**AOP Terminologies:**

**Pointcut**:- Advice means what and when but point cut mean **where**. Pointcut Means where we exexute the advice. Where out aspect will be run that is pointcut.

Example:

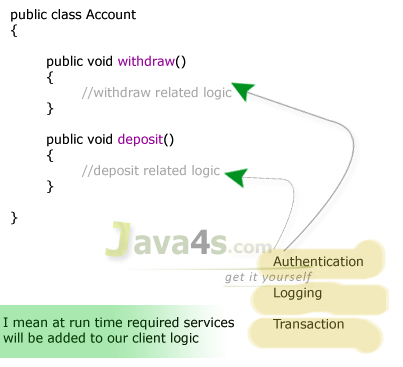
We have business of dry clean for that we need light. So we have electricity meter to count units of light.

**A brief Introduction of AOP terminology:-**

* ***Aspect***: A modularization of a concern that cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (**@AspectJ style**).
* ***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. Join point information is available in advice bodies by declaring a parameter of type **org.aspectj.lang.JoinPoint.**
* ***Advice*:** Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. Advice types are discussed below. Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the join point.
* ***Pointcut*:** A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP: Spring uses the AspectJ pointcut language by default.
* ***Introduction*:** (Also known as an inter-type declaration). Declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any proxied object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching.
* ***Target object*:** Object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* 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. *Proxy creation is transparent to users of the schema-based and @AspectJ styles of aspect declaration introduced in Spring 2.0.*
* ***Weaving*:** Linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

1. **Aspect:-**

* An aspect represent the cross-cutting functionality name, remember just name only.
* One real time service required for a business logic is called one Aspect.
* Aspect denotes only the cross-cutting functionality name not its implementation and all.



See here we are adding 3 services to our withdraw() and deposit() methods at run time.  So what is this ?  we have 3 Aspects here which are Authentication Aspect, Logging Aspect, Transaction Aspect/ May be Mailing Aspect in future.

Hope you understood right ? am again saying Aspect means just service name,  its implementation is irrelevant as of now.  That’s about Aspect in spring AOP.

Aspect is nothing some type of code ,class of code that we use in programs like logging aspect , transactional aspect, security aspect .

Aspect is some type of class which has non business logic. It contains a lots of methods that will be called.

1. **Advice:-**

We did see about Aspect already, Advice is the implementation of Aspect.  An Advice provides the code for implementation of the service. As an example consider logging  service, logging is an Aspect and Advice denotes the implementation of Log4j.

**Advice:-**Advice means **what** and **when** of Aspect( what we do and when will do

## Types of Advices

* Before Advice
* After Advice
* Throws Advice
* Around Advice
* After returning Advice

## Before Advice

* This advice contains implementation of the services which are need to be applied before business logic of method is going to execute.
* During compilation time the services will not be applied to our logic, services will apply only at run time.
* In order to create a Before advice, class should implement MethodBeforeAdvice interface.
* MethodBeforeAdvice interface is given in org.sp-fw.aop.\* package.
* If we implement MethodBeforeAdvice interface then we need to override a method called before() method.
* The services which are implemented in before() method are executed at before business logic.

## After Advice (After Returning Advice both are different)

* In the annotations After Advice and After Returning Advice are different, but here both are almost same
* This is also same as Before Advice, But this advice contains services which are applied after completion of our business method logic
* In order to create an after returning advice in spring, our class should implement an interface called AfterReturningAdvice, given in org.sp-fw.aop.\* package and we need to override a method given by this interface called afterReturning()

Brief intro of types of advice:-

1. ***Before advice***:- before the method ,called before method calling on which aspect apply
2. ***After (finally) advice***:- after method called , is called after invoking the method ,it is called before coming out come of method
3. ***After returning advice***: after returning is called. after returning if method is returning some thing means this is called after complete flow of execution(success fully) of that method on which this aspect apply. It is called after coming out come of method. Agr method return kr rha h to return krne ke baad call hoga
4. ***After throwing advice***: it is called if method throws exception.
5. ***Around:-*** it called before and after both. This is mixer of before and advice.
6. **@Aspect** declares the class as aspect.
7. **@Pointcut** declares the pointcut expression.

The annotations used to create advices are given below:

1. **@Before** declares the before advice. It is applied before calling the actual method.
2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
4. **@Around** declares the around advice. It is applied before and after calling the actual method.
5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.
6. **JoinPoint**

While creating the business logic of the method the additional services are needed to be injected (which we saw already) at different places or points, we call such points as join points.  At a join point a new services will be  added into the normal flow of a business method.

While executing the business method, the services are required at the following **3** places (generally), we call them as JoinPoints..

* Before business logic of the method starts
* After business logic of the method got completed
* If business logic throws an exception at run time

At the join point, an Aspect is injected, nothing but the implementation of Aspect i mean Advice will be injected.

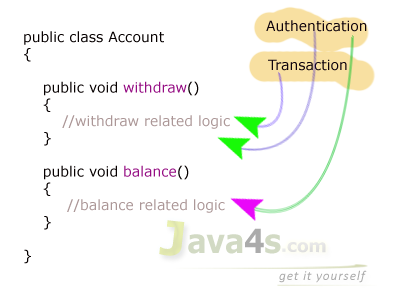
**“**We can use JoinPoint as parameter in the advice methods and using it get the method signature or the target object.

We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined. We can use [Generic objects](http://www.journaldev.com/1663/java-generics-tutorial-example-class-interface-methods-wildcards-and-much-more) also in the advice arguments**.”**

1. **Pointcut:-**

“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.”

Let us describe regarding spring AOP pointcut, for what methods what services need to be executed will be taken care by pointcut. A pointcut defines what advices are required at what join points.  In fact all business methods of a class doesn’t require all services.  So a pointcut informs to the IOC container that what business methods of a class needs what type of services.

[](http://www.java4s.com/wp-content/uploads/2012/08/withdraw-balance.png)

withdraw() method needs Authentication,Transaction services but balance() method needs Transaction service only.

We have 2 types of pointcuts in AOP

* Static Pointcut
* Dynamic Pointcut

Pointcut verifies whether a particular method of particular class is eligible for getting the advice or not. It means the pointcut verifies the class names and method names, but not run time parameters of the method.  in spring AOP we have 2 static pointcut classes

* NameMatchMethodPointcut
* RegxpMethodPointcut

Above 2 classes are predefined static pointcut classes given by spring AOP framework.  These classes are just going to verify whether the method name is matching with given condition or not, but these 2 classes doesn’t check whether the class names are matching or not. According to the predefined static point cuts, they wont verify the class name is matching or not but makes all classes eligible to get advices by default.

Let us see one by one static pointcut with example.

NameMatchMethodPointcut class is going to verify whether the method names of a spring bean class are matching with the given criteria or not.  While configuring this pointcut class into xml, we use mappedName or mappedNames property of the class into xml file. This class is a predefined class, so we need to configure the class directly into xml file like.

<bean id="id1" class="org.spfw.aop.support.NameMatchMethodPointcut">

    <property name="mappedName">

           <value>set\*</value>

    </property>

</bean>

Means the pointcut class identifies that setter methods of the bean class are eligible for getting advices.  If there is no commonalities in method names we can provide individual method names by configuring mappedNames property.

mappedNames is the collection type List :-), so we need <list>—-</list> element while configuring the collection, hmm some thing..

<bean id="id1" class="org.spfw.aop.support.NameMatchMethodPointcut">

    <property name="mappedNames">

           <list>

               <value>method1</value>

               <value>method2</value>

               <value>method3</value>

           </list>

    </property>

</bean>

According to above configuration,  method names with method1, method2, method3 are only eligible to get advices, we will see the example in the next article, stay tuned.

RegularExpressionMethodPointcut class is going to verify whether the method name of the class is matching with the given regular expression or not.  If matches then those methods are eligible to get Advices.  While configuring these classes into xml file, we are going to configure any one of following 2 properties..

* pattern
* patterns

some thing…

<bean id="id1" class="org.spfw.aop.support.RegularExpressionMethodPointcut">

    <property name="pattern">

       <value>get.\*N.+[0-9]</value>

    </property>

</bean>

According to above configuration, a method name should start with get, and between get and letter N there should be one or more characters and finally that should end with a digit.  If method name satisfies this pattern then it is eligible for getting advices.

While working with regular expressions, we have following indications..  
**.**  – matches for single character  
**.\*** – matches for zero [or] more characters  
**.+** – matches for 1 [or] more characters  
**.?** – matches for zero [or] 1 character  
**[A-Z]** – matches for an upper case alphabet  
**[abc]** – matches for either a or b or c  
**[0-9]** – matches for a digit

If we want to configure multiple regular expressions then we need to configure patterns property with list as NameMatchMethodPointcut, will see the example in the next article.

## **Understanding Pointcut**

Pointcut is an expression language of Spring AOP.

The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

1. @Pointcut("execution(\* Operation.\*(..))")
2. private void doSomething() {}

The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

#### **Understanding Pointcut Expressions**

Let's try the understand the pointcut expressions by the examples given below:

1. @Pointcut("execution(public \* \*(..))")

It will be applied on all the public methods.

1. @Pointcut("execution(public Operation.\*(..))")

It will be applied on all the public methods of Operation class.

1. @Pointcut("execution(\* Operation.\*(..))")

It will be applied on all the methods of Operation class.

1. @Pointcut("execution(public Employee.set\*(..))")

It will be applied on all the public setter methods of Employee class.

1. @Pointcut("execution(int Operation.\*(..))")

It will be applied on all the methods of Operation class that returns int value.

### Pointcuts and Wildcard Expressions

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. Spring AOP only supports method execution join points for Spring beans, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines exactly which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the @Pointcut annotation (the method serving as the pointcut signature must have a void return type).  
  
An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named 'anyOldTransfer' that will match the execution of any method named 'transfer':

@Pointcut("execution(\* transfer(..))")// the pointcut expression

private void anyOldTransfer() {}// the pointcut signature

# Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

* ***execution*** - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
* ***within*** - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
* ***this*** - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
* ***target*** - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
* ***args*** - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
* ***@target*** - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
* ***@args*** - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
* ***@within*** - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
* ***@annotation*** - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

# **Combining pointcut expressions**

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: anyPublicOperation (which matches if a method execution join point represents the execution of any public method); inTrading (which matches if a method execution is in the trading module), and tradingOperation (which matches if a method execution represents any public method in the trading module).

1. @Pointcut("execution(public \* \*(..))")
2. private void anyPublicOperation() {}
4. @Pointcut("within(com.xyz.someapp.trading..\*)")
5. private void inTrading() {}
7. @Pointcut("anyPublicOperation() && inTrading()")
8. private void tradingOperation() {}

# **Sharing common pointcut definitions**

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class SystemArchitecture {

/\*\*

\* A join point is in the web layer if the method is defined

\* in a type in the com.xyz.someapp.web package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.web..\*)")

public void inWebLayer() {}

/\*\*

\* A join point is in the service layer if the method is defined

\* in a type in the com.xyz.someapp.service package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.service..\*)")

public void inServiceLayer() {}

/\*\*

\* A join point is in the data access layer if the method is defined

\* in a type in the com.xyz.someapp.dao package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.dao..\*)")

public void inDataAccessLayer() {}

/\*\*

\* A business service is the execution of any method defined on a service

\* interface. This definition assumes that interfaces are placed in the

\* "service" package, and that implementation types are in sub-packages.

\*

\* If you group service interfaces by functional area (for example,

\* in packages com.xyz.someapp.abc.service and com.xyz.def.service) then

\* the pointcut expression "execution(\* com.xyz.someapp..service.\*.\*(..))"

\* could be used instead.

\*

\* Alternatively, you can write the expression using the 'bean'

\* PCD, like so "bean(\*Service)". (This assumes that you have

\* named your Spring service beans in a consistent fashion.)

\*/

@Pointcut("execution(\* com.xyz.someapp.service.\*.\*(..))")

public void businessService() {}

/\*\*

\* A data access operation is the execution of any method defined on a

\* dao interface. This definition assumes that interfaces are placed in the

\* "dao" package, and that implementation types are in sub-packages.

\*/

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

public void dataAccessOperation() {}

}

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(public \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

*'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.*

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

*'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

*'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.*

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

*'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

*'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

*'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.*

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

*'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.*

* any join point (method execution only in Spring AOP) on a Spring bean named 'tradeService':

bean(tradeService)

* any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression '\*Service':

bean(\*Service)