Aspect-Oriented Programming (AOP) Overview

© luv2code LLC

Application Architecture

© luv2code LLC

Code for Data Access Object (DAO)

```
public void addAccount(Account theAccount, String userId) {

        entityManager.persist(theAccount);
}
```


© luv2code LLC

New Requirement - Logging

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!

© luv2code LLC

DAO - Add Logging Code

```
public void addAccount(Account theAccount, String userId) {



    entityManager.persist(theAccount);

}
```


© luv2code LLC

DAO - Add Logging Code

```
public void addAccount(Account theAccount, String userId) {

    entityManager.persist(theAccount);

}
```


© luv2code LLC

New Requirement - Security

From: The Boss

- Need to add security code to our DAO
  - Make sure user is authorized before running DAO method

© luv2code LLC

Add Security Code

```
public void addAccount(Account theAccount, String userId) {



    entityManager.persist(theAccount);

}
```


© luv2code LLC

Add Security Code

```
public void addAccount(Account theAccount, String userId) {

    entityManager.persist(theAccount);

}
```


© luv2code LLC

Add Security Code

```
public void addAccount(Account theAccount, String userId) {

    entityManager.persist(theAccount);

}
```


© luv2code LLC

By the way

From: The Boss

- Let's add it to all of our layers…

© luv2code LLC

I'm Going Crazy Over Here

© luv2code LLC

Two Main Problems

- **Code Tangling**
  - For a given method: addAccount(…)
  - We have logging and security code tangled in

- **Code Scattering**
  - If we need to change logging or security code
  - We have to update ALL classes

© luv2code LLC

Other possible solutions?

- **Inheritance?**
  - Every class would need to inherit from a base class
  - Can all classes extends from your base class? … plus no multiple inheritance

- **Delegation?**
  - Classes would delegate logging, security calls
  - Still would need to update classes if we wanted to
    - add/remove logging or security
    - add new feature like auditing, API management, instrumentation

© luv2code LLC

Aspect-Oriented Programming

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

Cross-Cutting Concerns

- "Concern" means logic / functionality

© luv2code LLC

Cross-Cutting Concerns

© luv2code LLC

Aspects

- Aspect can be reused at multiple locations
- Same aspect/class … applied based on configuration

© luv2code LLC

AOP Solution

- Apply the Proxy design pattern

MainApp

```
// call target object
targetObj.doSomeStuff();
```

TargetObject

```
public void doSomeStuff() {
    ...
}
```


© luv2code LLC

Benefits of AOP

- **Code for Aspect is defined in a single class**
  - Much better than being scattered everywhere
  - Promotes code reuse and easier to change
- **Business code in your application is cleaner**
  - Only applies to business functionality: addAccount
  - Reduces code complexity
- **Configurable**
  - Based on configuration, apply Aspects selectively to different parts of app
  - No need to make changes to main application code … very important!

© luv2code LLC

Additional AOP Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **Exception handling**
  - log exception and notify DevOps team via SMS/email
- **API Management**
  - how many times has a method been called user
  - analytics: what are peak times? what is average load? who is top user?

© luv2code LLC

AOP: Advantages and Disadvantages

Advantages

- Reusable modules
- Resolve code tangling
- Resolve code scatter
- Applied selectively based on configuration

Disadvantages

- Too many aspects and app flow is hard to follow
- Minor performance cost for aspect execution (run-time weaving)

© luv2code LLC

Aspect-Oriented Programming (AOP)
Spring AOP Support

© luv2code LLC

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

© luv2code LLC

Advice Types

- **Before advice**: run before the method
- **After finally advice**: run after the method (finally)
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **Around advice**: run before and after method

© luv2code LLC

Weaving

- Connecting aspects to target objects to create an advised object

- Different types of weaving
  - Compile-time, load-time or run-time

- Regarding performance: run-time weaving is the slowest

© luv2code LLC

AOP Frameworks

- Two leading AOP Frameworks for Java

Spring AOP

AspectJ

© luv2code LLC

Spring AOP Support

- Spring provides AOP support
- Key component of Spring
  - Security, transactions, caching etc
- Uses run-time weaving of aspects

© luv2code LLC

AspectJ

- Original AOP framework, released in 2001
  - www.eclipse.org/aspectj
- Provides complete support for AOP
- Rich support for
  - join points: method-level, constructor, field
  - code weaving: compile-time, post compile-time and load-time

© luv2code LLC

Spring AOP Comparison

Advantages

- Simpler to use than AspectJ
- Uses Proxy pattern
- Can migrate to AspectJ when using @Aspect annotation

Disadvantages

- Only supports method-level join points
- Can only apply aspects to beans created by Spring app context
- Minor performance cost for aspect execution (run-time weaving)

© luv2code LLC

AspectJ Comparison

Advantages

- Support all join points
- Works with any POJO, not just beans from app context
- Faster performance compared to Spring AOP
- Complete AOP support

Disadvantages

- Compile-time weaving requires extra compilation step
- AspectJ pointcut syntax can become complex

© luv2code LLC

Comparing Spring AOP and AspectJ

- Spring AOP only supports
  - Method-level join points
  - Run-time code weaving (slower than AspectJ)

- AspectJ supports
  - join points: method-level, constructor, field
  - weaving: compile-time, post compile-time and load-time

© luv2code LLC

Comparing Spring AOP and AspectJ

- Spring AOP is a light implementation of AOP
- Solves common problems in enterprise applications

- My recommendation
  - Start with Spring AOP … easy to get started with
  - If you have complex requirements then move to AspectJ

© luv2code LLC

Additional Resources

- Spring Reference Manual: www.spring.io

- *AspectJ in Action*
  - by Raminvas Laddad

- *Aspect-Oriented Development with Use Cases*
  - by Ivar Jacobson and Pan-Wei Ng

© luv2code LLC

Aspect-Oriented Programming (AOP)
@Before Advice

Advice Types

- **Before advice**: run before the method
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **After finally advice**: run after the method (finally)
- **Around advice**: run before and after method

@Before Advice - Interaction

```
MainApp

// call target object
targetObj.doSomeStuff();
```

```
TargetObject

public void doSomeStuff() {
  ...
}
```

Advice - Interaction

@Before

```
TargetObject

public void doSomeStuff() {
  …
}
```

Advice - Interaction

@Before

```
TargetObject

public void doSomeStuff() {
  …
}
```

@AfterReturning

@Before Advice - Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **API Management**
  - how many times has a method been called user
  - analytics: what are peak times? what is average load? who is top user?

AOP Example - Overview

Step-By-Step

```
MainApp

// call target object
theAccountDAO.addAccount();
```

```
TargetObject - AccountDAO

public void addAccount() {
  ...
}
```

Spring Boot AOP Starter

- Add the dependency for Spring Boot AOP Starter

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

- Since this dependency is part of our pom.xml
  - Spring Boot will **automatically** enable support for AOP
  - No need to explicitly use @EnableAspectJAutoProxy … we get it for free

Development Process - @Before

Step-By-Step

1. Create target object: AccountDAO
2. Create main app
3. Create an Aspect with @Before advice

Step 1: Create Target Object: AccountDAO

```
public interface AccountDAO {

    void addAccount() {
}
```

```
@Component
public class AccountDAOImpl implements AccountDAO {

    public void addAccount() {

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

    }
}
```

Step 2: Create main app

```java
@SpringBootApplication
public class AopdemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(AopdemoApplication.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(AccountDAO theAccountDAO) {

      return runner -> {

         demoTheBeforeAdvice(theAccountDAO);
      };
   }

   private void demoTheBeforeAdvice(AccountDAO theAccountDAO) {

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

}
```

Step 3: Create an Aspect with @Before advice

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    ...

}
```

Step 3: Create an Aspect with @Before advice

```
@Aspect
@Component
public class MyDemoLoggingAspect {

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

      ...

   }

}
```

Step 3: Create an Aspect with @Before advice

```
@Aspect
@Component
public class MyDemoLoggingAspect {

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

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

   }

}
```

Best Practices: 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

AOP - Pointcut Expressions

I made a promise to you …

```
@Component
public class MyDemoLoggingAspect {

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

    // let's start with an @Before advice

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

}
```

Pointcut expression

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

AOP Terminology

- **Pointcut**: A predicate expression for where advice should be applied

Pointcut Expression Language

- Spring AOP uses AspectJ's pointcut expression language

- We will start with execution pointcuts
  - Applies to execution of methods

Match on Method Name

Pointcut Expression Language

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

- The pattern is optional if it has “?”

Pointcut Expression Examples

Match on method names

- Match only **addAccount()** method in **AccountDAO** class

Modifier

Return type

```
@Before("execution(public void com.luv2code.aopdemo.dao.AccountDAO.addAccount())")
```

Declaring

Method

Pointcut Expression Examples

Match on method names

- Match any **addAccount()** method in **any** class

Modifier

Return type

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

Method

Pointcut Expression Examples

Match on method names (using wildcards)

- Match methods **starting** with **add** in any class

Modifier

Return type

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

Method

Pointcut Expression Examples

Match on method names (using wildcards)

- Match methods starting with **processCreditCard** in any class

Modifier

Return type

```
@Before("execution(public VerificationResult processCreditCard*())”)
```

Method

Pointcut Expression Examples

- Use wildcards on return type

Return type

```
@Before("execution(public * processCreditCard*())")
```

Method

Pointcut Expression Examples

- Modifier is optional … so you don't have to list it

Return type

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

Method

AOP - Pointcut Expressions (cont)

Match on Parameters

Parameter Pattern Wildcards

- 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

Pointcut Expression Examples

Match on method <u>parameters</u>

- Match **addAccount** methods with **no arguments**

Pointcut Expression Examples

Match on method parameters

- Match **addAccount** methods with **no arguments**

Return type

Param no-args

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

Method

Pointcut Expression Examples

Match on method parameters

- Match **addAccount** methods that have **Account** param

Return type

Param Type

```
@Before("execution(* addAccount(com.luv2code.aopdemo.Account))")
```

Method

Pointcut Expression Examples

Match on method <u>parameters</u> (using wildcards)

- Match **addAccount** methods with **any number of arguments**

Pointcut Expression Examples

Match on method parameters (using wildcards)

- Match **addAccount** methods with **any number of arguments**

Return type

Param Type

```
@Before("execution(* addAccount(..))”)
```

Method

Match on Package

Package - Pointcut Expression Examples

Match on methods in a <u>package</u>

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

Aspect-Oriented Programming (AOP)
Pointcut Declarations

Problem

- How can we reuse a pointcut expression?
  - Want to apply to multiple advices

```
@Before("execution(* com.luv2code.aopdemo.dao.*.*(..))")
public void beforeAddAccountAdvice() {
    ...
}
```

Possible Solution

- Could just do the old copy/paste method … yuk!

```
@Before("execution(* com.luv2code.aopdemo.dao.*.*(..))")
public void beforeAddAccountAdvice() {
    ...
}
```

Possible Solution

- Could just do the old copy/paste method … yuk!

```
@Before("execution(* com.luv2code.aopdemo.dao.*.*(..))")
public void beforeAddAccountAdvice() {
    ...
}

@Before("execution(* com.luv2code.aopdemo.dao.*.*(..))")
public void performApiAnalytics() {
    ...
}
```

Ideal Solution

- **Create a pointcut declaration once**
- **Apply it to multiple advices**

Development Process

Step-By-Step

1. Create a pointcut declaration

2. Apply pointcut declaration to advice(s)

Step 1- Create Pointcut Declaration

- Create a pointcut declaration once

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
private void forDaoPackage() {}
```

Step 1- Create Pointcut Declaration

```
@Aspect
@Component
public class MyDemoLoggingAspect {



}
```

Step 1- Create Pointcut Declaration

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    @Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
    private void forDaoPackage() {}



}
```

Step 2 - Apply to Multiple Advices

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    @Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
    private void forDaoPackage() {}

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





}
```

Step 2 - Apply to Multiple Advices

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    @Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
    private void forDaoPackage() {}

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

    @Before("forDaoPackage()")
    public void performApiAnalytics() {
        ...
    }

}
```

Benefits of Pointcut Declarations

- **Easily reuse pointcut expressions**
- **Update pointcut in one location**
- **Can also share and combine pointcut expressions (coming up later)**

Aspect-Oriented Programming (AOP)
Combine Pointcuts

Problem

- How to apply multiple pointcut expressions to single advice?
- Execute an advice only if certain conditions are met
- For example
  - All methods in a package EXCEPT getter/setter methods

Combining Pointcut Expressions

- **Combine pointcut expressions using logic operators**
  - **AND (&&)**
  - **OR (| |)**
  - **NOT (!)**

Combining Pointcut Expressions

- Works like an "if" statement
- Execution happens only if it evaluates to true

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

@Before("expressionOne() || expressionTwo()")

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

Example:

- **All methods in a package EXCEPT getter/setter methods**

Development Process

Step-By-Step

1. Create a pointcut declarations

2. Combine pointcut declarations

3. Apply pointcut declaration to advice(s)

Step 1- Create Pointcut Declaration

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
private void forDaoPackage() {}
```

Step 1- Create Pointcut Declaration

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
private void forDaoPackage() {}

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

Step 1- Create Pointcut Declaration

```
@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() {}
```

Step 2 - Combine 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() {}

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

Step 3 - Apply Pointcut Declaration to Advice(s)

```
...

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

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

Aspect-Oriented Programming (AOP)
Control Aspect Order

Problem

- How to control the order of advices being applied?

Problem

- How to control the order of advices being applied?

The order is undefined

To Control Order

- **Refactor: Place advices in separate Aspects**
- **Control order on Aspects using the @Order annotation**
- **Guarantees order of when Aspects are applied**

Development Process

Step-By-Step

1. Refactor: Place advices in separate Aspects

2. Add @Order annotation to Aspects

Step 1 - Refactor: Place advices in separate Aspects

Step 2 - Add @Order annotation

- 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

- We want the following order:

1. MyCloudLogAspect
2. MyLoggingDemoAspect
3. MyApiAnalyticsAspect

@Order annotation

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

```
@Aspect
@Component
@Order(2)
public class MyLoggingDemoAspect {
 ...
}
```

```
@Aspect
@Component
@Order(3)
public class MyApiAnalyticsAspect {
 ...
}
```

@Order annotation

- Lower numbers have higher precedence
  - Range: Integer.MIN_VALUE to Integer.MAX_VALUE
  - Negative numbers are allowed
  - Does not have to be consecutive

@Order annotation

- Example with negative numbers

@Order annotation

- **FAQ: What if aspects have the exact same @Order annotation?**

```
@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

Will still run AFTER MyCloudLogAspect and BEFORE MyLoggingDemoAspect

Aspect-Oriented Programming (AOP)
Reading Method Arguments with JoinPoints

Problem

- **When we are in an aspect (ie for logging)**
  - **How can we access method parameters?**

Development Process

Step-By-Step

1. Access and display Method Signature

2. Access and display Method Arguments

Step 1 - Access and display Method Signature

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



}
```

Step 1 - Access and display Method Signature

```
@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)
```

Step 2 - Access and display Method 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
```

Aspect-Oriented Programming (AOP) Progress Check

Progress Check

Work Completed

- @Before advice type
- Pointcut Expressions
- JoinPoints

Work To Do - More Advice Types

- @AfterReturning
- @AfterThrowing
- @After
- @Around

Aspect-Oriented Programming (AOP)
@AfterReturning Advice

Advice Types

- **Before advice**: run before the method
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **After finally advice**: run after the method (finally)
- **Around advice**: run before and after method

@AfterReturning Advice - Interaction

Advice - Interaction

```
TargetObject

public void doSomeStuff() {
  ...
}
```

@Before

@AfterReturning

Successful execution
(no exceptions)

@AfterReturning Advice - Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **Post-processing Data**
  - Post process the data before returning to caller
  - Format the data or enrich the data (really cool but be careful)

Example

- Create an advice that will run after a method call (success execution)

@AfterReturning Advice

- This advice will run after the method call (success execution)

```
@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 method called

Access the Return Value

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





}
```

Access the Return Value

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




}
```

Access the Return Value

```
@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);

}
```

Best Practices: 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

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

Aspect-Oriented Programming (AOP)
@AfterReturning Advice - Modify Return Value

@AfterReturning Advice - Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **Post-processing Data**
  - Post process the data before returning to caller
  - Format the data or enrich the data (really cool but be careful)

Post-Process / Modify Data

Modify the Return Value

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

  // modify “result” list: add, remove, update, etc …




}
```

Modify the Return Value

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

  // modify “result” list: add, remove, update, etc …
  if (!result.isEmpty()) {

      Account tempAccount = result.get(0);

      tempAccount.setName("Daffy Duck");
  }
}
```

Post-Process / Modify Data

Calling program

```
// call method to find the accounts
List<Account> theAccounts = theAccountDAO.findAccounts();
```

Calling program will get the modified result

Aspect-Oriented Programming (AOP)
@AfterThrowing Advice

Advice Types

- **Before advice**: run before the method
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **After finally advice**: run after the method (finally)
- **Around advice**: run before and after method

@AfterThrowing Advice - Interaction

Advice - Interaction

Sequence Diagram

@AfterThrowing Advice - Use Cases

- Log the exception
- Perform auditing on the exception
- Notify DevOps team via email or SMS
- Encapsulate this functionality in AOP aspect for easy reuse

Example

- Create an advice that will run after an exception is thrown

@AfterThrowing Advice

- This advice will run after an exception is thrown

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

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

}
```

Access the Exception object

- Need to access the exception object

Access the Exception

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





}
```

Access the Exception

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




}
```

Access the Exception

```
@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);

}
```

Exception Propagation

- At this point, we are only intercepting the exception (reading it)
- However, the exception is still propagated to calling program

Exception Propagation

- If you want to stop the exception propagation
  - then use the **@Around** advice
  - covered in later videos

Development Process - @AfterThrowing

Step-By-Step

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