Spring AOP enables Aspect-Oriented Programming in spring applications.

**Why AOP?**

AOP is used in applications that have cross-cutting concerns, that is, pieces of logic or code that are written in multiple classes/layers as per the requirements.

Common examples:

* Transaction Management
* Logging
* Exception Handling (especially when you may want to have detailed traces or have some plan of recovering from exceptions)
* Security aspects
* Instrumentation

## Concerns

## A concern is a piece of code that performs a specific task.

## There are two types of concerns:

1) “Core” concerns are codes used for business logic

2) “Cross concerns” are functions that are conceptually separate from the application's business logic but affect the entire service layer.

Example: logging, auditing, declarative transactions, security, caching.

AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well. You can add/remove concerns without recompiling complete source code simply by changing configuration files.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns.

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.

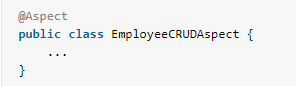
**AOP Concepts and Terminology**

AOP concepts and terminologies are as follows:

* Join point
* Advice
* Pointcut
* Introduction
* Target Object
* Aspect
* Interceptor
* AOP Proxy
* Weaving

#### **Aspect**

It is a class that contains advices, join points etc. To create an aspect, you need to apply @Aspect annotation on your aspect class and register it in **applicationContext.xml**

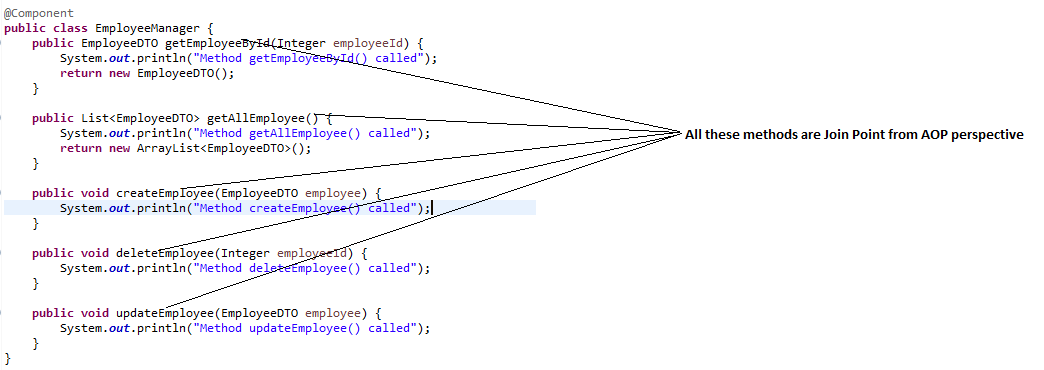




#### **Join point**

Join point is a point of execution of the 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.

All the methods defined inside EmployeeManager are joint points.



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

Example of a Logging aspect class



**Fig-1**

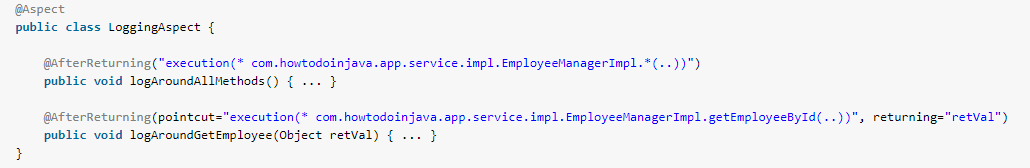
#### **Advice**

Advice represents an action taken by an aspect at a join point. In Fig1 all logBefore and logAfter methods are advice methods

Advice is associated with a pointcut expression and runs at any join point matched by the pointcut.

There are different types of advices:

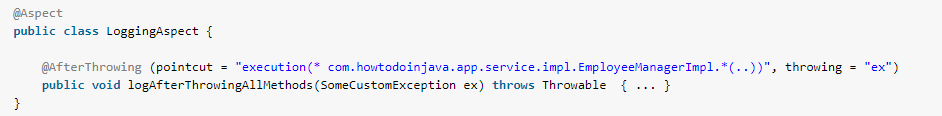
* **Before Advice:** it executes before a join point.
* **After Returning Advice:** it executes after a joint point completes normally. for example, if a method returns without throwing an exception.



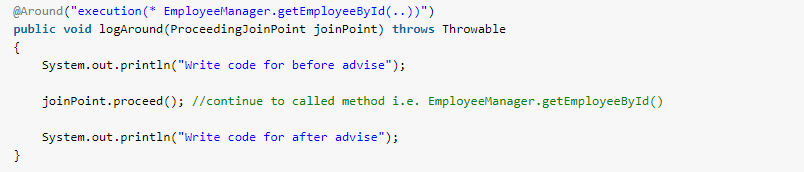
1. Sometimes you need access the actual return value – that was returned from method, you can get that return value as shown using returning attribute inside @AfterReturning annotation.
2. The name used in the returning attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value.
3. Please note that any returning clause also restricts **matching to only those method executions that return a value** of the specified type (Object or it’s subtypes in this case, which will match any return value).

* **After Throwing Advice:** it executes if method exits by throwing an exception.

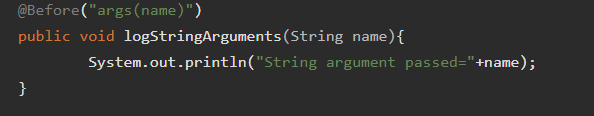
Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the throwing attribute to both restrict matching (if desired, use Throwable as the exception type otherwise) and bind the thrown exception to an advice parameter.



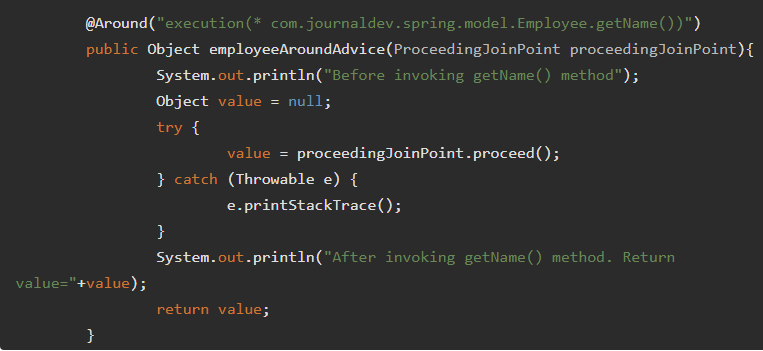
* **After (finally) Advice:** it executes after a join point regardless of join point exit whether normally or exceptional return.
* **Around Advice:** Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. An around advice can be written as below.



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**](https://www.journaldev.com/1663/java-generics-example-method-class-interface) also in the advice arguments.



Around advice are always required to have **ProceedingJoinPoint** as argument and we should use it’s proceed () method to invoke the target object advised method. If advised method is returning something, it’s advice responsibility to return it to the caller program. For void methods, advice method can return null. Since around advice cut around the advised method, we can control the input and output of the method as well as it’s execution behavior.



**Pointcut**

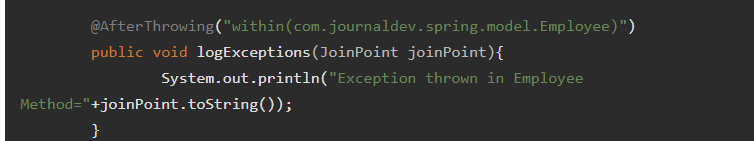
A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Expressions passed in **@Before** and **@After** annotations (i.e. “execution(\* EmployeeManager.getEmployeeById(..))“) are pointcuts (Pointcut expression).

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.

We can use asterisk (\*) as wild card in Pointcut expressions, **logBefore2**() will be applied for all the methods of EmployeeManager class.\

We can use within in pointcut expression to apply advice to all the methods in the class. We can use @AfterReturning advice to get the object returned by the advised method.  
We have throwException() method in the Employee bean to showcase the use of After Throwing advice.



**Weaving**

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

**AOP Proxy**

An object created by the AOP framework 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.

**Interceptor**

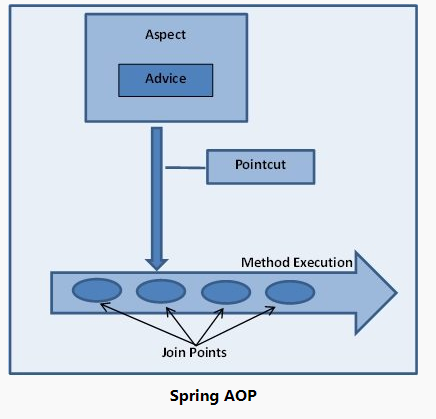
It is an aspect that contains only one advice.

**AOP Implementations**

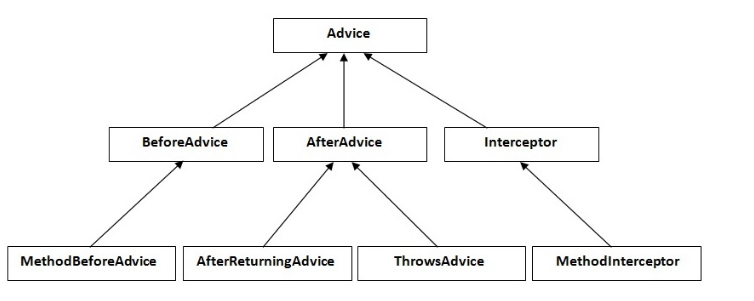
AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

AspectJ has grown into a complete and popular AOP framework, Spring supports the use of POJO aspects written with [**AspectJ**](https://eclipse.org/aspectj/) annotations in its AOP framework. Since AspectJ annotations are supported by more and more AOP frameworks

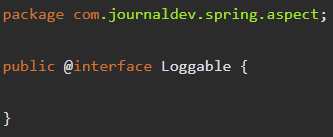


Hierarchy of Advice:

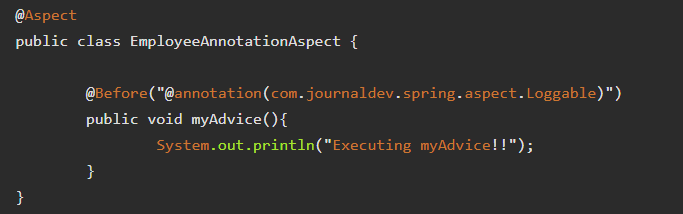


### **Spring Advice with Custom Annotation Pointcut**

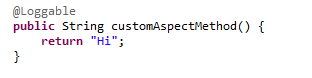
1. An alternative approach is to create a custom annotation and annotate the methods where we want the advice to be applied.
2. Spring Framework @Transactional annotation is a great example of this approach.
3. Create a Loggable interface:



1. Create an Aspect class with @Before advice and pass Loggable interface



1. Annotate the method with @Loggable so that myAdvice will be applied.



When this method is called then “Executing customized method!!” will be printed.

# **Spring AOP AspectJ pointcut expression examples**

#### **Match all methods within a class in another package**

execution (\* com.howtodoinjava.EmployeeManager.\*(..))

The above pointcut expression matches all the methods declared in the EmployeeManager interface. The preceding wildcard matches methods with any modifier (public, protected, and private) and any return type. The two dots in the argument list match any number of arguments.

#### **Match all methods within a class within same package**

execution (\* EmployeeManager.\*(..))

1. **Match all public methods in EmployeeManager**

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

1. **Match all public methods in EmployeeManager with return type EmployeeDTO**

execution (public EmployeeDTO EmployeeManager.\*(..))

1. **Match all public methods in EmployeeManager with return type EmployeeDTO and first parameter as EmployeeDTO**

execution (public EmployeeDTO EmployeeManager.\*(EmployeeDTO, ..))

#### **Match all methods defined in classes inside package com.howtodoinjava**

within(com.howtodoinjava.\*)

#### **Match all methods defined in classes inside package com.howtodoinjava and classes inside all sub-packages as well**

sub-packages use two dots.

|  |
| --- |
| within(com.howtodoinjava..\*) |

# **Spring - Declarative Transaction Management with @Transactional Annotation**

**@Transactional** provides annotation-based declarative transaction support which is similar to EJB container-managed transaction. With this annotation, we can specify transaction behavior to individual methods without coupling business logic with transaction code.

**EnableTransactionManagement** annotation enables Spring's annotation-driven transaction management.

**checked** exception transactions do not rollback implicitly unless we specify **'rollbackFor'** attribute of @**Transactional**.

There are 7 types of propagation supported by Spring:

* **PROPAGATION\_REQUIRED** – Support a current transaction; create a new one if none exists.
* **PROPAGATION\_SUPPORTS** – Support a current transaction; execute non-transactionally if none exists.
* **PROPAGATION\_MANDATORY** – Support a current transaction; throw an exception if no current transaction exists.
* **PROPAGATION\_REQUIRES\_NEW** – Create a new transaction, suspending the current transaction if one exists.
* **PROPAGATION\_NOT\_SUPPORTED** – Do not support a current transaction; rather always execute non-transactionally.
* **PROPAGATION\_NEVER** – Do not support a current transaction; throw an exception if a current transaction exists.
* **PROPAGATION\_NESTED** – Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

In most cases, you may just need to use the PROPAGATION\_REQUIRED.

**REQUIRED behavior**

Spring **REQUIRED** behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the Spring container will create a new one. If multiple methods configured as **REQUIRED** behavior are called in a nested way they will be assigned **distinct logical transactions,** but they will all share the **same physical transaction**. In short this means that if an inner method causes a transaction to rollback, the outer method will fail to commit and will also rollback the transaction. Let's see an example:

**Outer bean**

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequired();

} catch(RuntimeException e){

// handle exception

}

}

**Inner bean**

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired() {

throw new RuntimeException("Rollback this transaction!");

}

Note that the inner method throws a **RuntimeException** and is annotated with **REQUIRED** behavior. This means that it will use the **same** transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

**Note-1:**The only exceptions that set a transaction to rollback state by default are the unchecked exceptions (like **RuntimeException**). If you want checked exceptions to also set transactions to rollback you must configure them to do so.

**Note-2:**When using declarative transactions, i.e. by using only annotations, and calling methods from the same bean directly (self-invocation), the **@Transactional** annotation will be ignored by the container. If you want to enable transaction management in self-invocations, you must configure the transactions using **aspectj**.

**REQUIRES\_NEW behavior**

**REQUIRES\_NEW** behavior means that a new physical transaction will always be created by the container. In other words, the inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result: they will run in **distinct physical transactions**.

**Outer bean**

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequiresNew(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequiresNew();

} catch(RuntimeException e){

// handle exception

}

}

**Inner bean**

@Override

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void testRequiresNew() {

throw new RuntimeException("Rollback this transaction!");

}

The inner method is annotated with **REQUIRES\_NEW** and throws a **RuntimeException** so it will set its transaction to rollback but will **not** affect the outer transaction. The outer transaction is paused when the inner transaction starts and then resumes after the inner transaction is concluded. They run independently of each other, so the outer transaction may commit successfully.