Prepared By Radha V Krishna

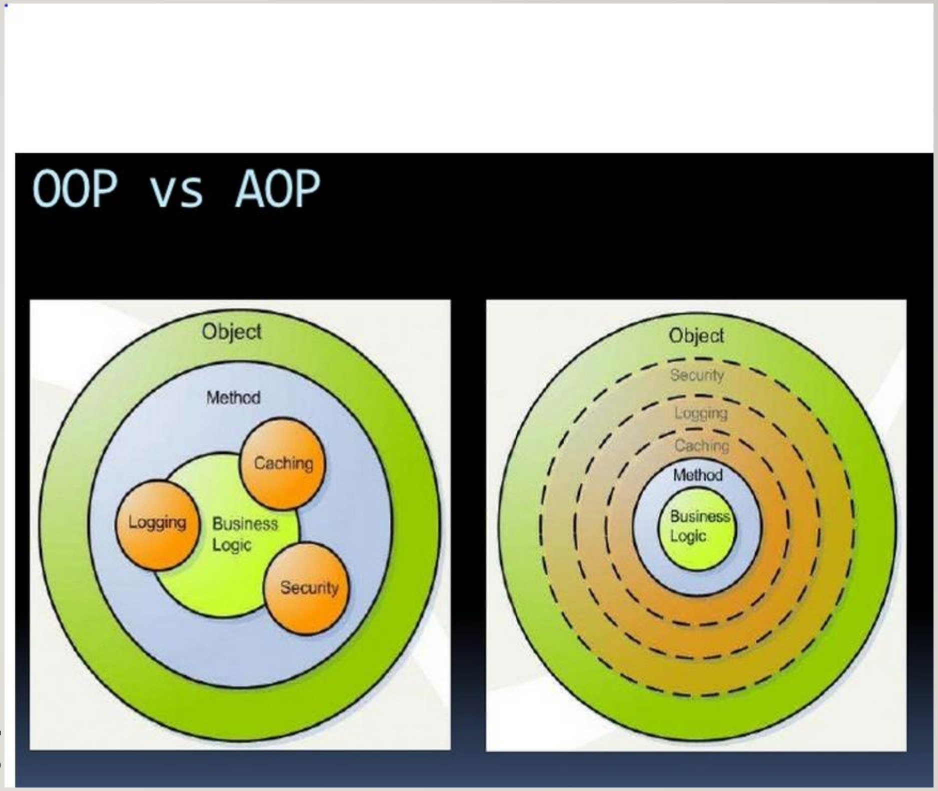
AOP

ASPECT ORIENTED PROGRAMMING

WHAT IS AOP

- *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, logging, security etc that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.)

AOP allows you us to systematically apply a set of code modules,called aspects,to another (typically larger) body of target code.The end result is a shuffling of aspect code with the body of the code that it cuts across.however the crosscutting aspects are coded and maintained seperately and the target code can be coded and maintained completely free of the crosscutting aspects.in AOP lingo,this is called *separation of concerns*.



AOP - LINGO

Prepared By Radha V Krishna

Aspect: a modularization of a concern that cuts across multiple classes.

```
@Aspect
public class StudentAspect {
@Before("execution(public String getName())")
public void getNameAdvice(){
System.out.println("Executing Advice on getName()");
}
```

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.

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..

```
package com.classes;

public class Student {
private String name="sam";

public String getName() {
return name;
}
.........
```

Advice: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.

This advice executes before a getName() is called in the same package with the same prototype.

//spring.xml

```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">

<!-- Enable AspectJ style of Spring AOP -->
<aop:aspectj-autoproxy />
<bean name="studentBean" class="com.classes.Student">
</bean>
 <bean name="studentAspect" class="com.classes.StudentAspect" />
 <bean name="studentAroundAspect" class="com.classes.StudentAroundAspect"
/>
</beans>
```

Enables AspectJ compiler

- 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.

```
public class TestClient {
public static void main(String[] args) {
ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("/spring.xml");
Student stu =(Student)ctx.getBean("studentBean");
System.out.println("Name:"+stu.getName());
```

Output:

```
Executing Advice on getName()
Name:sam
```

Advice types

```
@Before("execution(* com.classes.*.get*())")
public void getAllAdvice(){
System.out.println("Service method getter called");
}
```

Executes before all methods starting with get pattern in all classes of com.classes package Like getName() ,getAge() with matching arguments etc..

```
@After("execution(public void set*(*))")
public void setNameAdvice(){
System.out.println("Executing Advice on set***()");
}
```

Executes after all methods starting with set pattern in present package setName(..) ,setAge(..) with any arguments etc..

```
@AfterThrowing("execution(public void test())")
public void getThrowAdvice()
{
System.out.println("Executing advice on throwing exception..");
}
```

Executes when test method in the present package throws exception.

```
@AfterReturning("execution(public String getName())")
public void getReturnAdvice()
{
System.out.println("Executing advice on Returning exception..");
}
```

After returning advice: Advice to be executed after a join point completes normally: for example, if a method return without throwing an exception.

https://docs.spring.io/spring/docs/2.5.x/reference/aop.html

```
@Aspect
public class StudentAroundAspect {
@Around("execution(* com.classes.Student.getName())")
public Object studentAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
System.out.println("Before invoking getName() method--Around");
Object value = null;
try {
value = proceedingJoinPoint.proceed();
} catch (Throwable e) {
e.printStackTrace();
}
System.out.println("After invoking getName() method. Return value="+value);
return value;
}
}
```

```
Student stu = (Student)ctx.getBean("studentBean");
System.out.println(stu.getName());
```

Output:

```
Executing Advice on getName()
Before invoking getName() method--Around
After invoking getName() method. Return
value=sam
Executing advice on Returning exception..
sam
```

In Spring boot add starter dependency , configuration not required

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
```