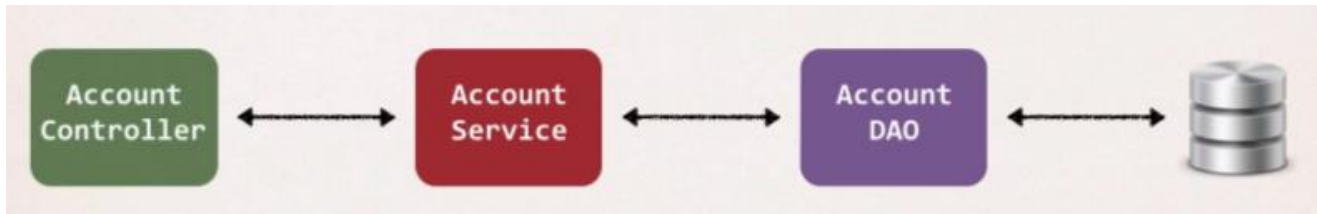


Section 34: AOP: Aspect-Oriented Programming Overview

3 / 3 | 21min

➤ Overview:

Application Architecture: **Service layer design pattern**



We will use this to apply some AOP concept.

Consider this code for Data Access Object (DAO)

```
public void addAccount(Account theAccount, String userId) {  
    Session currentSession = sessionFactory.getCurrentSession();  
    currentSession.save(theAccount);  
}
```

Now There is some new Requirement comes **from the Boss**:

- Need to add logging to our DAO Methods.
- Add some logging statements before the start of the method.
- Possibly more places, but get started on that ASAP!

So, we have added the logging message using Sys.println or log4j, etc.

```
public void addAccount(Account theAccount, String userId) {  
    // add code for Logging  
  
    Session currentSession = sessionFactory.getCurrentSession();  
    currentSession.save(theAccount);  
}
```

But **the BOSS** come up with **one more requirement** that we need to add some security code tour DAO and Make sure user is authorized before running the DAO method.

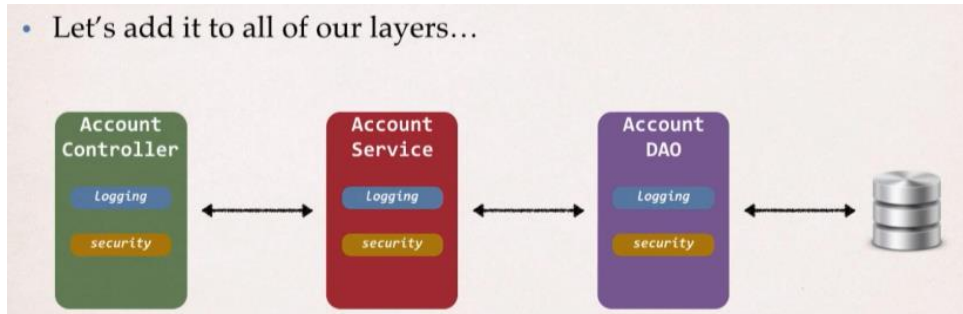
```

public void addAccount(Account theAccount, String userId) {
    // add code for logging
    // add code for security check

    Session currentSession = sessionFactory.getCurrentSession();
    currentSession.save(theAccount);
}

```

But we need to add them to all the layers, i.e. in Service layer, DAO layer, controller layers.



Now what next can we expect; replicate the work in whole system i.e. to all other 100 classes.

Two major problems we have here:

1. Code tangling

- For a given method: addAccount(...)
- We have logging and security code tangled in

2. Code Scattering

- If we need to change the logging or security code
- We have to update All Classes. i.e., may be in 1000 classes.

Other possible solutions?

1. Inheritance?

- Every class would need to inherit from a base class
- Can all classes extend from your base class? ... plus no multiple inheritance.

2. Delegation?

- Classes would delegate logging, security calls
- Still would need to update classes for that support if we wanted to
 - Add/remove logging or security
 - Add new Features like auditing, API management, Instrumentation.

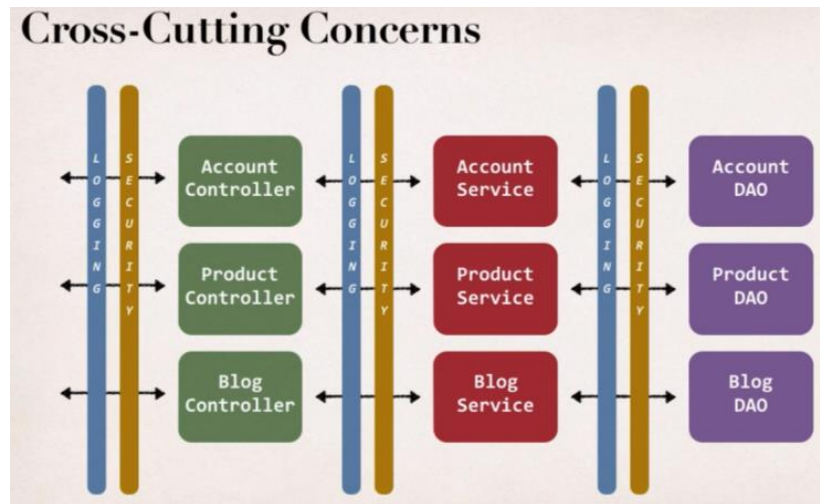
So, we didn't solve the original problem.

Aspect Oriented Programming comes into Rescue.

Aspect Oriented Programming:

- Programming technique based on concept of an aspect.
- Aspect encapsulates cross-cutting logic.

Cross-Cutting Concerns: This is a buzz word that we are going to hear everywhere when it comes to aspect-oriented programming. **Concerns** basically means logic or functionality i.e., a basic infrastructure code that all application needs.



Therefore, we encapsulate the common logic and then apply accordingly on the different areas in our application.

Aspects:

- Aspects can be reused at multiple locations.
- An Aspect is just a class, and the same aspect can be applied to different areas of my application based on the configuration.

Like In the configuration we can say apply the logging Aspects to the Account Controller class, or in Account Service class, or in Account DAO class.

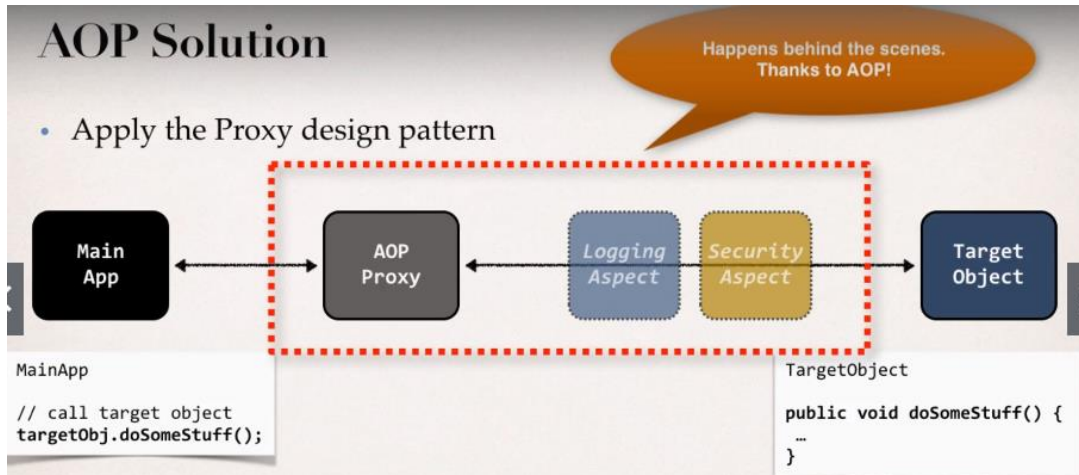


AOP Solution:

Apply the Proxy design pattern.

So, we have our Main App and the target object and calling the method `doSomeStuff()`; now the `main()` app has no idea of any AOP or aspects and it will simply gonna make a method call, and its very similar to make a phone call to our friend where we pick up the phone and call simply but behind the scene we can think like it's being monitored and similarly we can think like logging aspect and

Security aspect is listening our conversation and they can take any action based on what we say or pass, so it's can be think like some Spies.



Benefits of AOP:

1. Code for Aspect is defined in a single class.

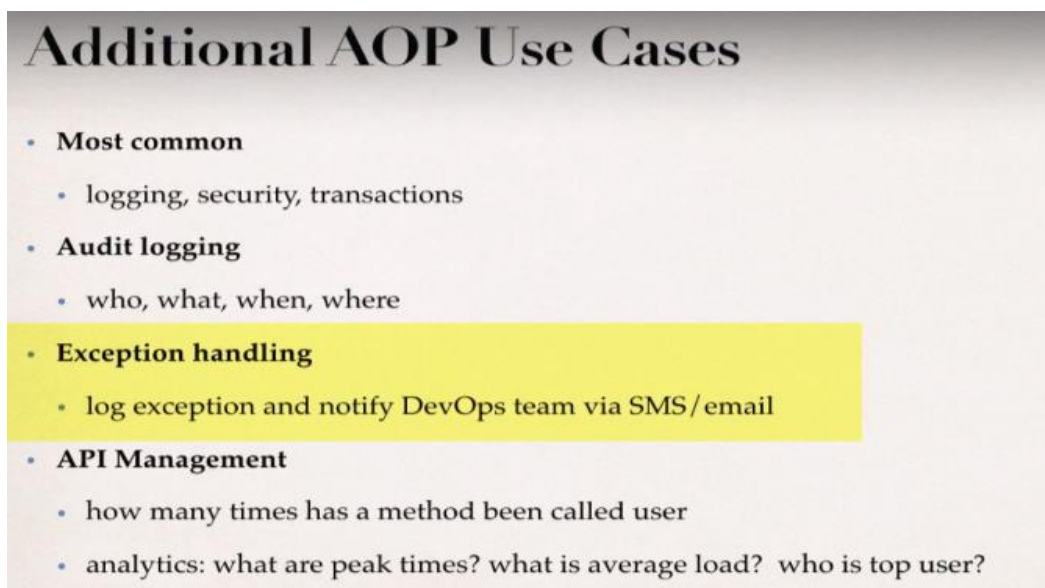
- Much better than being scattered everywhere.
- Promotes code reuse and easier to change.

2. Business code in your application is cleaner.

- Only applied to business functionality: addAccount
- Reduce code complexity.

3. Configurable

- Base on configuration, apply Aspects selectively to different parts of app
- No need to make changes to main Application code ...very important!



Disadvantages:

- To many aspects and app flow is hard to follow.
Consider the case where there is project of 20 developer and each creating their own set of 20 to 30 aspects then applying to the system, then it's really being hard to follow what's being called.
- Minor performance cost for aspect execution (run-time Weaving).

So, we don't want to overdo it, as If we have small number of aspects, we may not even notice the performance cost it can be very expensive on execution of operations in case there is lots of aspects.

➤ Comparing AOP And AspectJ

AOP terminology:

Aspect: Module of code for a cross-cutting concern (logging, security,)

Advice: What action is taken and when it should be applied.

Join Point: when to apply code during program execution.

Pointcut: A predicate expression for where advice should be applied.

Advice type

- **Before Advice:** run the code before the actual method executes.
- **After Finally advice:** run the code after the method finishes (like finally block of a try catch).
- **After returning advice:** run the code after the method for a successful execution.
- **After Throwing advice:** run the code after method (if exception thrown)
- **Around Advice:** run the code before and after the method.

- **Weaving:** Connecting aspects to target objects to create an advised object.
 - **There are different types of weaving:** Compile-time, load-time or run-time weaving.
 - **Regarding Performances:** Run time weaving is the slowest.

AOP frameworks

There are two leading AOP Frameworks for Java

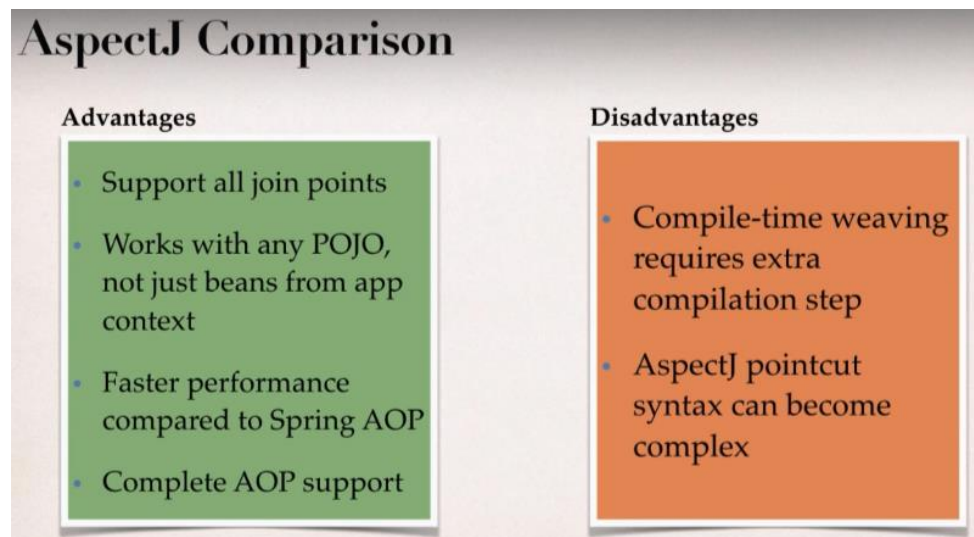
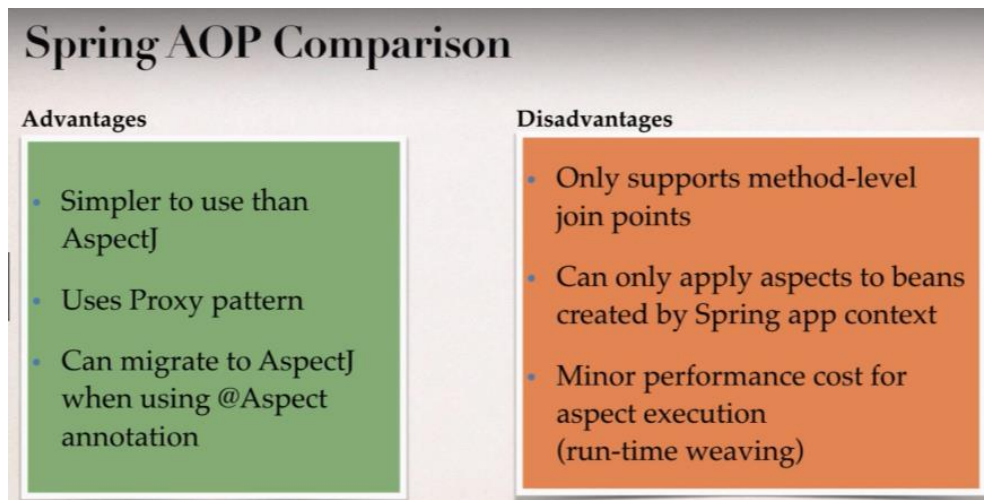
1. Spring AOP
2. AspectJ

Spring AOP

- Spring provides AOP Support and it uses AOP in the background.
 - Key Components of Spring where it uses AOP: Security, transactions, caching etc.
- Spring makes use of runtime weaving of aspects, so, spring makes use of proxy pattern to advice an object so the main application will talk to the proxy and then the aspects are processed and then it makes to the actual objects.

AspectJ

- Original AOP framework, released in 2001.
- <https://www.eclipse.org/aspectj>
- It provides complete support for AOP.
- Rich support for
 - **Join Points:** method-level, constructor, field
 - **Code weaving:** compile-time, post compile-time and load-time



- In short, AspectJ is very fast but it involves lots of complexity with it.
- And Spring AOP is a light implementation of AOP but it solves common problems in enterprise application.

Our Spring AOP Roadmap

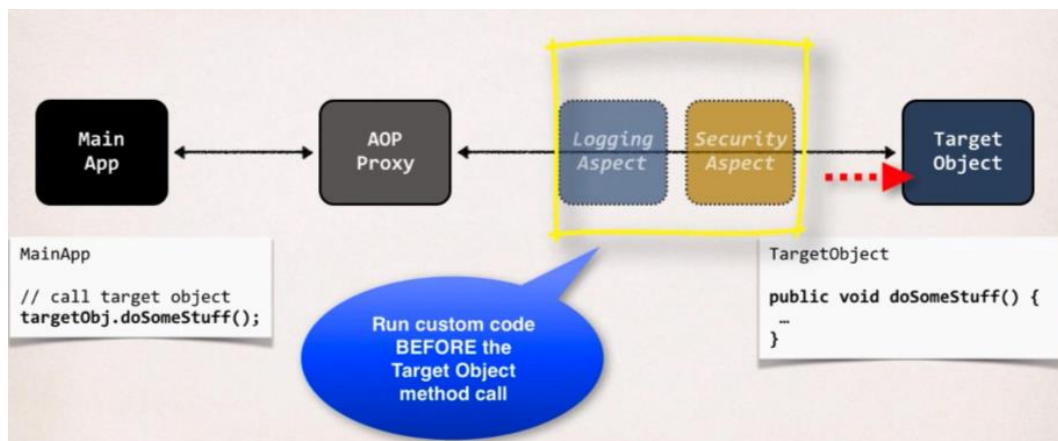
Step-By-Step

- Create **Aspects**
- Develop **Advices**
 - Before, After returning, After throwing,
 - After finally, Around
- Create **Pointcut** expressions
- Apply it to our big CRM project (Spring MVC + Hibernate)

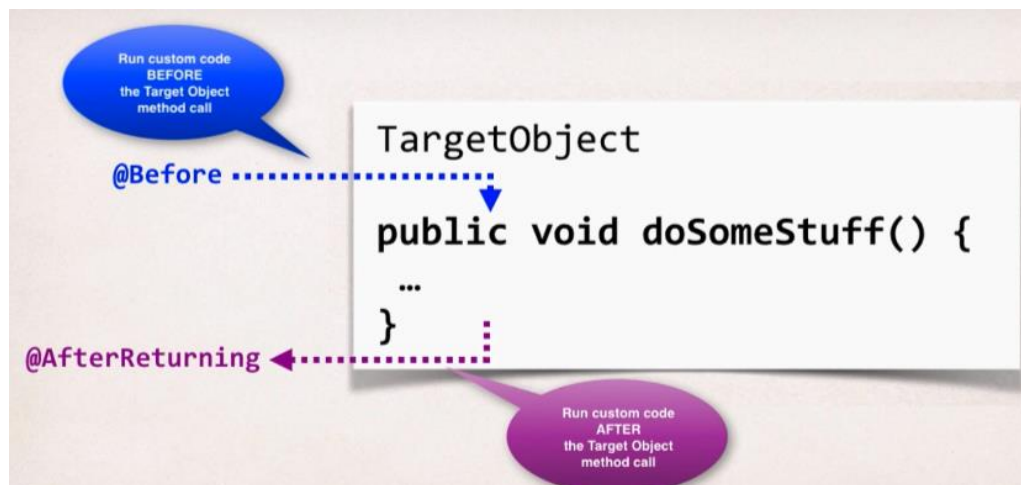
Section 35: AOP: @Before Advice Type

0 / 4 | 34min

Example that we are going to discuss; Here we want our custom code to execute before the target method execution.



Advice-Interaction:

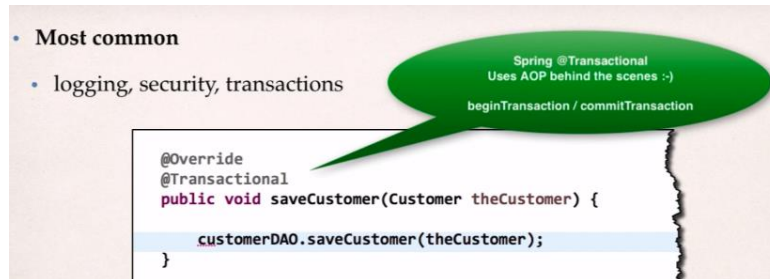


@BeforeAdvice – Use cases:

Most Common:

- Logging, security, transaction.

And @transactional actually uses AOP behind the scene for begin transaction and session creation, session closing and committing the transaction.



- Audit logging

So, when the method gets called then we can log who, what, when, where; may be we want to keep that information so we can use Audit logging for that.

- API Management

How many times has a method been called and from there we can use some data analytics like what are the peak times? What is the average load, who is top user?

In the coding example that we are going to discuss here:

Prerequisite: Adding AspectJ JAR file

- Need to download AspectJ Jar file.
- Even though we are using Spring AOP still we need AspectJ JAR file.
- Why?
 - Spring AOP uses some of the AspectJ annotations.
 - Spring AOP uses some of the AspectJ classes.

Spring AOP is lightweight AOP implementation.

Development Process:

1. Create target object: AccountDAO.

```
@Component
public class AccountDAO {

    public void addAccount() {

        System.out.println("DOING MY DB WORK: ADDING AN ACCOUNT");

    }
}
```


2. Create Spring Java Config class.

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan("com.luv2code.aopdemo")
public class DemoConfig {
}
```

Spring AOP
Proxy Support

This `@EnableAspectJAutoProxy` will allow us to make proxy in the background using Spring AOP with support from AspectJ.

3. Create main app.

```
public class MainDemoApp {

    public static void main(String[] args) {

        // read spring config java class
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(DemoConfig.class);

        // get the bean from spring container
        AccountDAO theDAO = context.getBean("accountDAO", AccountDAO.class);

        // call the business method
        theDAO.addAccount();

        // close the context
        context.close();
    }
}
```

4. Create Aspect with @Before advice.

We will create a normal class with `@Component` annotation and further we will add up `@Aspect` to mention that it's a special aspect class and in the result this class can run as our spy network and they can listen in on communication behind the scene.

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    @Before("execution(public void addAccount())")
    public void beforeAddAccountAdvice() {

        ...
    }
}
```

Pointcut
expression

Run this code BEFORE - target object method: "public void addAccount()"

We can give any method name we want and then inside it we can provide any custom code for logging or security. And the given code will execute before the given target object method present in pointcut expression.

```

@Aspect
@Component
public class MyDemoLoggingAspect {


    @Before("execution(public void addAccount())")
    public void beforeAddAccountAdvice() {

        System.out.println("Executing @Before advice on addAccount()");

    }

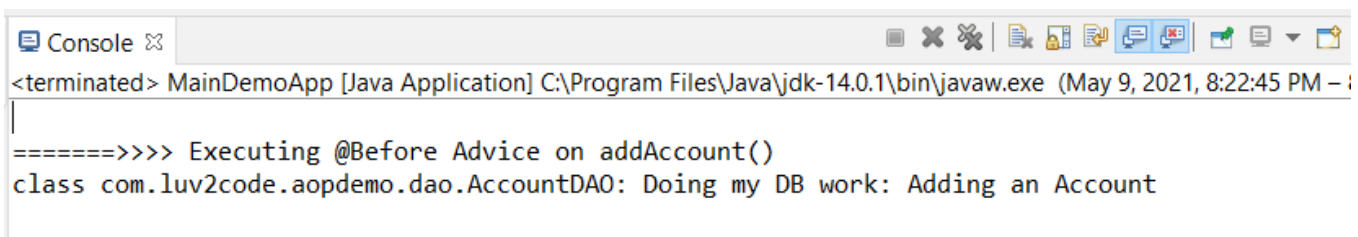
}

```



Best Practice: Aspect and Advices

- Keep the code small
 - Keep the code fast
 - Do not perform any expensive/slow operations
 - Get in and out as QUICKLY as possible.
- To download the AspectJ Jar file visit:
<https://mvnrepository.com/artifact/org.aspectj/aspectjweaver>
 - **Output** associated with our main() app:



```

<terminated> MainDemoApp [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 9, 2021, 8:22:45 PM - )
=====>>> Executing @Before Advice on addAccount()
class com.luv2code.aopdemo.dao.AccountDAO: Doing my DB work: Adding an Account

```

❖ What is Pointcut Expression? → "@Before("execution(public void addAccount())")"

So, basically it will tell Spring AOP system if the given conditions met then apply the below advice code.

Spring AOP actually uses the AspectJ's pointcut expression Language. AspectJ has lots of pointcut, but we will simply start with execution pointcut.

Pointcut Expression Language

We will use this to define the predicate to actually match on a given method.

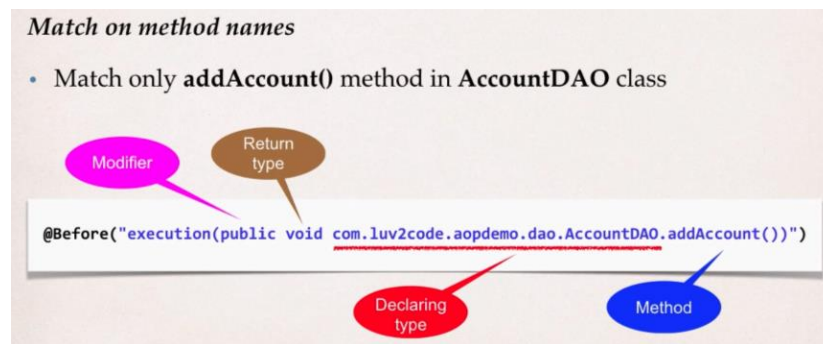
```

execution(modifiers-pattern? return-type-pattern declaring-type-pattern?
          method-name-pattern(param-pattern) throws-pattern?)

```

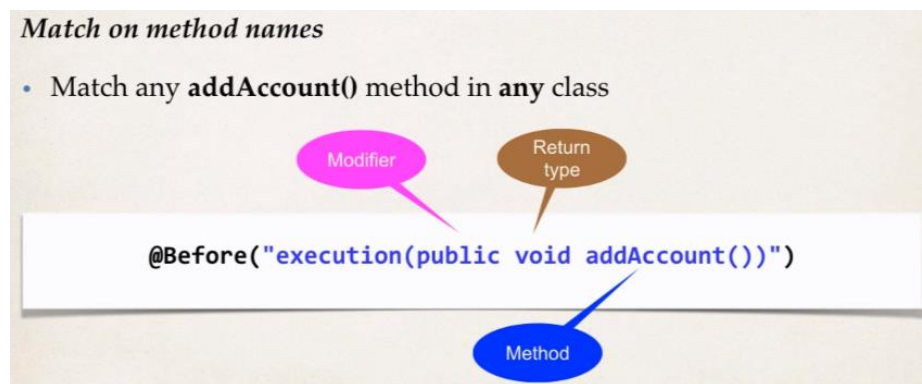
- ? represents optional pattern, which means we don't need to give those patterns if we don't need them. And Pattern can also make use of wildcards: ***(matches on everything)**
- Modifiers-pattern: where Spring AOP only supports **Public**
- Return-type-pattern: which means is it returning void, Boolean, String, List<Customer>, etc.
- Declaring-type-pattern: what is the class name of the type that we are going to use for this given method.
- Method-name-pattern: we can give the actual name of the method or wildcards.
- Param-pattern: we can match a method for a given parameter list.
- Throws-pattern: it matches to a method that throw the given exception.

Example 1:



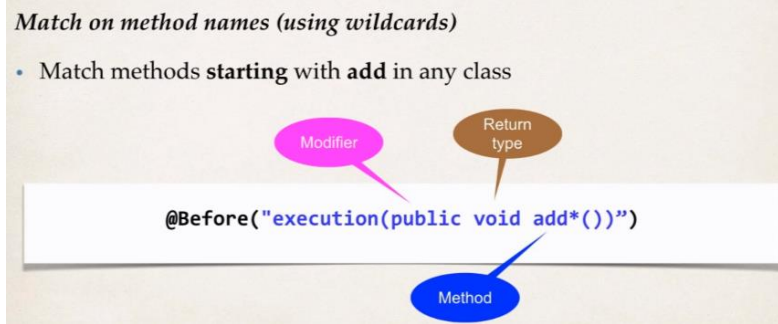
In the above example, any calls to the `addAccount()` method in the `AccountDAO` class will execute our given advice that we have setup for AOP, and this is all for matching on execution calls on given method.

Example 2:



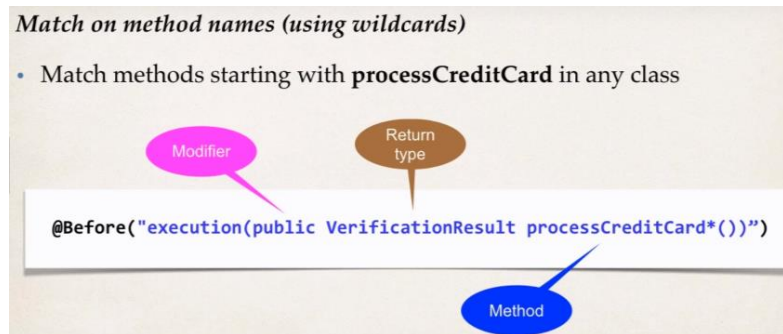
Any call to `addAccount()` method in any class, and this is where our spy network will jump and it will get to know that `main()` method is calling `addAccount()` then it will decide to apply the advice by running our AOP code then the actual method will get executed.

Example 3:



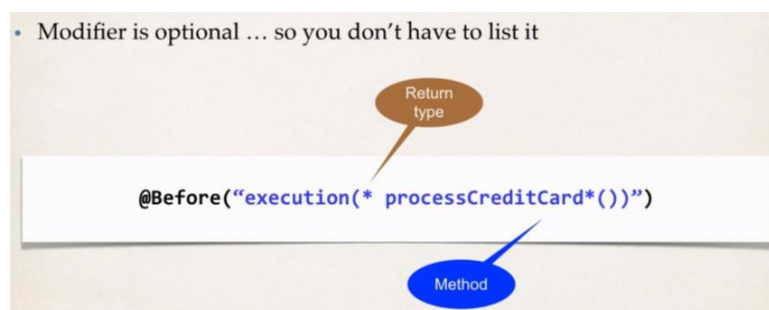
Here we are trying to match to any method starting from **add** and belongs to any class. Here Spring AOP will see the wildcard (*) and it will check for any method starting from **add** then we will apply our advice.

Example 4:



So, any method whose return type is VerificationResult and having access modifier public and method name starts with processCreditCard then Spring AOP will apply our advice.

Example 5:



As we know, modifier is optional and it's not mandatory to mention it but return type is mandatory and here we are accepting method to have any type of modifier and is method name starts with processCreditCard the Spring AOP will apply our advice on that method.

Use Case: Now let add another class having the same method: addAccount() and will keep the pointcut expression same we had.

File: MembershipDAO.java

```
@Component
public class MembershipDAO {

    public void addAccount() {
        System.out.println(getClass()+" Doing Stuff: Adding a membership account");
    }

}
```

File: MyDemoLoggingAspect.java

Keeping the aspect class as it is.

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    //this is where we will add all of our related advices for logging

    //let's start with @Before advice

    //@Before("execution(public void addAccount())")

    @Before("execution(public void addAccount())")
    public void beforeAddAccountAdvice() {

        System.out.println("\n=====>>> Executing @Before Advice on addAccount()");
    }

}
```

File: MainDemoApp.java

```
public class MainDemoApp {

    public static void main(String[] args) {

        //read spring config java class
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(DemoConfig.class);

        //get the bean from spring container
        AccountDAO theAccountDAO = context.getBean("accountDAO",AccountDAO.class);
        MembershipDAO theMembershipDAO = context.getBean("membershipDAO",
            MembershipDAO.class);

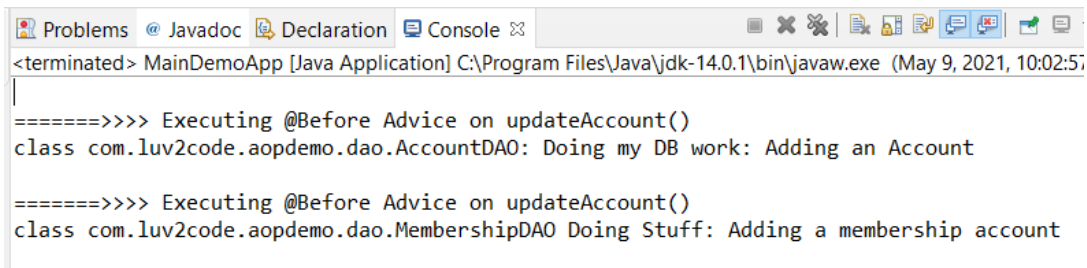
        //call the business method
        theAccountDAO.addAccount();
        theMembershipDAO.addAccount();

        //close the context
        context.close();
    }

}
```

So we are calling the addAccount() method and in result the aspect class will apply our advice, and later the addAccount() method will execute.

Output:



```
Problems @ Javadoc Declaration Console
<terminated> MainDemoApp [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 9, 2021, 10:02:57)

=====>>> Executing @Before Advice on updateAccount()
class com.luv2code.aopdemo.dao.AccountDAO: Doing my DB work: Adding an Account

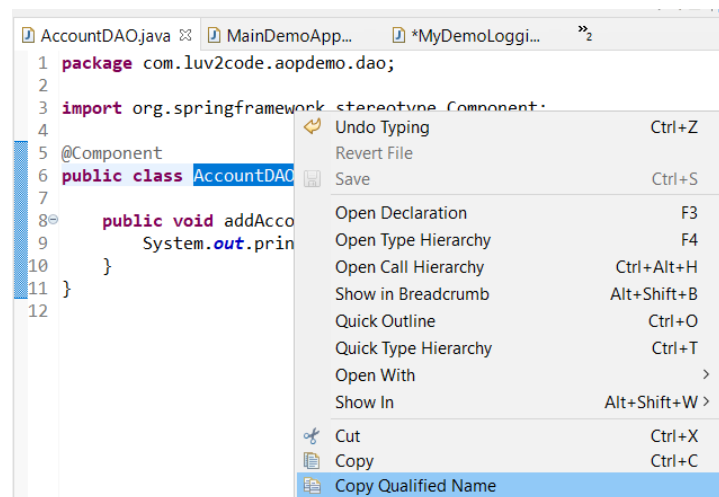
=====>>> Executing @Before Advice on updateAccount()
class com.luv2code.aopdemo.dao.MembershipDAO Doing Stuff: Adding a membership account
```

Use Case: Now this time we just want to call the `addAccount()` method of `Account` class but not the `Membership` class, so we need to update the `pointcut` expression accordingly.

Actually, we need to be super specific in this example.

We need to enter the qualified class name then only, the `addAccount()` method belong to that class will get applied by the advice mentioned in the aspect class.

✓ To copy the Qualified class name: eclipse shortcut:



File: `MyDemoLoggingAspect.java`

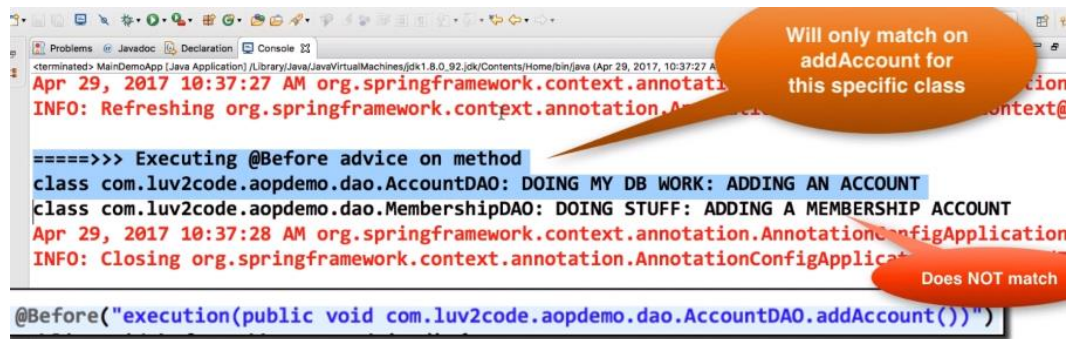
Keeping the aspect class as it is.

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    //this is where we will add all of our related advices for logging
    //let's start with @Before advice
    //@Before("execution(public void addAccount())")

    @Before("execution(public void com.luv2code.aopdemo.dao.AccountDAO.addAccount())")
    public void beforeAddAccountAdvice() {
        System.out.println("\n=====>>> Executing @Before Advice on addAccount()");
    }
}
```

Output:



```
<terminated> MainDemoApp [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Apr 29, 2017, 10:37:27 AM)
Apr 29, 2017 10:37:27 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext
====>>> Executing @Before advice on method
class com.luv2code.aopdemo.dao.AccountDAO: DOING MY DB WORK: ADDING AN ACCOUNT
class com.luv2code.aopdemo.dao.MembershipDAO: DOING STUFF: ADDING A MEMBERSHIP ACCOUNT
Apr 29, 2017 10:37:28 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext
@Before("execution(public void com.luv2code.aopdemo.dao.AccountDAO.addAccount())")
```

Match Method starting with “add” in any class:

```
@Before("execution(public void add*())")
```

File: MyDemoLoggingAspect.java

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    //this is where we will add all of our related advices for logging
    //let's start with @Before advice
    //@Before("execution(public void addAccount())")

    @Before("execution(public void add*())")
    public void beforeAddAccountAdvice() {
        System.out.println("\n====>>> Executing @Before Advice on addAccount()");
    }
}
```

File: MainDemoApp.java

```
public class MainDemoApp {

    public static void main(String[] args) {

        //read spring config java class
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(DemoConfig.class);

        //get the bean from spring container
        AccountDAO theAccountDAO = context.getBean("accountDAO", AccountDAO.class);
        MembershipDAO theMembershipDAO = context.getBean("membershipDAO",
            MembershipDAO.class);

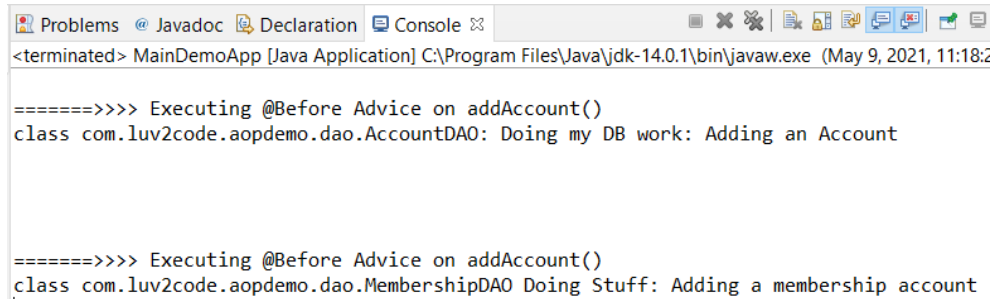
        //call the business method
        theAccountDAO.addAccount();
        System.out.println("\n\n");
        theMembershipDAO.addSillyMember();
        //close the context
        context.close();
    }
}
```

File: MembershipDAO.java

```
@Component
public class MembershipDAO {

    public void addSillyMember() {
        System.out.println(getClass()+" Doing Stuff: Adding a membership
        account");
    }
}
```

Output:



```
<terminated> MainDemoApp [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 9, 2021, 11:18:2
=====>>> Executing @Before Advice on addAccount()
class com.luv2code.aopdemo.dao.AccountDAO: Doing my DB work: Adding an Account

=====>>> Executing @Before Advice on addAccount()
class com.luv2code.aopdemo.dao.MembershipDAO Doing Stuff: Adding a membership account
```

Method Based On Return Type:

```
@Before("execution(void add*())")
```

Here, advice will get applied to all the methods start with “add” and whose return type is void. If return type not matches then advice will not be applied.

We will change the return type of addSillyMember() member in MembershipDAO class.

File: MembershipDAO.java

```
@Component
public class MembershipDAO {

    public boolean addSillyMember() {
        System.out.println(getClass()+" Doing Stuff: Adding a membership
        account");
        return true;
    }
}
```

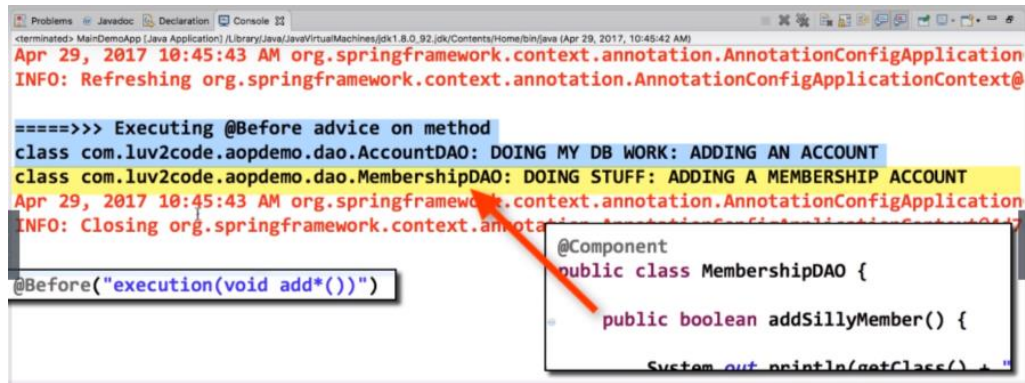
File: MyDemoLoggingAspect.java

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    //this is where we will add all of our related advices for logging
    //let's start with @Before advice
    //@Before("execution(public void addAccount())")

    @Before("execution(void add*())")
    public void beforeAddAccountAdvice() {
        System.out.println("\n=====>>> Executing @Before Advice on addAccount()");
    }
}
```

Output:



```
Apr 29, 2017 10:45:43 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@...

====>>> Executing @Before advice on method
class com.luv2code.aopdemo.dao.AccountDAO: DOING MY DB WORK: ADDING AN ACCOUNT
class com.luv2code.aopdemo.dao.MembershipDAO: DOING STUFF: ADDING A MEMBERSHIP ACCOUNT
Apr 29, 2017 10:45:43 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@...

@Before("execution(void add*())")
```

```
@Component
public class MembershipDAO {

    public boolean addSillyMember() {

        System.out.println(getClass() + "
```

Match method with ANY Return Type - use wildcards

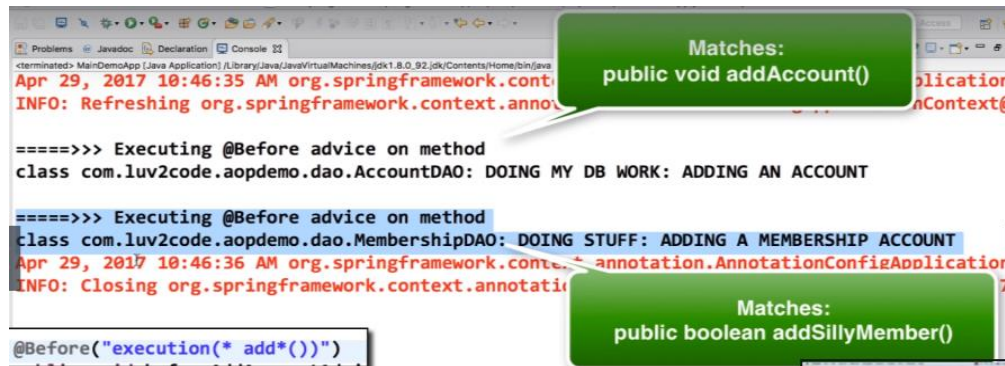
This time we don't care what the return is, we just put wildcard there and we are allowing any return type, be it string, Boolean, customer class, etc.

// this is where we add all the log messages
// let's start with an @Before

Will match with ANY Return Type for add* methods

```
@Before("execution(* add*())")
public void beforeAddAccountAdvice() {
```

Output:



```
Apr 29, 2017 10:46:35 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@...

====>>> Executing @Before advice on method
class com.luv2code.aopdemo.dao.AccountDAO: DOING MY DB WORK: ADDING AN ACCOUNT
====>>> Executing @Before advice on method
class com.luv2code.aopdemo.dao.MembershipDAO: DOING STUFF: ADDING A MEMBERSHIP ACCOUNT
Apr 29, 2017 10:46:36 AM org.springframework.context.annotation.AnnotationConfigApplicationContext: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@...

@Before("execution(* add*())")
```

Matches:
public void addAccount()

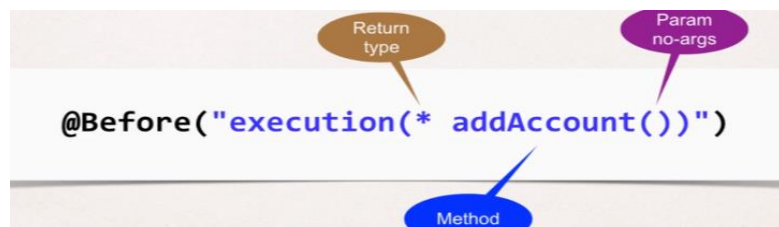
Matches:
public boolean addSillyMember()

Match on Parameters of Methods:

- For param-pattern
 - () - matches a method with no arguments
 - (*) - matches a method with one argument of any type
 - (..) - matches a method with 0 or more arguments of any type

Examples:

1. Match `addAccount` methods no arguments.



with

2. Match **addAccount** methods that have **Account** object as param. So, in this case we need to give fully qualified name of Account class.

Explanation:

File: **MyDemoLoggingAspect.java**



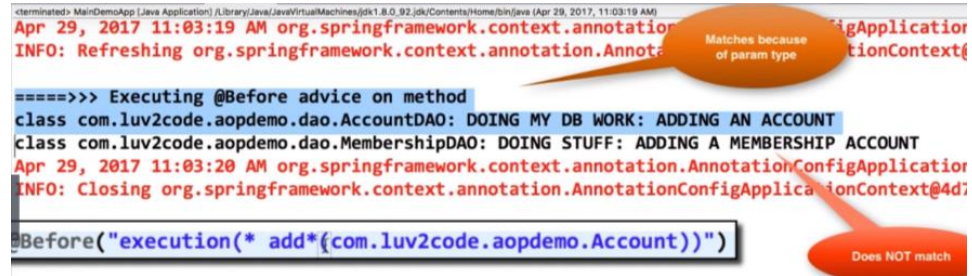
Create a class **Account**, and put it as a param in addAccount method in the AccountDAO class.

```
public class Account {  
  
    private String name;  
    private String level;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getLevel() {  
        return level;  
    }  
    public void setLevel(String level) {  
        this.level = level;  
    }  
}
```

File: **AccountDAO.java**

```
@Component  
public class AccountDAO {  
  
    public void addAccount(Account theAccount) {  
        System.out.println(getClass()+" : Doing my DB work: Adding an  
Account");  
    }  
}
```


Output:



```
<terminated> MainDemoApp [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Apr 29, 2017, 11:03:19 AM)
Apr 29, 2017 11:03:19 AM org.springframework.context.annotation.AnnotationConfigApplicationContext
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext
====>>> Executing @Before advice on method
class com.luv2code.aopdemo.dao.AccountDAO: DOING MY DB WORK: ADDING AN ACCOUNT
class com.luv2code.aopdemo.dao.MembershipDAO: DOING STUFF: ADDING A MEMBERSHIP ACCOUNT
Apr 29, 2017 11:03:20 AM org.springframework.context.annotation.AnnotationConfigApplicationContext
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4d
@Before("execution(* add*(com.luv2code.aopdemo.Account))")
```

Note:

1. in case we don't give the fully qualified Class name then it will throw error.
2. And if there is one more param after Account object then we can mention wildcards after that which means there can 0 or more params followed by Account object.

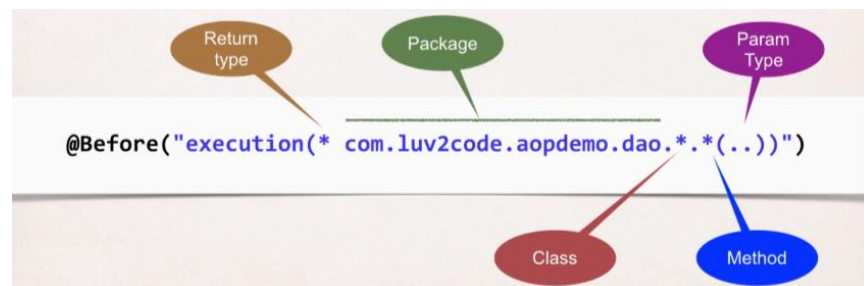
Ex. Putting wildcards.

```
@Before("execution(* add*( com.luv2code.aopdemo.Account, ..))")
public void beforeAddAccountAdvice() {
    System.out.println("\n====>>> Executing @Before Advice on addAccount()");
}
```

3. Match addAccount methods with any number of arguments.



4. Match any method in our DAO package: com.luv2code.aopdemo.dao

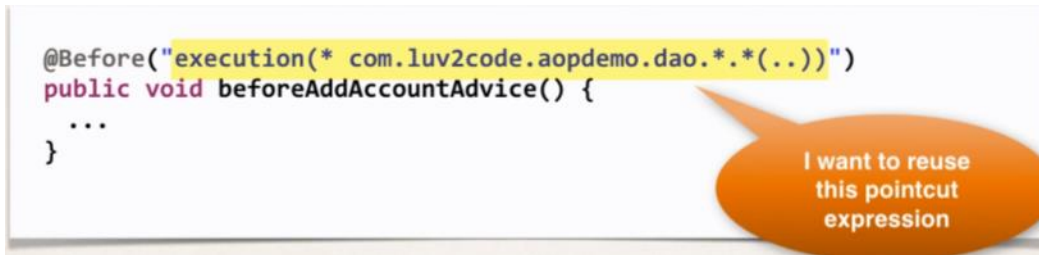


Explanation: Copy the package name such as: com.luv2code.aopdemo.dao, then spring AOP will only apply the advice on the objects created against the classes inside the mentioned package name.

➤ Pointcut Declarations:

Problem:

How can we reuse a pointcut expression? As we want to apply it to multiple advices in my AOP aspect.



Possible solutions are:

1. Copy and paste, but it's now ideal as on update we need to go to all the locations.
2. So, ideal solution should be creating a pointcut declaration once and apply it to multiple advices.

Development Process:

1. Create a pointcut declaration

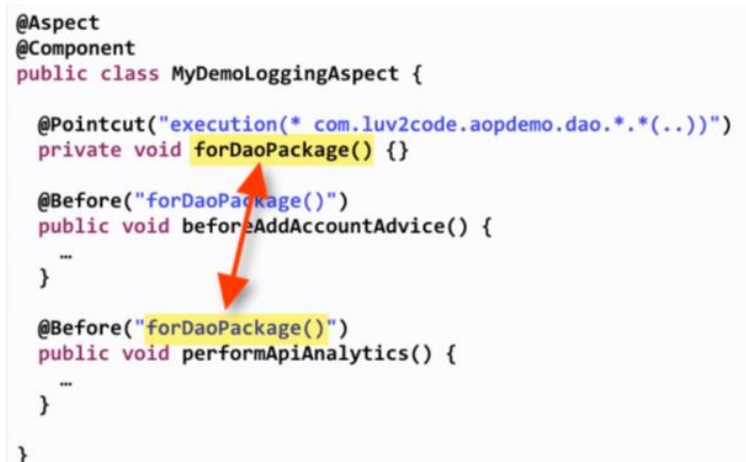
- We will use **@Pointcut** annotation.
- And then we will have a **reference** for the pointcut expression.
- Then setup a method name that we will use to reference the pointcut.
- Then we can take the method name which is having the reference of the pointcut expression and apply it to any number of advices.



Benefits of Pointcut Declarations:

- Easily reuse pointcut expressions
- Update pointcut at one location.
- Can also share and combine pointcut expressions.

2. pointcut declaration to advices.



➤ Combining Pointcuts:

Problems

1. How to apply multiple pointcut expressions to a single advice.
2. Execute an advice only if certain conditions are met.
3. For ex: All methods in a package **EXCEPT** getter/setter methods.

We can combine pointcut expressions using **logic operators**:

- **AND (&&)**
- **OR (||)**
- **NOT (!)**

Pointcut expressions when we combine works like “if” statement.

The Execution happens only if it evaluates to true. Examples given below:

```
@Before("expressionOne() && expressionTwo()")  
  
@Before("expressionOne() || expressionTwo()")  
  
@Before("expressionOne() && !expressionTwo()")
```

Example: All methods in a package **EXCEPT** getter/setter methods.

Development Process:

1. Create a pointcut declarations

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*(..))")  
private void forDaoPackage() {}  
  
// create pointcut for getter methods  
@Pointcut("execution(* com.luv2code.aopdemo.dao.*get*(..))")  
private void getter() {}  
  
// create pointcut for setter methods  
@Pointcut("execution(* com.luv2code.aopdemo.dao.*set*(..))")  
private void setter() {}
```

Now we will combine all these pointcut expressions that we have declared.

2. Combine pointcut declarations

```
// combine pointcut: include package ... exclude getter/setter  
@Pointcut("forDaoPackage() && !(getter() || setter())")  
private void forDaoPackageNoGetterSetter() {}
```

It means include the package but exclude any getter and setter. And we can now apply this combine pointcut on our advices.

3. Apply pointcut declaration to advices.

```

...
// combine pointcut: include package ... exclude getter/setter
@Pointcut("forDaoPackage() && !(getter() || setter())")
private void forDaoPackageNoGetterSetter() {}

@Before("forDaoPackageNoGetterSetter()")
public void beforeAddAccountAdvice() {
    ...
}

```

Apply pointcut declaration to advice

Output: We can see in the logs that no advice is being applied to our getter/setter methods in the mentioned package inside the pointcut expression.

```

=====>>> Executing @Before Advice on addAccount()

=====>>> Performing API Analytics
class com.luv2code.aopdemo.dao.AccountDAO: Doing my DB work: Adding an Account

=====>>> Executing @Before Advice on addAccount()

=====>>> Performing API Analytics|
class com.luv2code.aopdemo.dao.AccountDAO: doWork()
class com.luv2code.aopdemo.dao.AccountDAO: setName()
class com.luv2code.aopdemo.dao.AccountDAO: setServiceCode()
class com.luv2code.aopdemo.dao.AccountDAO: getName()
class com.luv2code.aopdemo.dao.AccountDAO: getServiceCode()

=====>>> Executing @Before Advice on addAccount()

=====>>> Performing API Analytics
class com.luv2code.aopdemo.dao.MembershipDAO Doing Stuff: Adding a membership account

=====>>> Executing @Before Advice on addAccount()

=====>>> Performing API Analytics
class com.luv2code.aopdemo.dao.MembershipDAO: I am going to sleep

```

Section 39: AOP: Ordering Aspects

0 / 3 | 24min

Control Aspect Order [Java project link](#)

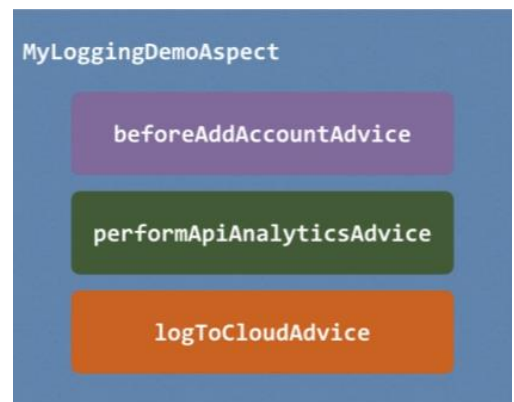
How to Control the order of advices being applied?

Let say we have aspect class; named MyLoggingDemoAspect which has four advices

1. beforeAddAccountAdvice
2. performApiAnalyticsAdvice
3. logToCloudAdvice

Acc. To the Specification the order of their execution is undefined, and spring will just pick one of them and run it and then pick the next and then next.

But in our app, we want to control the order on these advices inside the aspect that are applied and executed then we have to do some additional work.



To control Order

1. Refactor your code: Place advices in separate Aspects and
2. using this approach, we can actually control the order of the aspects using the `@Order` annotation.
3. It will guarantee order of when Aspects are applied.

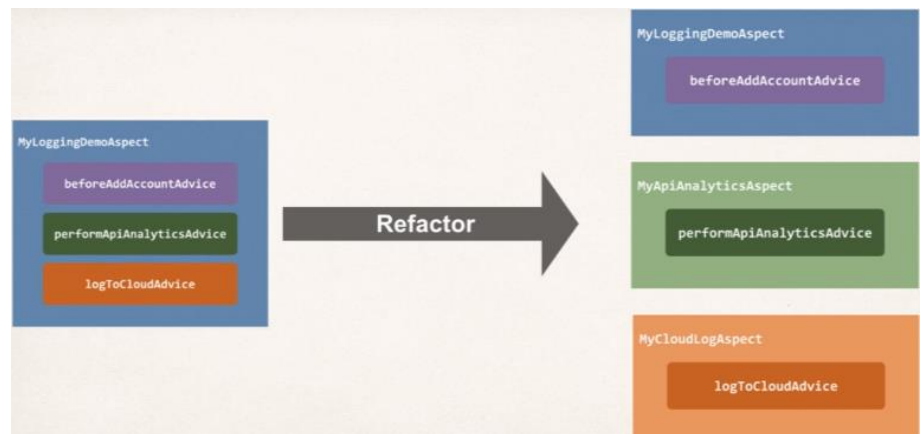
Development Process:

1. **Refactor:** Place advices in separate Aspects

So, we have divided the aspect into three other aspects:

1. MyLoggingdemoAspect
2. MyApiAnalyticsAspect
3. MyCloudLogAspect

This way we can order the different aspects.



2. Add **@Order** Annotation to Aspects

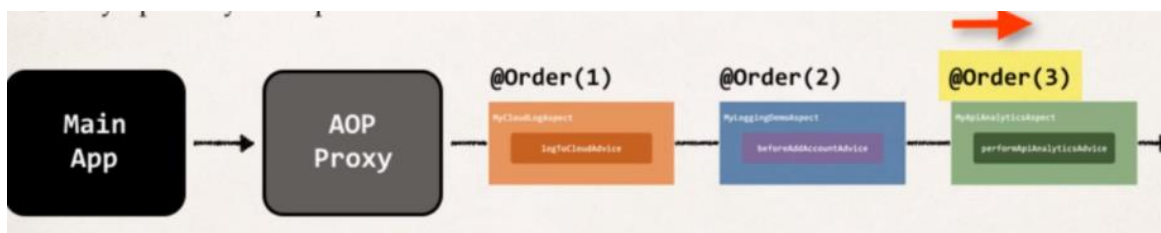
The aspects with lower number having advices will run before the aspects having higher number in the **@Order** annotation.

So, the order in which advices of different aspect will execute:

- Control order on Aspects using the `@Order` annotation

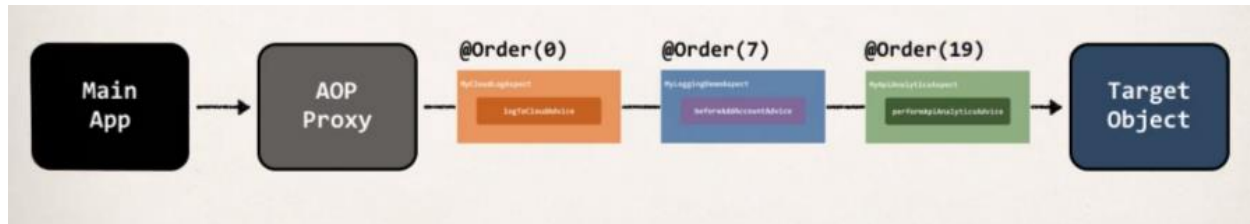
```
@Order(1)
public class MyCloudLogAspect {
    ...
}
```

- Guarantees order of when Aspects are applied
- Lower numbers have higher precedence

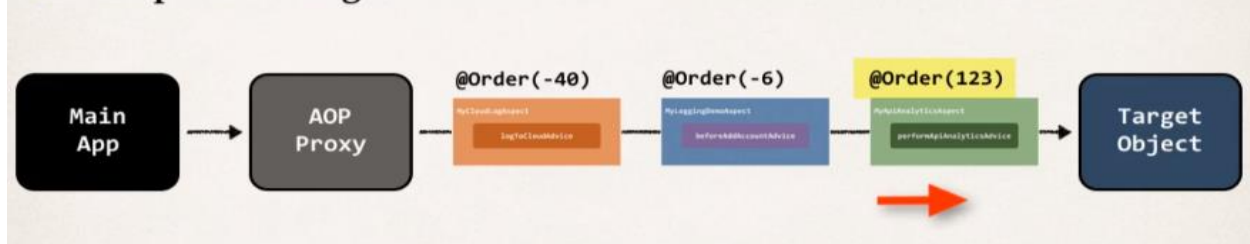


@Order Annotation

- Lower numbers having higher precedence
- Range: Integer.MIN_VALUE to Integer.MAX_VALUE.
- Negative Numbers are allowed.
- Also, the order sequence does not need to be consecutive as shown in attached pic.



• Example with negative numbers



FAQs: What if aspects have the exact same @Order annotation?

Both having @Order(6) annotation then the order becomes undefined.

However, they will still run After MyCloudLogAspect and Before MyLoggingDemoAspect.

```
@Order(1)
public class MyCloudLogAspect { ... }

@Order(6)
public class MyShowAspect { ... }

@Order(6)
public class MyFunnyAspect { ... }

@Order(123)
public class MyLoggingDemoAspect { ... }
```

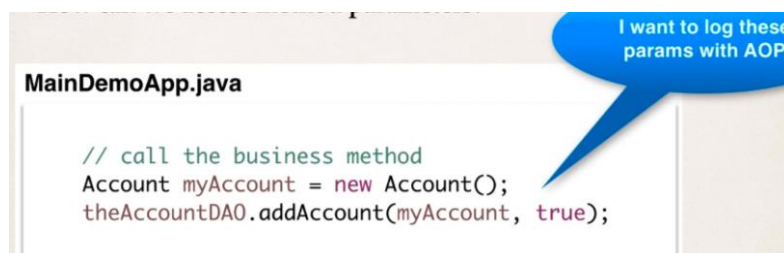
The order at this point is undefined

Section 40: AOP: JoinPoints

0 / 2 | 15min

Reading Method Arguments with JoinPoints [Java Project Link](#)

Problem: When we are in an aspect (i.e., for logging) then how can we access method parameters?



Development process:

1. Access and display method signature.

For that we can add a new argument i.e `JoinPoint` which gives us information about the method which will actually execute after this advice.

And to access method signature we will be using the method `getSignature()` and downcast it to `MethodSignature` object and then we are simply printing the method signature

- ✓ **`JoinPoint`** Belongs to `org.aspectj.lang` package.

MainDemoApp.java

```
// call the business method
Account myAccount = new Account();
theAccountDAO.addAccount(myAccount, true);
```

```
@Before("...")
public void beforeAddAccountAdvice(JoinPoint theJoinPoint) {

    // display the method signature
    MethodSignature methodSig = (MethodSignature) theJoinPoint.getSignature();

    System.out.println("Method: " + methodSig);
}

Method: void com.luv2code.aopdemo.dao.AccountDAO.addAccount(Account,boolean)
```

And in the **output**, we are getting fully qualified name of the method with their parameters that has to be passed.

2. Access and display method arguments.

And using the same **`JoinPoint`** we can get the arguments used by calling the method `getArgs()`, which will return an array of objects.

In the **output**,

We will get the account object with fully qualified class name and the Boolean value passed into the method call.

And then we can log the method signature as well as the arguments.

```
@Before("...")
public void beforeAddAccountAdvice(JoinPoint theJoinPoint) {

    // display method arguments

    // get args
    Object[] args = theJoinPoint.getArgs();

    // loop thru args
    for (Object tempArg : args) {
        System.out.println(tempArg);
    }
}

com.luv2code.aopdemo.Account@1ce24091
true
```

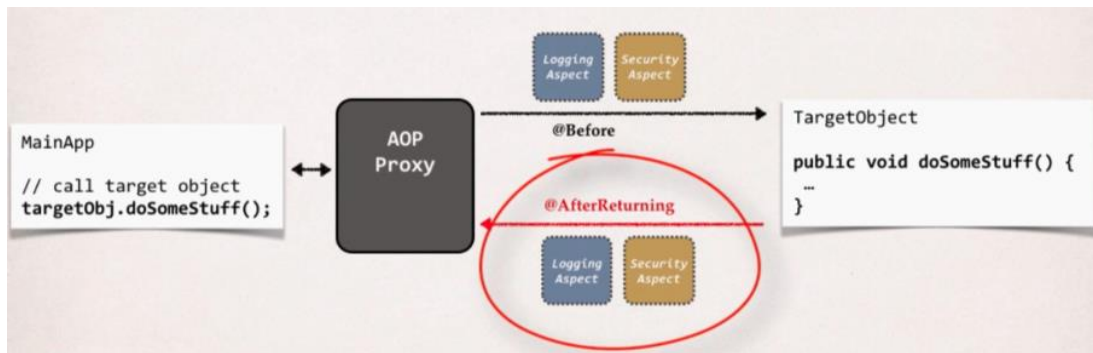
Section 41: AOP: @AfterReturning Advice Type

0 / 5 | 39min

➤ @AfterReturning Advice [Java Project Link](#)

This runs after the method's successful execution.

With AOP, we have methods calls going in and we can preprocess using `@Before`, and after the target object method gets executed then on returning back then we can make use of this advice type `@AfterReturning` which will have code that will execute after the target object method being executed successfully.



Use Case:

- ❖ **Most common:** Logging, security, transactions
- ❖ **Audit Logging:** Who, what, when, where
- ❖ **Post processing Data:**
 1. Post process the data before returning to the caller.
 2. Format the data or enrich the data.

Example:

```

@AfterReturning("execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..)")
public void afterReturningFindAccountsAdvice() {

    System.out.println("Executing @AfterReturning advice");

}
  
```

Access the Return value:

Need to access the return value of called method.

As in my example here, the method after its execution is returning List of Accounts. And we want access the return value i.e., List of accounts. And want to log them or process them.

```

Target Object
AccountDAO

List<Account> findAccounts()
  
```

So, for that we will use one attribute inside the pointcut expression returning and it keeps the parameter name which will hold the value of returned data.

```

@AfterReturning(
    pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..)",
    returning="result")
public void afterReturningFindAccountsAdvice() {
  
```

Parameter name for return value

We can use any name in the returning attribute inside the pointcut expression but we need to be consistent while giving the parameter to the advice.

So, we have to be consistent between the parameter args and annotation.

Ex:

```
@AfterReturning(  
    pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..)",  
    returning="result")  
    public void afterReturningFindAccountsAdvice(  
        JoinPoint theJoinPoint, List<Account> result) {  
  
    }  
}
```

Parameter name for return value

And by this time, AOP has injected the return value in the variable result and we are simply printing them on the console.

```
@AfterReturning(  
    pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..)",  
    returning="result")  
    public void afterReturningFindAccountsAdvice(  
        JoinPoint theJoinPoint, List<Account> result) {  
  
        // print out the results of the method call  
        System.out.println("\n====>> result is: " + result);  
  
    }  
}
```

Development Process: [AfterReturningDemoApp.java](#) is our main App

Development Process - @AfterReturning

Step-By-Step

1. Prep Work: Add constructors to Account class
2. Add new method: findAccounts() in AccountDAO
3. Update main app to call the new method: findAccounts()
4. Add @AfterReturning advice

Output: We are getting the return value inside the advice in our aspect.

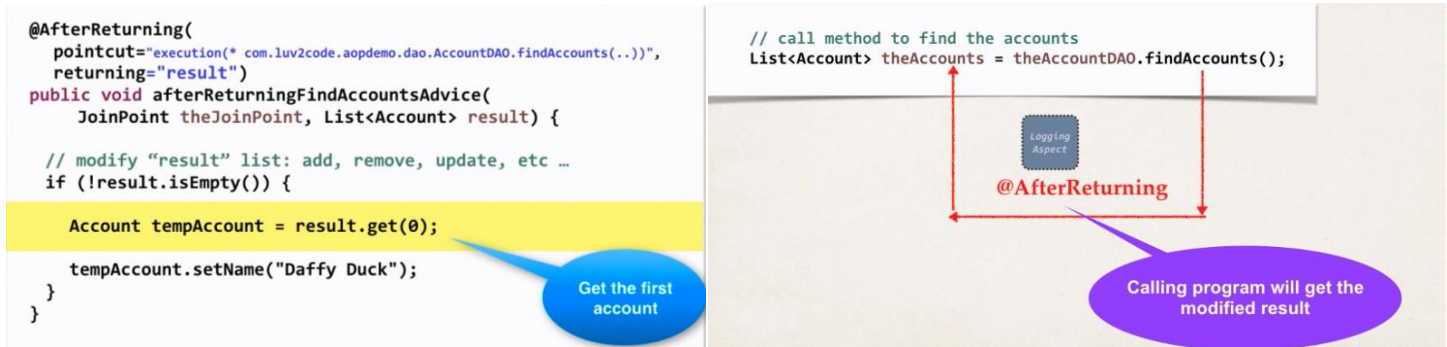
```
====>> Performing API analytics  
====>> Executing @AfterReturning on method: AccountDAO.findAccounts()  
====>> result is: [Account [name=John, level=Silver], Account [name=Madhu, level=Platinum], Account [name=Lucy, level=Gold]]  
  
Main Program: AfterReturningDemoApp  
----  
[Account [name=John, level=Silver], Account [name=Madhu, level=Platinum], Account [name=Lucy, level=Gold]]
```

From our Main Program

Post-processing/Modifying the Data: [AfterReturningDemoApp.java](#)

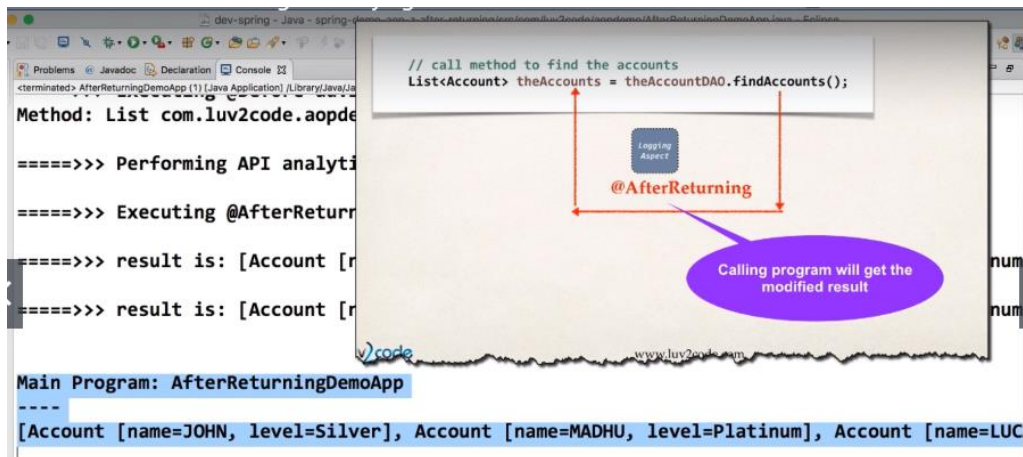
So, we can post process the data before returning to the caller.

In this example we are fetching the first Account Object and updating the name property of the object and then it will be sent to the caller in the main App. And the calling program will get the new data in the whole process.



So, the AOP is intercepting the calls and modifying the value in here.

Here is the Output: Converting all the names of the Account object to the uppercase.

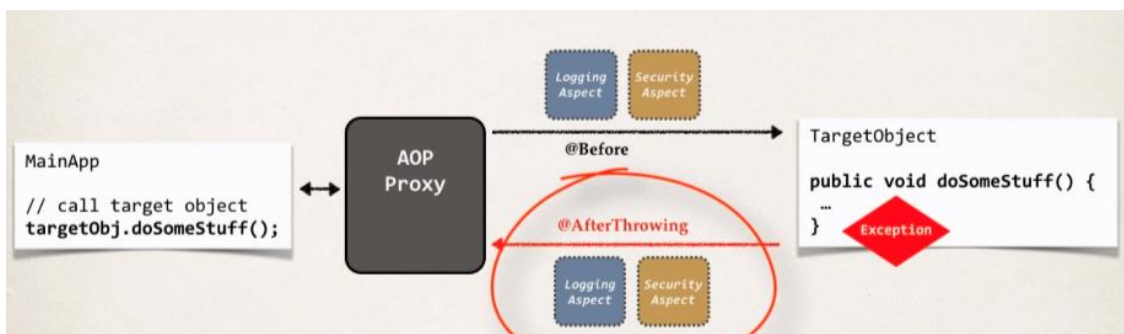


Section 42: AOP: @AfterThrowing Advice Type

0 / 2 | 21min

@AfterThrowing

Incase an exception is thrown, the using this advice type we can post process that exception which occurred in the target object.

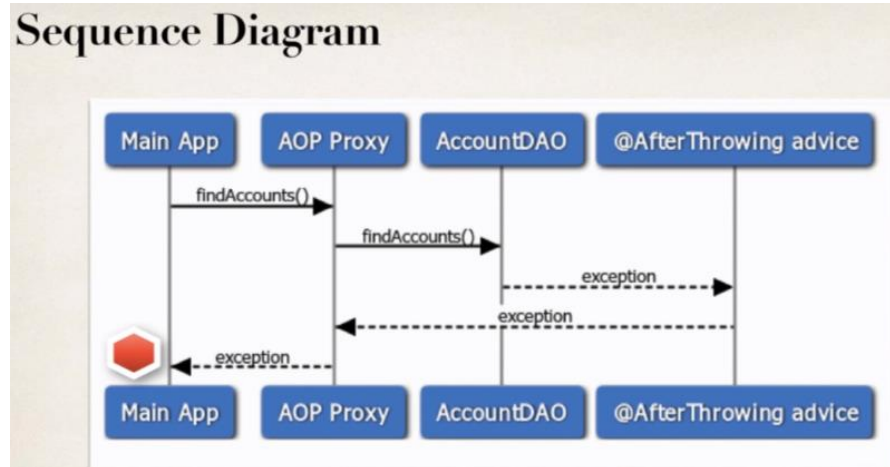


As soon as the exception is thrown then this `@AfterThrowing` advice type will get triggered.

Use Cases:

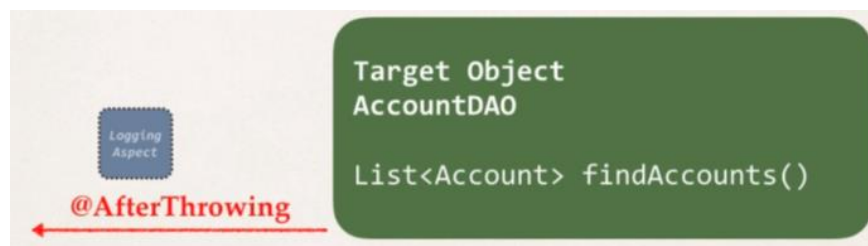
- Log the exception.
- Perform auditing on the exception

- Notify DevOps team via email or SMS.
- Encapsulate this functionality in AOP for easy reuse.



Example:

We will create an advice that will run after an exception is thrown.



So, we will use this annotation called **@AfterThrowing**, then I will give the pointcut expression.

Access the Exception Object

For that we will use throwing attribute inside the annotation, and later we will enter on more parameter inside the method arguments which will hold the exception object.

```

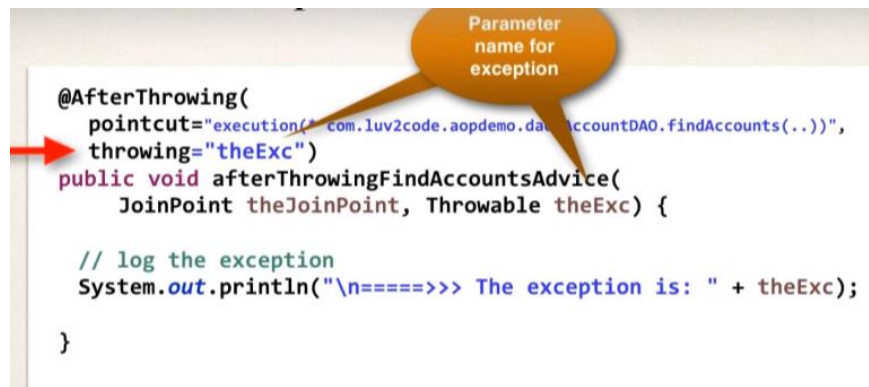
@AfterThrowing(
    pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..))",
    throwing="theExc")
public void afterThrowingFindAccountsAdvice() {
  
```

Parameter name for exception

- Throwable theExc will matched the actual parameter that we have entered in the annotation.
 - When the exception will be thrown then we can handle it in our advice and make use of this theExc object.
- ```

@AfterThrowing(
 pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..))",
 throwing="theExc")
public void afterThrowingFindAccountsAdvice(
 JoinPoint theJoinPoint, Throwable theExc) {

```



```

@AfterThrowing(
 pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..))",
 throwing="theExc")
public void afterThrowingFindAccountsAdvice(
 JoinPoint theJoinPoint, Throwable theExc) {

 // log the exception
 System.out.println("\n====>>> The exception is: " + theExc);

}

```

Parameter name for exception

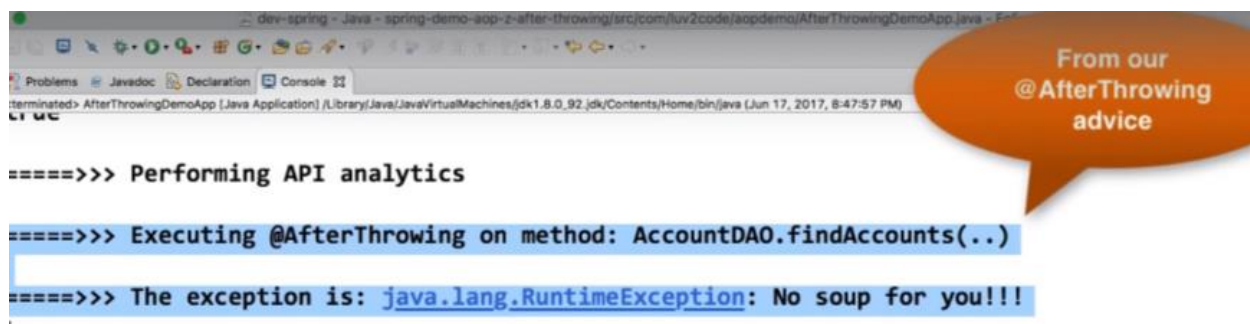
- At this point, we are only intercepting the exception (reading it).
  - However, the exception is still propagated to calling program.
- Now if we want to stop the **Exception Propagation** Such that the exception never makes to the main app i.e calling method then we have to use **@Around** advice.

## Development Process - @AfterThrowing

1. In Main App, add a try / catch block for exception handling
2. Modify AccountDAO to simulate throwing an exception
3. Add @AfterThrowing advice

### Output:

And the Advice code gets triggered and log the exception before the exception make its way to the calling method, after the target method gives exception.



```

=====>>> Performing API analytics

=====>>> Executing @AfterThrowing on method: AccountDAO.findAccounts(..)

=====>>> The exception is: java.lang.RuntimeException: No soup for you!!!

```

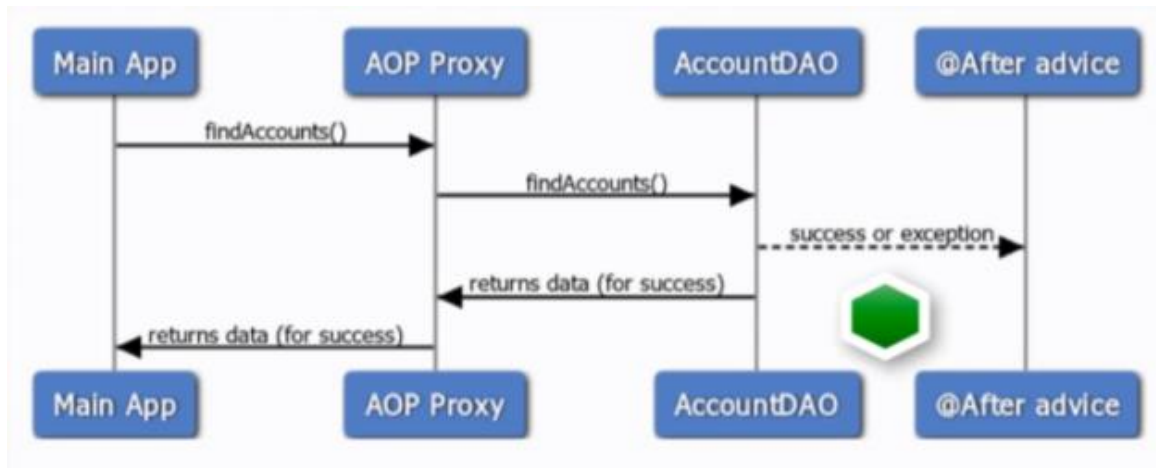
From our @AfterThrowing advice

## Section 43: AOP: @After Advice Type

0 / 3 | 13min

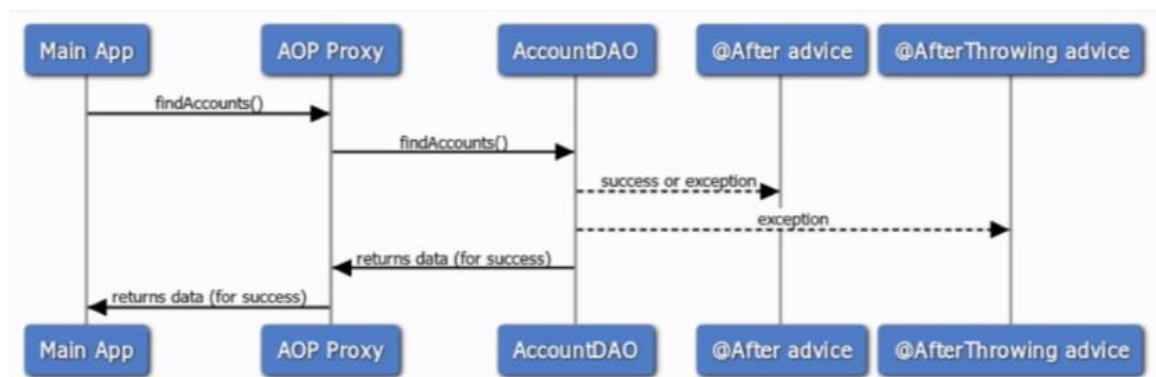
### @After Overview [Java Project Link](#)

This advice runs after the method is completed regardless of outcome/exceptions. This advice will always run as our finally block of try and catch in java.



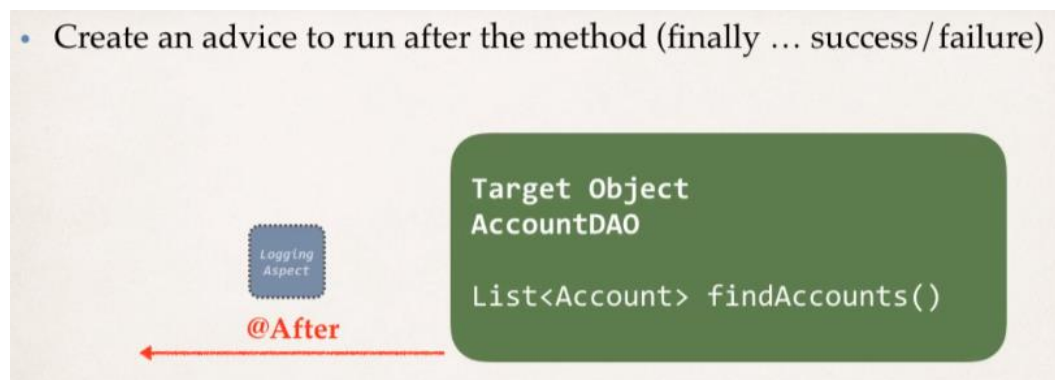
### @AfterThrowing and @After, which will execute first?

@After will execute before @AfterThrowing. It's just like in try catch where finally will always execute before the exception is being thrown.



### Example:

- Create an advice to run after the method (finally ... success/failure)



## Coding @After Advice

- This advice will run after the method (finally ... success / fail)

Pointcut Expression

```
@After("execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..)")
public void afterFinallyFindAccountsAdvice() {

 System.out.println("Executing @After (finally) advice");

}
```

## @After Advice - Tips

- The @After advice does not have access to the exception
- If you need exception, then use @AfterThrowing advice
- The @After advice should be able to run in the case of success or error
- Your code should not depend on happy path or an exception
- Logging / auditing is the easiest case here

### Development Process:

1. Prep work
2. Add @After Advice
3. Test for failure/ exception case
4. Test for success case.

- In case of failure/exception;  
**output:**

As we have discussed that @AfterThrowing will execute after @After advice but here it's happening otherwise.

```
Problems @ Javadoc Declaration Console
<terminated> AfterfinallyDemoApp [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 10, 2021, 11:36:39 PM - 1
=====>>> Executing @Before Advice on addAccount()
Method: List com.luv2code.aopdemo.dao.AccountDAO.findAccount(boolean)
true
=====>>> Logging to cloud in async fashion
=====>>> Performing API Analytics
=====>>> Executing @AfterThrowing on method: AccountDAO.findAccount(...)
=====>>> result is: java.lang.RuntimeException: No Soup For you!
=====>>> Executing @After (Finally) on method: AccountDAO.findAccount(...)

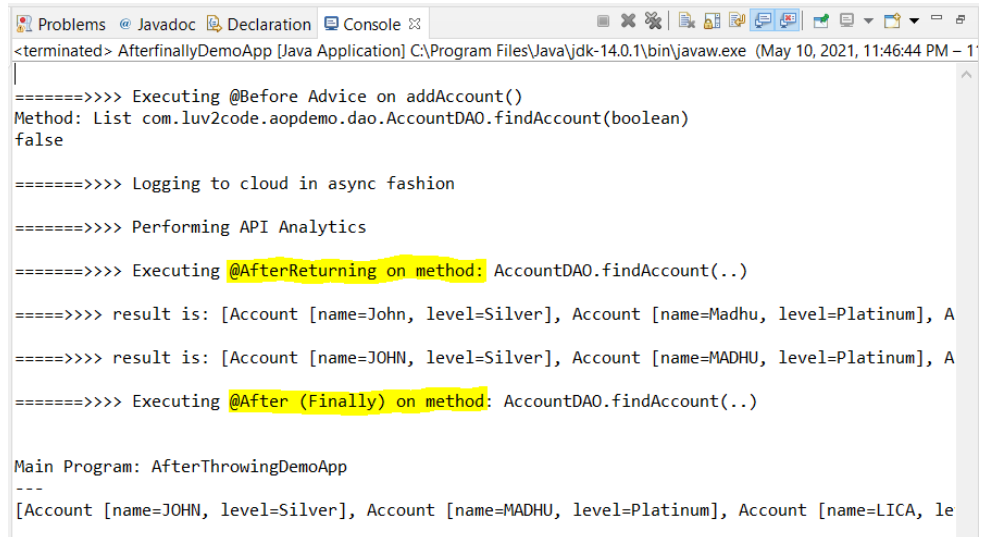
Main Program ... Caught Exception: java.lang.RuntimeException: No Soup For you!

Main Program: AfterThrowingDemoApp

null
```

- In case of success; **output:**

Again, the same thing is happening, So, we have the reason regarding this below:



```
<terminated> AfterFinallyDemoApp [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 10, 2021, 11:46:44 PM - 1'
|
=====>>> Executing @Before Advice on addAccount()
Method: List com.luv2code.aopdemo.dao.AccountDAO.findAccount(boolean)
false

=====>>> Logging to cloud in async fashion

=====>>> Performing API Analytics

=====>>> Executing @AfterReturning on method: AccountDAO.findAccount(..)

=====>>> result is: [Account [name=John, level=Silver], Account [name=Madhu, level=Platinum], A
=====>>> result is: [Account [name=JOHN, level=Silver], Account [name=MADHU, level=Platinum], A
=====>>> Executing @After (Finally) on method: AccountDAO.findAccount(..)

Main Program: AfterThrowingDemoApp

[Account [name=JOHN, level=Silver], Account [name=MADHU, level=Platinum], Account [name=LICA, le
```

## HEADS UP - @After Advice running after @AfterThrowing advice

You may have noticed that in the latest versions of Spring, the @After Advice is running **AFTER** the @AfterThrowing advice. This output is different than what I showed in the original video. There were recent changes in the Spring 5.2.7 (released on 9 June 2020).

Starting with Spring 5.2.7:

- if advice methods defined in the same @Aspect class that need to run at the same join point
- the @After advice method is invoked AFTER any @AfterReturning or @AfterThrowing advice methods in the same aspect class

So, in our case, the @After and @AfterThrowing are in the same aspect class: MyDemoLoggingAspect.java, hence in latest Spring 5.2.7, the @After will run AFTER the @AfterThrowing.

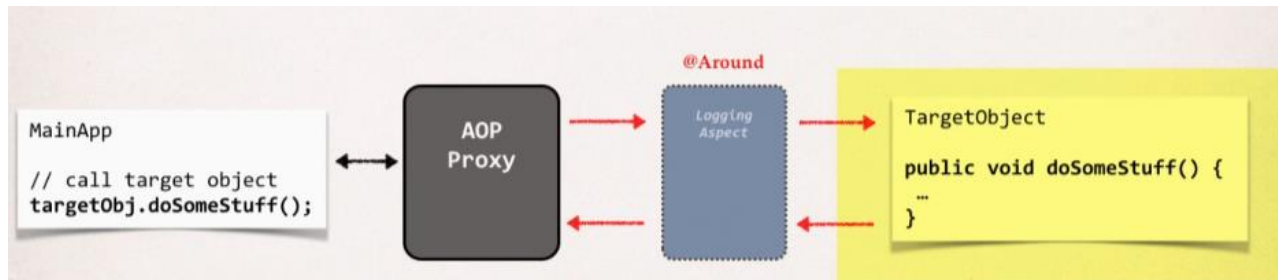
As of Spring Framework 5.2.7, advice methods defined in the same @Aspect class that need to run at the same join point are assigned precedence based on their advice type in the following order, from highest to lowest precedence: @Around, @Before, @After, @AfterReturning, @AfterThrowing. Note, however, that an @After advice method will effectively be invoked after any @AfterReturning or @AfterThrowing advice methods in the same aspect, following AspectJ's "after finally advice" semantics for @After.



## Section 44: AOP: @Around Advice Type

0 / 7 | 51min

@Around Advice: [Java Project Link](#)

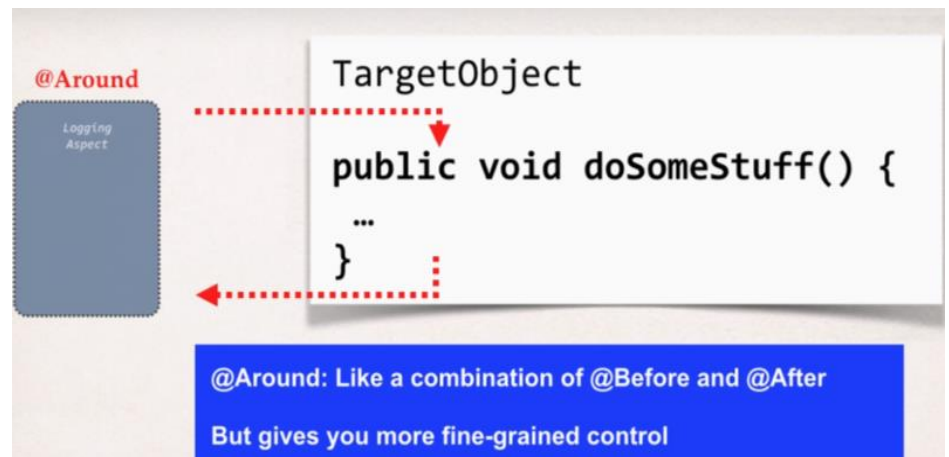


This advice will run before and after the method execution.

### Use Case:

1. Most common: Logging. Auditing, security
2. Preprocessing and Post Processing data.
3. Instrumentation/profiling code.

i.e., How long does it take for a section of code to run?



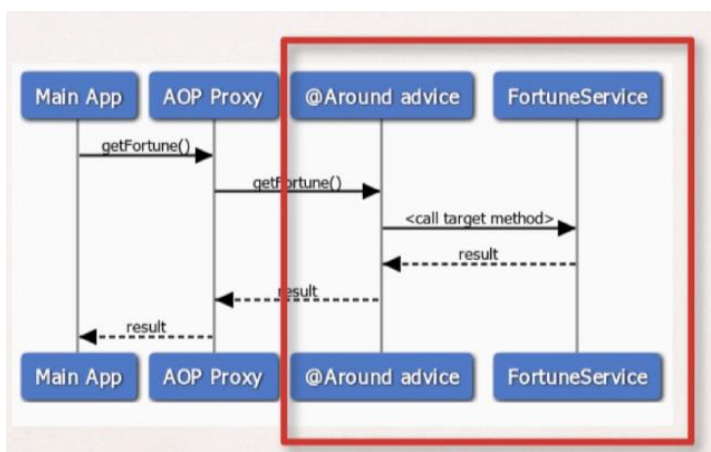
4. Manage Exceptions

i.e., Swallow/handle/stop Exception

There may be cases where we don't want exception to be propagated all the way to the main app.

Here for Example, we will revisit our FortuneService that we have discussed initially in the course:

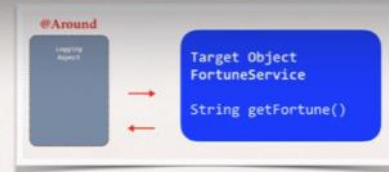
### Sequence Diagram:



Target Object  
FortuneService  
String getFortune()

# ProceedingJoinPoint

- When using @Around advice
- You will get a reference to a “**proceeding join point**”
- This is a handle to the **target method**
- Your code can use the **proceeding join point** to execute **target method**



When we make use of the @Around Advice, then we get a reference of proceeding JoinPoint.

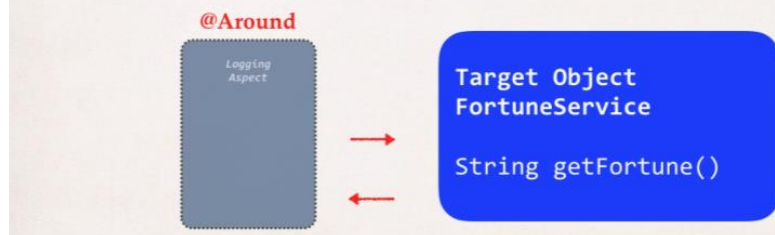
```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..)")
public Object afterGetFortune(
 ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {
```

Handle to  
target method

## Example:

Here, we will check how long does a method take to run.

- Create an advice for instrumentation / profiling code
- How long does it take for a section of code to run?



## Coding:

```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..)")
public Object afterGetFortune(
 ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {
```

```
// get begin timestamp
long begin = System.currentTimeMillis();
```

```
// now, let's execute the method
Object result = theProceedingJoinPoint.proceed();
```

Handle to  
target method

Execute the  
target method

```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..)")
public Object afterGetFortune(
 ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {
```

```
// get begin timestamp
long begin = System.currentTimeMillis();
```

```
// now, let's execute the method
Object result = theProceedingJoinPoint.proceed();
```

```
// get end timestamp
long end = System.currentTimeMillis();
```

```
// compute duration and display it
long duration = end - begin;
System.out.println("\n====> Duration: " + duration + " milliseconds");
```

```
return result;
```

We are getting the duration and printing it onto the console and then finally returning the result to the calling program.

## Development Process:

1. Prep Work
2. Create TrafficFortuneService

File: [TrafficFortuneService.java](#)

```
public class TrafficFortuneService {

 public String getFortune() {

 //simulate a delay
 try {
 TimeUnit.SECONDS.sleep(5);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }

 //return a fortune
 return "Expected Heavy Traffic this morning";
 }
}
```

3. Update main app to call TrafficFortuneService

File: [AroundDemoApp.java](#)

```
//read spring config java class
AnnotationConfigApplicationContext context =
 new AnnotationConfigApplicationContext(DemoConfig.class);

TrafficFortuneService theFortuneService =
 context.getBean("trafficFortuneService",TrafficFortuneService.class);

System.out.println("\nMain Program: AroundDemoApp");

System.out.println("Calling Fortune");

String data = theFortuneService.getFortune();
System.out.println("\nMy Fortune is: "+data);

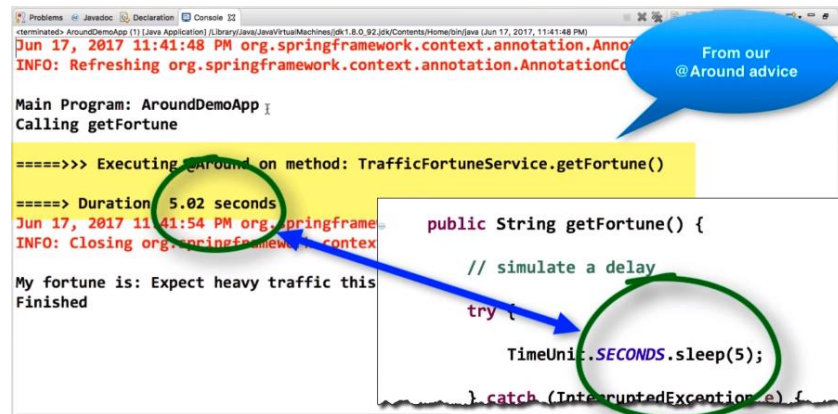
System.out.println("FINISH");

//close the context
context.close();
```

4. Add @Around Advice.

File: [MyDemoLoggingAspect.java](#)

## Output:



```

<terminated> AroundDemoApp (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Jun 17, 2017, 11:41:48 PM)
Jun 17, 2017 11:41:48 PM org.springframework.context.annotation.AnnotationConfigApplicationContext:INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4d7
Main Program: AroundDemoApp
Calling getFortune

====>> Executing around on method: TrafficFortuneService.getFortune()
====> Duration 5.02 seconds
Jun 17, 2017 11:41:54 PM org.springframework.context.annotation.AnnotationConfigApplicationContext:INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4d7
My fortune is: Expect heavy traffic this morning
Finished

public String getFortune() {
 // simulate a delay
 try {
 TimeUnit.SECONDS.sleep(5);
 } catch (InterruptedException e) {
 // ...
 }
}

```

As in the above output we can see that our output on the console is being printed out in the end.

## So, we need to resolve this order issue:

Here is the sample:

### Root cause:

1. Data is printing to two different output streams.
2. Spring is printing to the logger output stream
3. System.out.println is printing to the standard out output stream.

### Solution:

1. To have everything in order, you should send to same output stream
2. We'll change our code to use the logger output stream...same as Spring.

### File: AroundWithLoggerDemoApp.java

```

public class AroundWithLoggerDemoApp {

 private static Logger myLogger = Logger.getLogger(AroundWithLoggerDemoApp.class.getName());

 public static void main(String[] args) {
 ...
 }
}

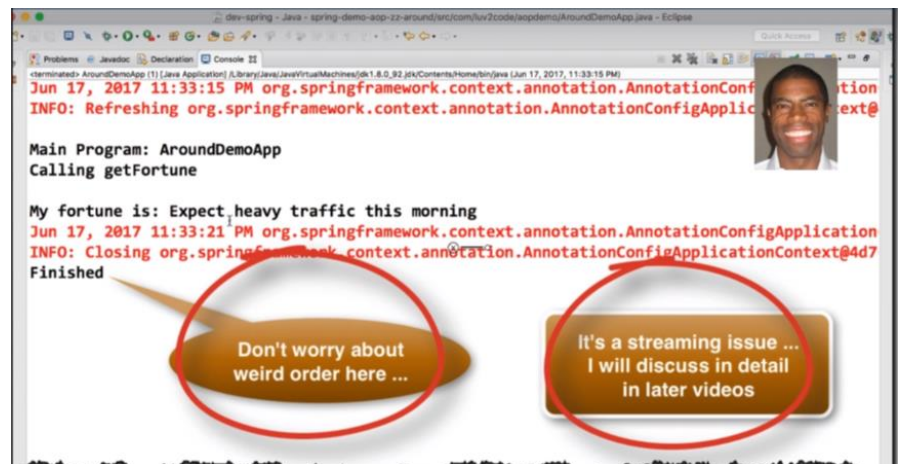
```

And Replace all the System.out.println with myLogger.info.

### File: MyDemoLoggingAspect.java

Perform the same thing as we did above.

And in the **output** we will get everything in order.

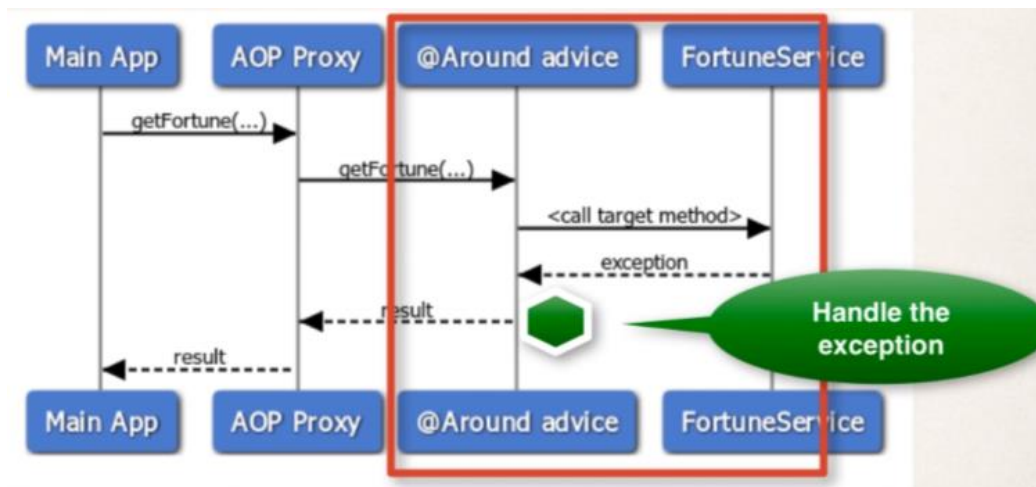


## ➤ @Around Advice – Handle Exception [Java Project Link](#)

### ProceedingJoinPoint:

- For an exception thrown from proceeding Join point
  - You can handle.swallow /stop the exception
  - Or you can simply rethrow the exception
- This gives you fine-grained control over how the target method is called.

### Sequence diagram



So, when the exception is thrown, we actually handle it at our @Around Advice and in @Around Advice we can log the exception and we will assign a default fortune that we can pass back as a result of the main program, and the main program will never know that the exception is thrown as we have handled it in our @Around Advice.

### Handling Exception

We will keep `theProceedingJoinPoint.proceed()` inside the try catch block because if anything goes wrong while calling that method, then we can handle that in the catch block.

- First, we are logging the exception.
- Then, we are giving the default fortune to the calling method.
- At bottom, we are returning the result.

Here, we can rethrow the exception or assign a default value and it really depends upon our application.

```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..)")")
public Object afterGetFortune(
 ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {

 Object result = null;

 try {
 // let's execute the method
 result = theProceedingJoinPoint.proceed();
 } catch (Exception exc) {
 // log exception
 System.out.println("@Around advice: We have a problem " + exc);

 // handle and give default fortune ... use this approach with caution
 result = "Nothing exciting here. Move along!";
 }

 return result;
}
```

Use with caution ...  
depends on your app



# Development Process - @Around

1. Prep work
2. Add trip wire to simulate an exception
3. Modify @Around advice to handle exception

File: [AroundHandleExceptionDemoApp.java](#)

```
boolean tripWire = true;
String data = theFortuneService.getFortune(tripWire);
```

File: [TrafficFortuneService.java](#)

```
public String getFortune(boolean tripWire) throws Exception {
 if(tripWire) {
 throw new Exception("Major Accident! Highway is closed");
 }
 return getFortune();
}
```

File: [MyDemoLoggingAspect.java](#)

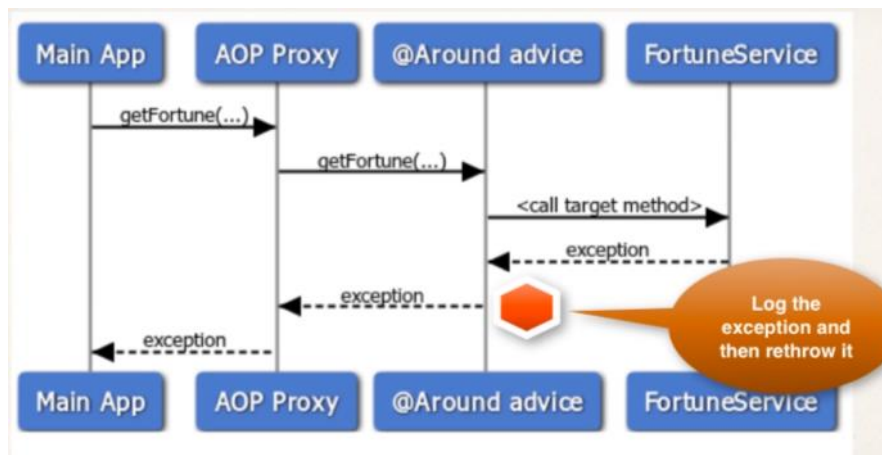
```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..)")
public Object aroundGetFortune(
 ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {
 //print out which method we are advising on
 String method = theProceedingJoinPoint.getSignature().toShortString();
 myLogger.info("\n=====>>> Executing @Around on method: "+method);
 //get begin timestamp
 long begin = System.currentTimeMillis();
 //now, let's execute the method
 Object result = null;
 try{
 result = theProceedingJoinPoint.proceed();
 }
 catch (Exception e) {
 //log the exception
 myLogger.warning(e.getMessage());
 //give user a custom message
 result = "Major Accident! but no Worries,"
 + "your orivate AOP Helicopter is on the way!";
 }
 //get end timestamp
 long end = System.currentTimeMillis();
 //compute duration and display it
 long duration = end-begin;
 myLogger.info("\n====>> Duration: "+duration/1000.0+" seconds");
 return result;
}
```

Output:

```
Main Program: AroundDemoApp
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.AroundDemoApp main
INFO: Calling getFortune
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.AroundDemoApp main
INFO:
====>>> Executing @Around on method: TrafficFortuneService.getFortune(..)
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.aspect.MyDemoLoggingAspect aroundGetFortune
WARNING: Major accident! Highway is closed!
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.aspect.MyDemoLoggingAspect aroundGetFortune
INFO:
====> Duration: 0.014 seconds
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.AroundHandleExceptionDemoApp main
INFO:
My fortune is: Major accident! But no worries, your private AOP helicopter is on the way!
Jul 01, 2017 7:00:56 AM com.luv2code.aopdemo.AroundHandleExceptionDemoApp main
INFO: Finished
Jul 01, 2017 7:00:56 AM org.springframework.context.annotation.AnnotationConfigApplicationContext
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4d7
```

## ➤ Rethrow the Exceptions

Sequence Diagram



We need to do some minor changes to our project. Comment out the previously made custom message.

File: **MyDemoLoggingAspect.java**

```
//now, let's execute the method
Object result = null;

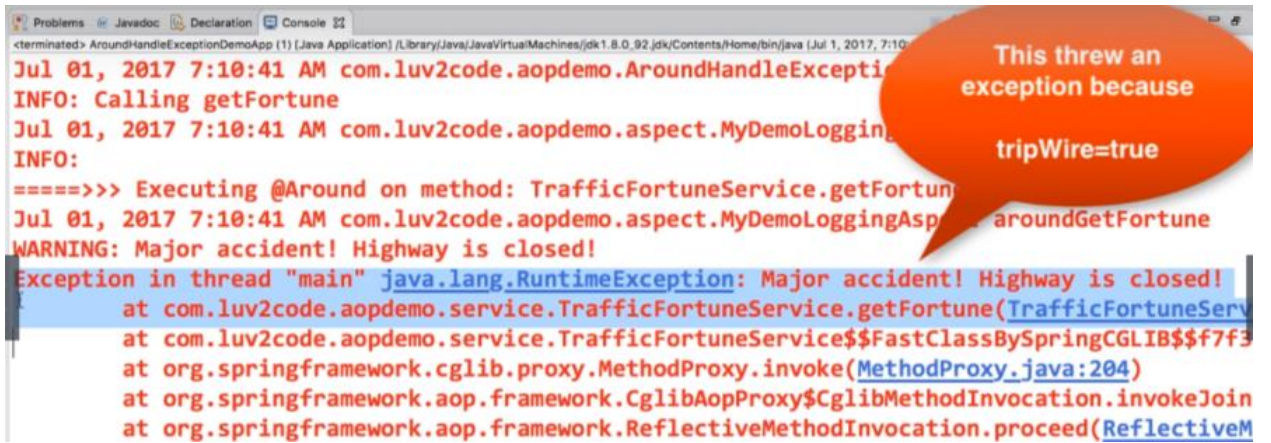
try{
 result = theProceedingJoinPoint.proceed();
}
catch (Exception e) {

 //log the exception
 myLogger.warning(e.getMessage());
 /*//give user a custom message
 result = "Major Accident! but no Worries,"
 + "your private AOP Helicopter is on the way!";*/

 //rethrow Exception
 throw e;
}
```

And Call the main() app and will get the exception thrown to the calling method.

## Output:



```
Problems Javadoc Declaration Console
<terminated> AroundHandleExceptionDemoApp (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Jul 1, 2017, 7:10:41 AM)
Jul 01, 2017 7:10:41 AM com.luv2code.aopdemo.AroundHandleExceptionDemoApp aroundHandleException
INFO: Calling getFortune
Jul 01, 2017 7:10:41 AM com.luv2code.aopdemo.aspect.MyDemoLoggingAspect aroundGetFortune
INFO:
=====>>> Executing @Around on method: TrafficFortuneService.getFortune()
Jul 01, 2017 7:10:41 AM com.luv2code.aopdemo.aspect.MyDemoLoggingAspect aroundGetFortune
WARNING: Major accident! Highway is closed!
Exception in thread "main" java.lang.RuntimeException: Major accident! Highway is closed!
 at com.luv2code.aopdemo.service.TrafficFortuneService.getFortune(TrafficFortuneService.java:204)
 at com.luv2code.aopdemo.service.TrafficFortuneService$$FastClassBySpringCGLIB$$f7f3c1e1.invoke()
 at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
 at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy$CglibMethodInvocation.java:706)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)
```

This threw an exception because tripWire=true