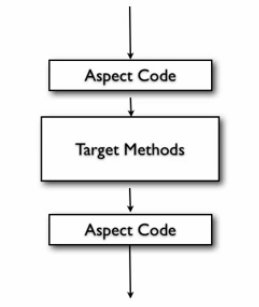


Problems-

1. Need to add cross-cutting concerns (not be part of problem domain, it may be infrastructure, security, transaction, etc) into each Object
2. Need to modify existing code of each object
3. If modification required inside Cross-cutting then need to modify everywhere (if based on legacy codebase)

In AOP-





### AOP terminology

* **Aspect -** A module which has a set of APIs providing cross-cutting requirements. It contains number of Advice. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. In Spring AOP, aspects are implemented using regular classes (**XML configuration**) or regular classes annotated with the **@Aspect** annotation
* **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. Different types of advice include "around," "before" and "after" advice. 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
  + *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
  + Around advice – Run around the method execution, combine all three advices above.
* **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.
* **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.
* **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.

### Add AspectJ dependencies

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.7.4</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjtools</artifactId>

<version>1.7.4</version>

</dependency>

### Example1: Apply Logging aspect before a method call

1. Create Customer pojo & Customer service & its bean configuration

com.gps.spring.aop.pojo.Customer.java

com.gps.spring.aop.service.CustomerService.java

CustomerConfig.xml

com.gps.spring.aop.client.CustomerServiceAopClient.java

1. Run & execute this DI related stuff
2. Create an Logging Aspect class & define before advice with Joinpoints using SPEL

com.gps.spring.aop.aspect.LoggingAspect.java

1. Enable AspectJ auto proxy inside configuration file & add bean for LoggingAspect

<aop:aspectj-autoproxy />

1. Run & execute

### Pointcuts & Wildcard Expressions

1. Above Advice & Joinpoints will be applicable for every method whose name is getName(). Now our requirement is apply it only on Customer.getName() method.

Then we need restrict by mentioning fully qualified classname. Now above advice will be-

@Before("execution(public String com.gps.spring.aop.pojo.Customer.getName())")

**public** **void** loggingAdvice() {

System.***out***.println("Advice run, Get Method called");

}

com.gps.spring.aop.aspect.LoggingAspect.java

1. Apply an advice on different methods

Use wildcard to achieve it.

Need to apply an advice to any of the get Method.

@Before("execution(public String get\*())")

**public** **void** loggingAdvice() {

System.***out***.println("Advice run, Get Method called");

}

com.gps.spring.aop.aspect.LoggingAspect.java

1. Apply an advice on any method start with “get” & have arguments

@Before("execution(public String get\*(\*))")

1. Apply an advice on any method start with “get” & weather have arguments or not

@Before("execution(public String get\*(..))")

1. Apply multiple advice on single point

@Before("execution(public String getName())")

**public** **void** loggingAdvice() {

System.***out***.println("Advice run, Get Method called");

}

@Before("execution(public String getName())")

**public** **void** secondAdvice() {

System.***out***.println("Second Advice run, Get Method called");

}

1. Apply multiple advice on single point using **pointcuts**

Create a pointcuts for a complex expression & use it inside every advice (instead to define complex expression everywhere inside each advice).

// Define a point for any complex expression

@Pointcut("execution(public String getName())")

**public** **void** pointcutForGetName() {

}

// Use defined pointcut reference instead to use complex expression

@Before("pointcutForGetName()")

**public** **void** loggingAdvice() {

System.***out***.println("Advice run, Get Method called");

}

// Use defined pointcut reference instead to use complex expression

@Before("pointcutForGetName()")

**public** **void** secondAdvice() {

System.***out***.println("Second Advice run, Get Method called");

}

1. Apply advice to all of the methods of a class

@Pointcut("within(com.gps.spring.aop.dao.EmployeeManagerImpl)")

**public** **void** pointcutForGetName() {

}

1. Apply advice to all of the methods of all classes inside package

@Pointcut("within(com.gps.spring.aop.dao.\*)")

**public** **void** pointcutForGetName() {

}

1. Apply advice to all of the methods of all classes inside package & its sub-packages

@Pointcut("within(com.gps.spring.aop.dao..\*)")

**public** **void** pointcutForAllDao() {

}

1. Apply multiple pointcuts into single advice.

@Before("pointcutForGetName() && pointcutForAllDao()")

**public** **void** secondAdvice() {

System.***out***.println("Second Advice run, Get Method called");

}

### JoinPoint & Advice Arguments

1. **JoinPoint** – Using this class we can get details about the actual method get called OR where advice will be applied. It means all the places in our code where Advice will apply.

**For Example**-

CustomerAppConfig.xml

com.gps.spring.aop.aspect.LoggingAspectWithJoinPoint.java

com.gps.spring.aop.client.CustomerServiceJoinPointClient.java

1. **Advice** – Can get arguments value inside advice via **args**.

**For Example**-

CustomerAppConfig.xml

com.gps.spring.aop.aspect.LoggingAspectWithJoinPoint.java

com.gps.spring.aop.client.CustomerServiceJoinPointClient.java

### After Advice

**For Example**-

RoleConfig.xml

com.gps.spring.aop.pojo.RoleVO.java

com.gps.spring.aop.aspect.LoggingRoleAspect.java

com.gps.spring.aop.client.RoleAopClient.java

### Around Advice

Use case where we want to apply advice on both Before & After then we will use **around** advice. Using this advice we can also play with return value with the help of ProceedingJoinPoint class.

**For Example**-

RoleConfig.xml

com.gps.spring.aop.pojo.RoleVO.java

com.gps.spring.aop.aspect.LoggingRoleAspect.java

com.gps.spring.aop.client.RoleAopClient.java

### Naming Conventions

* Put all aspects inside aspect package
* Keep complete MVC design as it is like Controller, Service, DAO, Model
* Put all resources inside respective package & then import all of these into single one metadata configuration file
* Always try to put all Implementation inside “impl” package which should be separate one

For ex- interface inside “com.gps.spring.aop.service” then its implementation should inside “com.gps.spring.aop.service.impl” package

### Custom Annotation

We it is difficult to decide or write any Pointcut expression then we can use custom annotation to do such Pointcut.

**For Example**-

RoleConfig.xml

com.gps.spring.aop.pojo.RoleVO.java

com.gps.spring.aop.aspect.Loggable.java

com.gps.spring.aop.aspect.LoggingRoleAspect.java

com.gps.spring.aop.client.RoleAopClient.java

### AspectJ XML Configuration Example

* An aspect is declared using the <aop:aspect> element. A pointcut helps in determining the join points to be executed with different advices.

<bean id="**transactionAspectBean**"

class="com.gps.spring.aop.aspect.EmployeeCRUDTransactionAspect" />

<aop:pointcut id="**transactionOperation**"

expression="execution(\*

com.gps.spring.aop.dao.EmployeeManager.getEmployeeById(..))" />

<aop:aspect id="transactionAspect" ref="**transactionAspectBean**">

<aop:before pointcut-ref="**transactionOperation**" method="**getEmployeeById**" />

</aop:aspect>

* **For Example**-

com.gps.spring.aop.pojo.EmployeeDTO.java

com.gps.spring.aop.dao.EmployeeManager.java

com.gps.spring.aop.dao.EmployeeManagerImpl.java

com.gps.spring.aop.aspect.EmployeeCRUDLoggingAspect.java

com.gps.spring.aop.aspect.EmployeeCRUDTransactionAspect.java

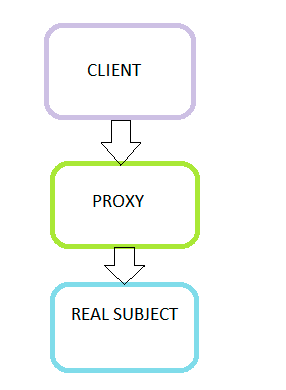
com.gps.spring.aop.client.EmployeeDaoAopClient.java

EmployeeAppConfig.xml

### Question – Why we can’t apply advice on any method of Final class?

### How Spring AOP works OR Proxies Design Pattern?

We use Object oriented concepts to achieve AOP. In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.



**For Example**-

com.gps.spring.aop.service.FactoryService.java

com.gps.spring.aop.aspect.ProxyAdvice.java

com.gps.spring.aop.pojo.proxies.RoleVOProxy.java

com.gps.spring.aop.client.ProxyClientApp.java