Aspect Oriented Programming (AOP)

Aspect Oriented Programming(AOP)

WHY

Separating cross-cutting concerns(=repeatative tasks) from the business logic/ request handling /persistence

IMPORTANT

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other(eg : Controller separate from Service or DAO layers) and AOP helps you decouple cross-cutting concerns from the objects that they affect.(eg : transactional code or security related code can be kept separate from B.L or exception handling code from request handling method)
eg of ready made aspects provided by SC : tx management,security , exc handling

eg scenario --

Think of a situation Your manager tells you to do your normal development work(eg - write stock trading appln) + write down everything you do and how long it takes you.

A better situation would be you do your normal work, but another person observes what youre doing and records it and measures how long it took.

Even better would be if you were totally unaware of that other person and that other person was able to also observe and record , not just yours but any other peoples work and time.

That's separation of responsibilities.   --- This is what spring offers you through AOP.

It is NOT an alternative to OOP BUT it complements OOP.

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.(concern=task/responsibility) such as transaction management,logging,security --- that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP jargon)

Enables the modularization of cross cutting concerns(=task)
Eg : Logging,Security,Transaction management, Exception Handling

Similar in functionality to ---In EJB framework -- EJBObject
Struts 2 -- interceptors
Servlet -- filters.
RMI -- stubs
Hibernate --- proxy (hib frmwork -- lazy --- load or any--many associations --rets typically un-inited proxy/proxies)

AOP with Spring Framework

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework.

Like DI, AOP supports loose coupling of application objects.

The functionalities that span multiple points of an application are called cross-cutting concerns.

With AOP, applicationwide concerns(common concerns-responsibilities or cross-cutting concerns like eg - declarative transactions , security,logging,monitoring,auditing,exception handling....) are decoupled from the objects to which they are applied.

Its better for application objects(service layer/controller/rest controller/DAO) to focus on the business domain problems that they are designed for and leave certain ASPECTS to be handled by someone else.

Job of AOP framework is --- Separating these cross-cutting concerns(repeatative tasks) from the core business logic

AOP is like triggers in programming languages such as Perl, .NET.

Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

Key Terms of AOP

Advice : Action(=cross cutting concern) to take either before/after or around the method (req handling logic ,OR B.L OR data access logic) execution. eg: transactional logic(begin tx,commit,rollback,session closing)

Advice describes WHAT is to be done & WHEN it's to be done.

eg : logging. It is sure that each object will be using the logging framework to log the event happenings , by calling the log methods. So, each object will have its own code for logging. i.e the logging functionality requirement is spread across multiple objects (call this as Cross cutting Concern, since it cuts across multiple objects). Wont it be nice to have some mechanism to automatically execute the logging code, before executing the methods of several objects?

---

2. Join Point : Place in application WHERE advice should be applied.(i.e which B.L methods should be advised)
(Spring AOP, supports only method execution join point)

---

3. Pointcut : Collection of join points.

It is the expression used to define when a call to a method should be intercepted.
eg :
@Pointcut("execution (Vendor com.app.bank.*.*(double))")
advice logic{....}

---

4. Advisor Group of Advice and Pointcut into a single unit.

---

5. Aspect : class representing advisor(advice logic + point cut definition)--
@Asepct -- class level annotation.

---

6. Target : Application Object containing Core domain logic.(To which advice gets applied at specified join points) --supplied by Prog

---

7. Proxy : Object created after applying advice to the target object(created by SC dynamically by implementing typically service layer i/f) ---consists of cross cutting concern(repeatative jobs , eg : tx management,security, exc handling)

8.Weaving -- meshing(integration) cross cutting concern around B.L
(3 ways --- compile time, class loading time or spring supported --dynamic --method exec time or run time)

Examples of readymade aspects :
Transaction management & security.

Types of Advice --appear in Aspect class

@Before : This advice (cross cutting concern) logic gets Executed only before B.L method execution.
@AfterReturning Executes only after method returns in successful manner
@AfterThrowing - Executes only after method throws exception
@After -- Executes always after method execution(in case of success or failure)
@Around -- Most powerful, executes before & after.

Regarding pointcuts
Sometimes we have to use same Pointcut expression at multiple places, we can create an empty method with @Pointcut annotation and then use it as expression in advices.

eg of PointCut annotation syntax

```
@Before("execution (* com.app.bank.*.*(..))")

@Pointcut("execution (* com.app.bank.*.*(double))")

// point cut expression
@Pointcut("execution (* com.app.service.*.add*(..))")
        // point cut signature -- empty method .
        public void test() {
        }
```

eg of Applying point cut
1. @Before(value = "test()")

```
public void logBefore(JoinPoint p) {.........}

2. @Pointcut("within(com.app.service.*)")
    public void allMethodsPointcut(){}

@Before("allMethodsPointcut()")
    public void allServiceMethodsAdvice(){...}

3. @Before("execution(public void com.app.model..set*(*))")
 public void loggingAdvice(JoinPoint joinPoint){pre processing logic ....}

4.  //Advice arguments, will be applied to bean methods with single String argument
    @Before("args(name)")
    public void logStringArguments(String name){....}

5. //Pointcut to execute on all the methods of classes in a package
    @Pointcut("within(com.app.service.*)")
    public void allMethodsPointcut(){}

6.@Pointcut("execution(* com.core.app.service.*.*(..))") // expression
private void meth1() {}  // signature

7.@Pointcut("execution(* com.app.core.Student.getName(..))")
private void test() {}

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

Steps in AOP Implementation
1. Create core java project.
2. Add AOP jars to runtime classpath.
3. Add aop namespace to spring config xml.
4. To  Enable the use of the @AspectJ style of Spring AOP & automatic proxy
generation, add <aop:aspectj-autoproxy/>
5. Create Business object class. (using stereotype anotations)
6. Create Aspect class, annotated with @Aspect & @Component
7. Define one or more point cuts as per requirement
Eg of Point cut definition.
@PointCut("execution (* com.aop.service.Account.add*(..))")
public void test() {}
OR
@Before("execution (* com.aop.service.Account.add*(..))")
public void logIt()
{
   //logging advice code
}

Use such point cut to define suitable type of advice.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

Test the application.

execution --- exec of B.L method

eg : @Before("execution (* com.app.bank.*.*(..))")
public void logIt() {...}
Above tell SC ---- to intercept ---ANY B.L method ---
having ANY ret type, from ANY class from pkg -- com.app.bank
having ANY args
Before its execution.

Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (In around advice this is replaced by ProceedingJoinPoint, which is a subclass of JoinPoint.)

The org.aspectj.lang.JoinPointJoinPoint interface methods
1. Object[] getArgs() -- returns the method arguments.
2. Object getThis() --returns the proxy object
3 Object getTarget() --returns the target object
4. Signature getSignature() -- returns a description of the method that is being advised
5. String toString() -- description of the method being advised