Spring AOP

Sample code

```
public boolean login(String uname, String pwd) {
        logger.debug("User " + uname + " is trying to login");
//cross cutting concern
        if(Util.authenticate(uname, pwd)) {
                logger.debug("User " + uname + " login
successful");  //cross cutting concern
        }
        else {
                logger.debug("User " + uname + " login failed");
//cross cutting concern
        }
}
```

AOP Introduction

- Spring Framework is developed on two core concepts:
  - Dependency Injection
  - Aspect Oriented Programming (AOP)
- Enterprise applications come across several cross cutting concerns applicable to objects & modules.
- Using AOP we can cut the cross cutting concerns.

AOP concepts

Aspect:

An aspect is a class that implements cross cutting concerns.

Join Point:

A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.

Advice:

Advices are the methods that define actions taken for a particular join point. For example Struts2 interceptors or Servlet Filters.

Pointcut:

Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not.

AOP concepts continue…

Target Object:

They are the objects on which advices are applied.

AOP proxy:

Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes.

Pointcut expressions

- @Pointcut("execution(* Arithmetic.divide(..))")
- @Pointcut("within(com.spring.someapp.web..*)")
- @Pointcut("execution(public * *(..))")
- @Pointcut("target(com.spring.service.TradeService)")
- @Pointcut("args(java.io.Serializable)")
- @Pointcut(“@annotation(org.springframework.transaction.annotation.Transactional)”)

AOP Advice Types

@Before:

These advices runs before the execution of join point methods. We can use @Before annotation to mark an advice type as Before advice.

@After:

An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using @After annotation.

@AfterReturning:

Sometimes we want advice methods to execute only if the join point method executes normally. We can use @AfterReturning annotation to mark a method as after returning advice.

AOP Advice Types continue…

@AfterThrowing:

This advice gets executed only when join point method throws exception. We use @AfterThrowing annotation for this type of advice.

@Around:

Using Around advice, we can write advice code that gets executed before and after the execution of the join point method. We use @Around annotation to create around advice methods.

AOP configuration

AOP can be configured in 2 ways:

- Annotation based configuration
- XML based configuration

Annotation based configuration

```
<bean
class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJ
AutoProxyCreator" />
@Aspect
public class ArithmeticAspect {
   @Pointcut("execution(* Arithmetic.divide(..))")
   public void parameter_pointcut(){}//pointcut name

   @Before("parameter_pointcut()")//applying pointcut on before advice
   public void checkParameters(JoinPoint jp)//it is advice (before advice)
   {
            Object args[] = jp.getArgs();
     System.out.println("checking parameters: " + args[0] + " -- " + args[1]);
   }
}
```

XML based configuration

```
<aop:config>
  <aop:aspect id="myaspect" ref="arithmeticAspectBean" >
    <aop:pointcut id="pointCutBefore"   expression="execution(* com.spring.aop.Arithmetic.divide(..))" />
    <aop:before method="checkParameters" pointcut-ref="pointCutBefore" />
  </aop:aspect>
</aop:config>
```