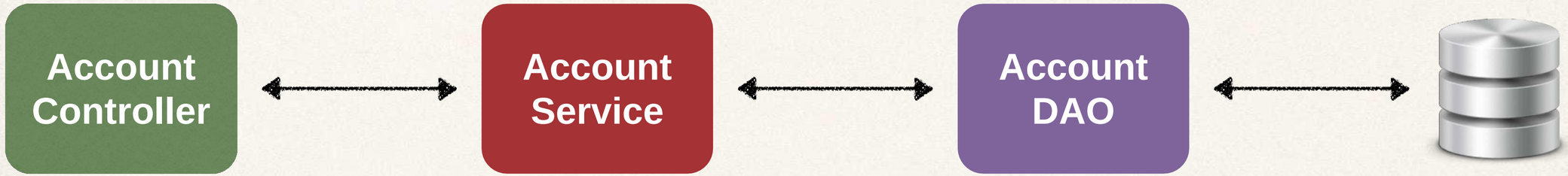
Aspect-Oriented Programming (AOP) Overview

Application Architecture



Code for Data Access Object (DAO)

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

        entityManager.persist(theAccount);
}
```

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!

DAO - Add Logging Code

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


    entityManager.persist(theAccount);


}
```

DAO - Add Logging Code

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

    [redacted]

    entityManager.persist(theAccount);

}
```

New Requirement - Security

From: The Boss

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

Add Security Code

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

    entityManager.persist(theAccount);

}
```

Add Security Code

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


    entityManager.persist(theAccount);


}
```

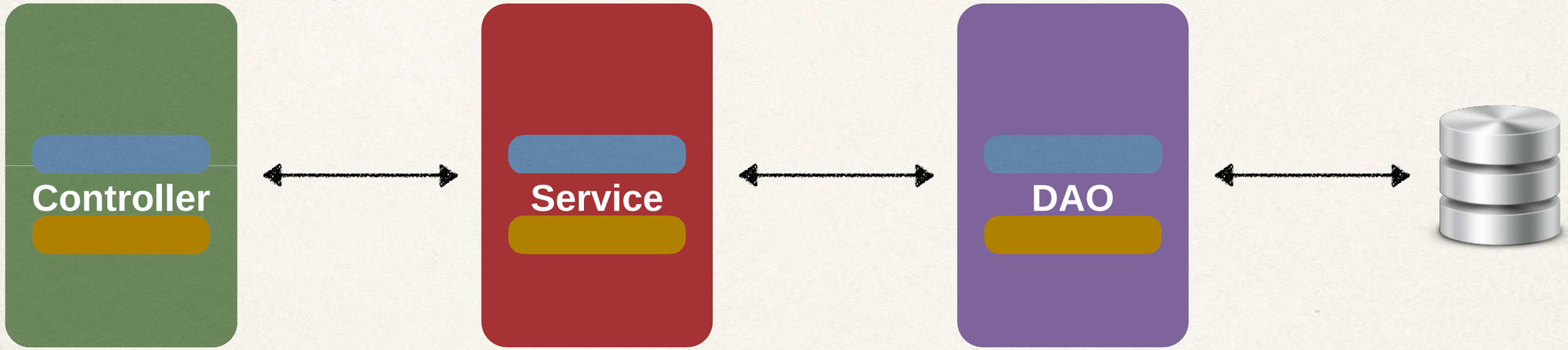
Add Security Code

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

    

    entityManager.persist(theAccount);

}
```

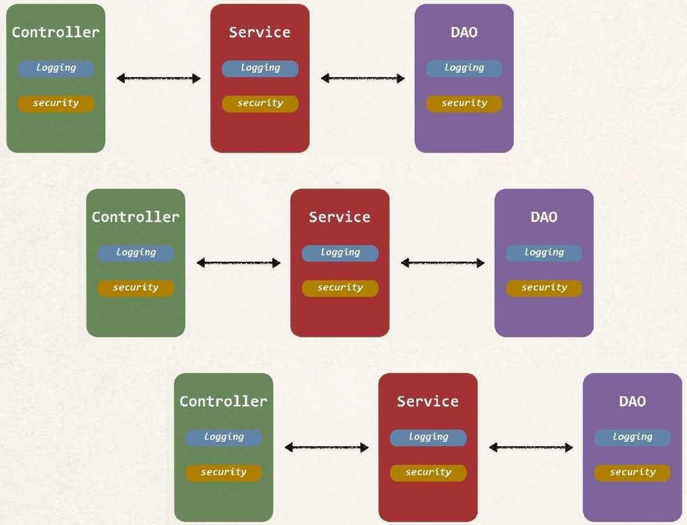
By the way

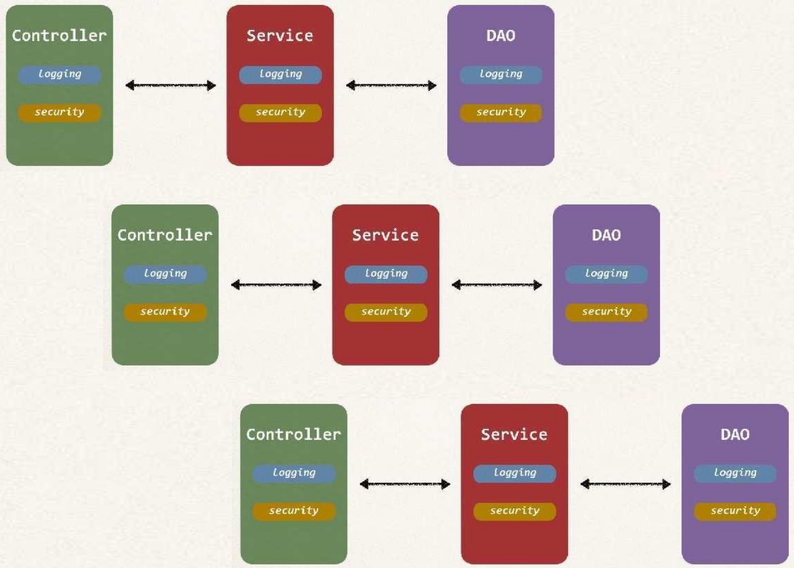
From: The Boss

- Let's add it to all of our layers…



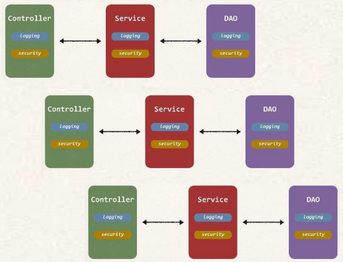
I'm Going Crazy Over Here



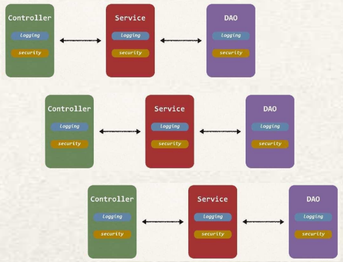


Two Main Problems

- **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



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

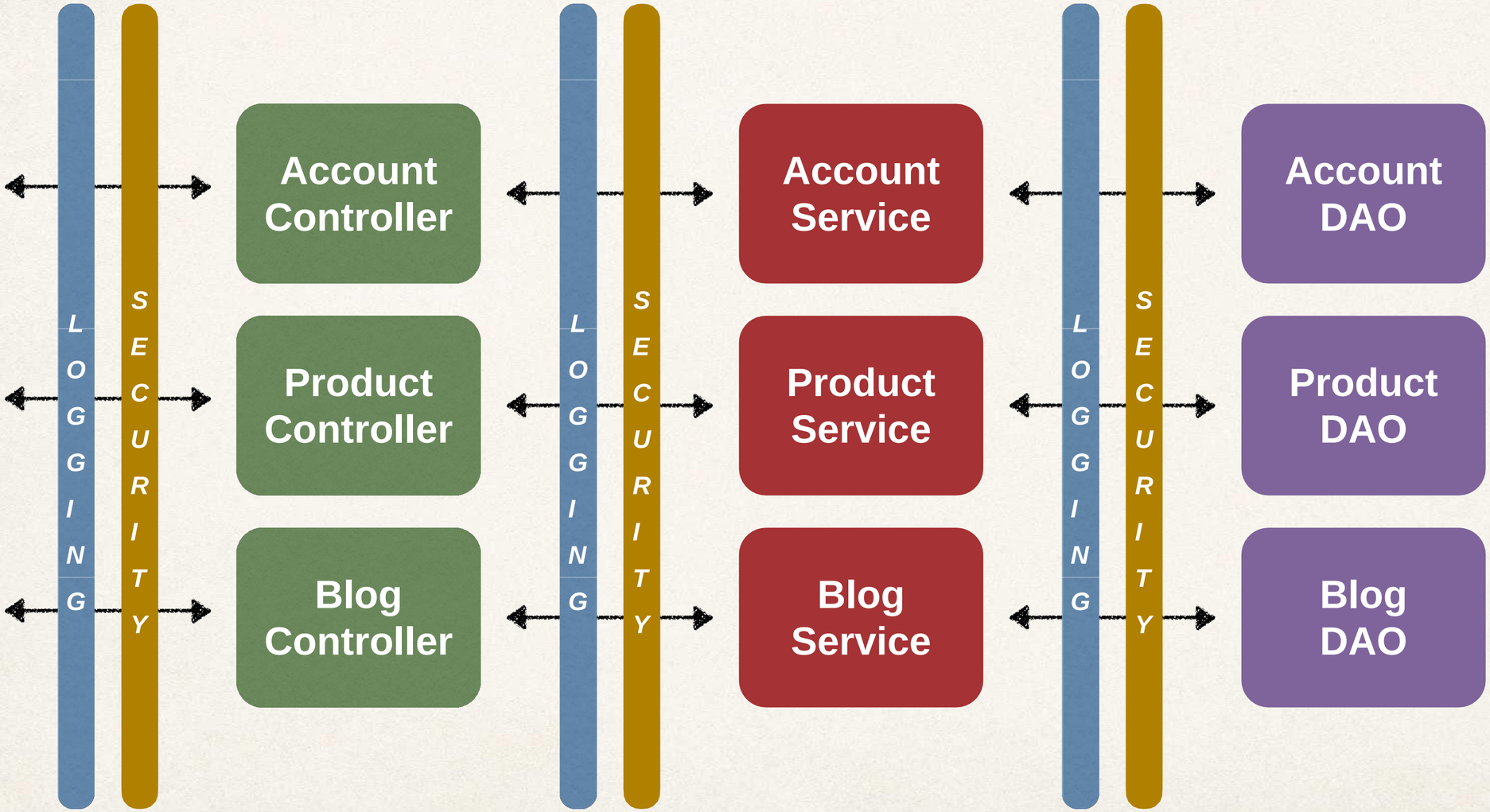
Aspect-Oriented Programming

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

Cross-Cutting Concerns

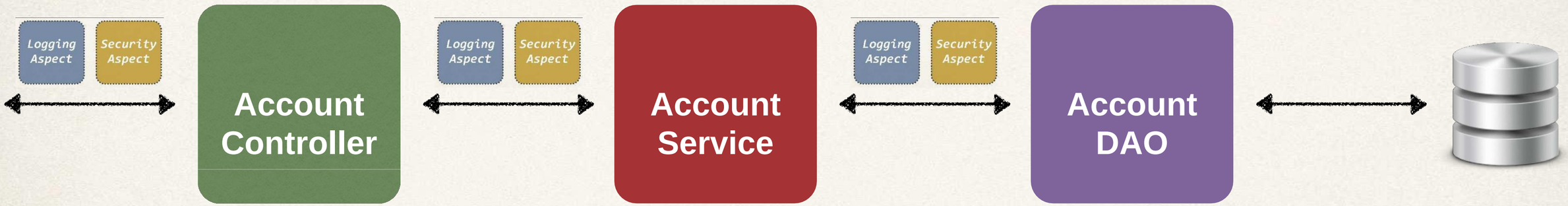
- “Concern” means logic / functionality

Cross-Cutting Concerns



Aspects

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



AOP Solution

- Apply the Proxy design pattern

MainApp

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

TargetObject

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

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!

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?

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)

Aspect-Oriented Programming (AOP)
Spring AOP Support

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 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

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

AOP Frameworks

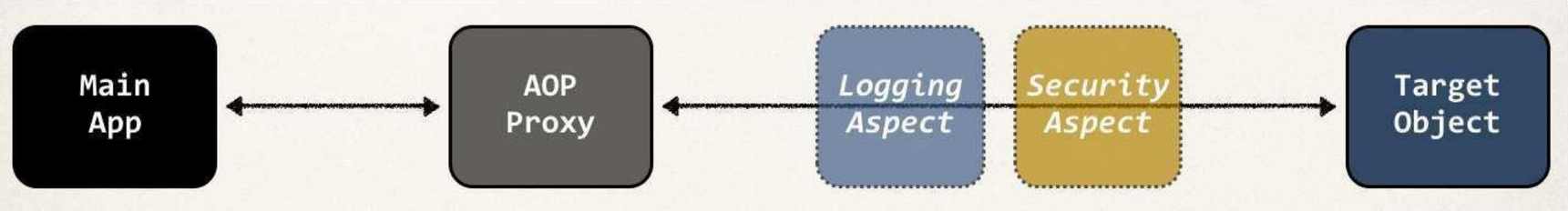
- Two leading AOP Frameworks for Java

Spring AOP

AspectJ

Spring AOP Support

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



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

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)

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

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

Comparing Spring AOP and AspectJ

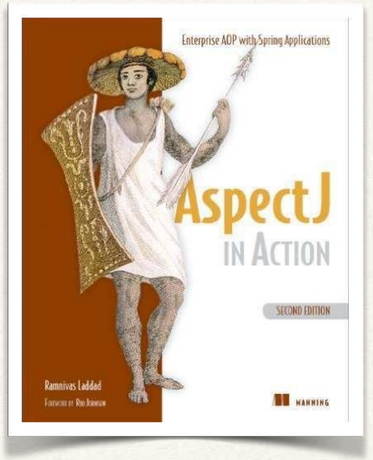
- Spring AOP is a light implementation of AOP
- Solves common problems in enterprise applications

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

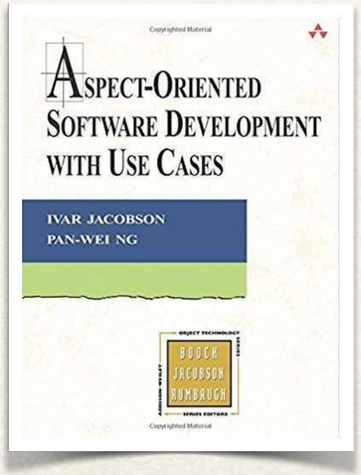
Additional Resources

- 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



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

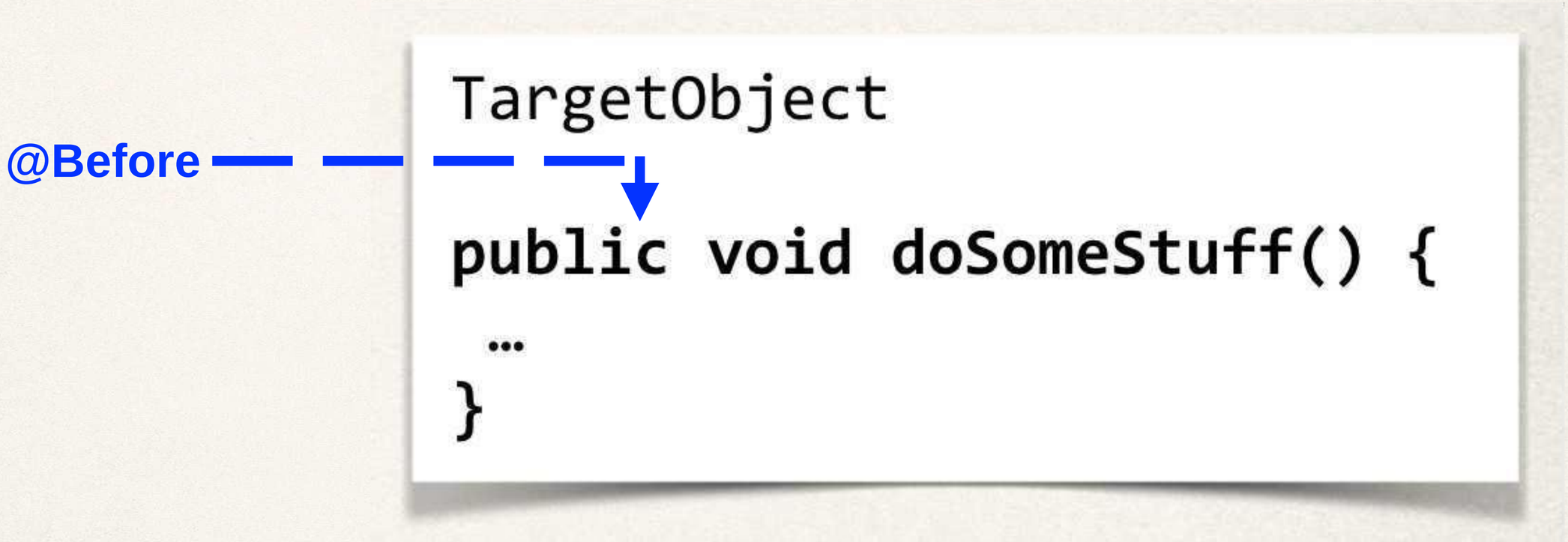
MainApp

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

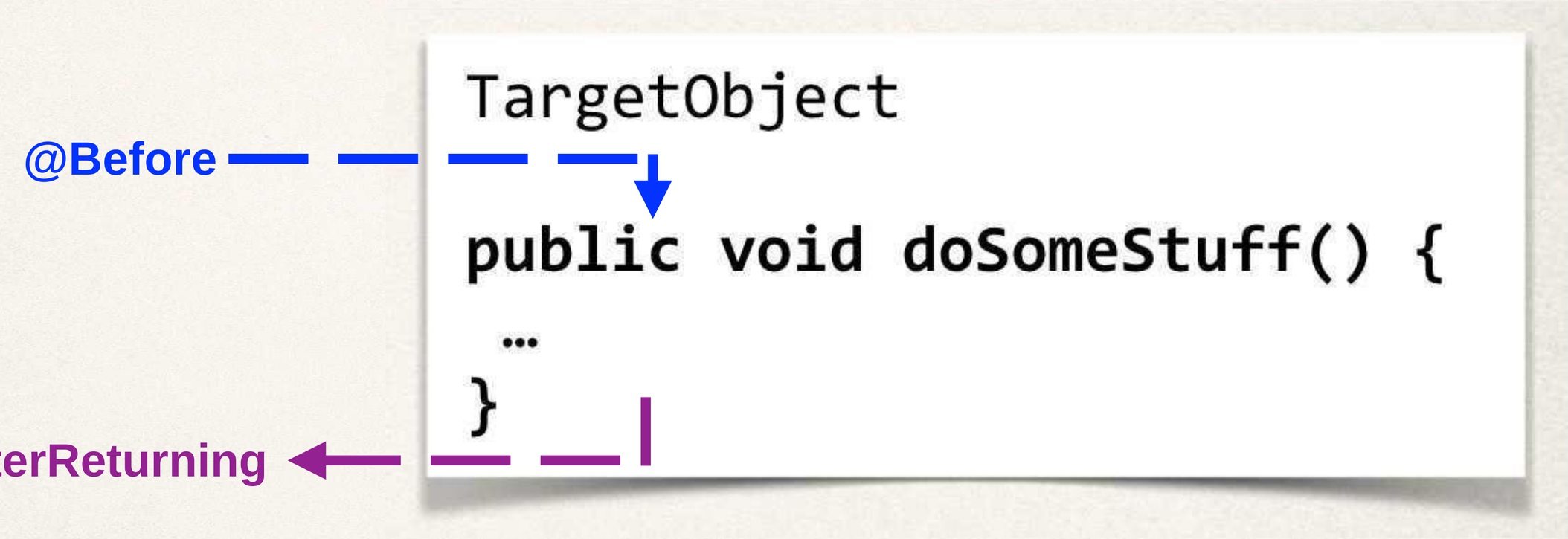
TargetObject

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

Advice - Interaction



Advice - Interaction

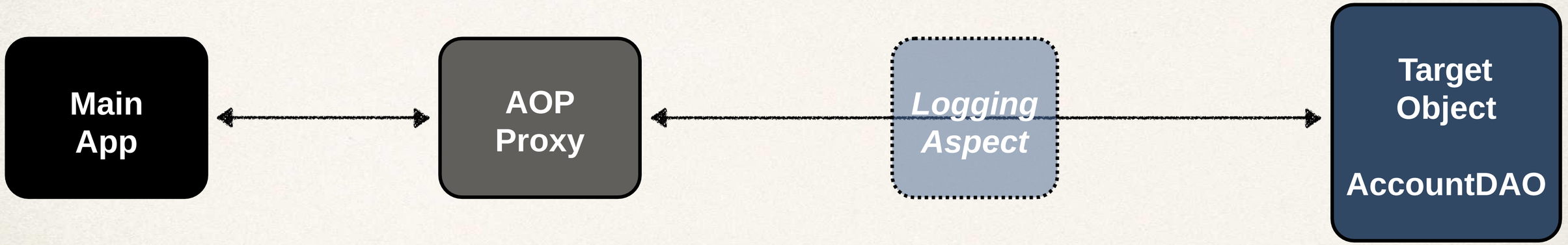


@Before Advice - Use Cases

- **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

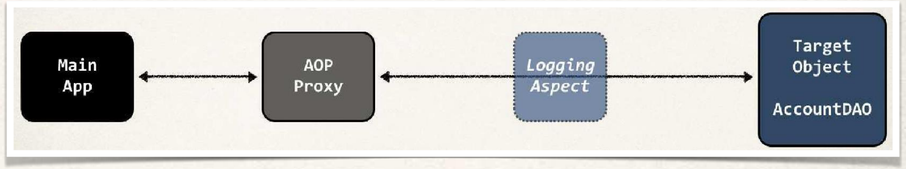
Development Process - @Before

Step-By-Step

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

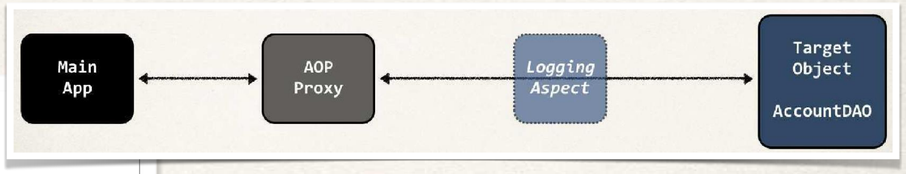
Step 1: Create Target Object: AccountDAO

```
public interface AccountDAO {

    void addAccount() {
}
```




```
@Component
public class AccountDAOImpl implements AccountDAO {

    public void addAccount() {

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

    }
}
```

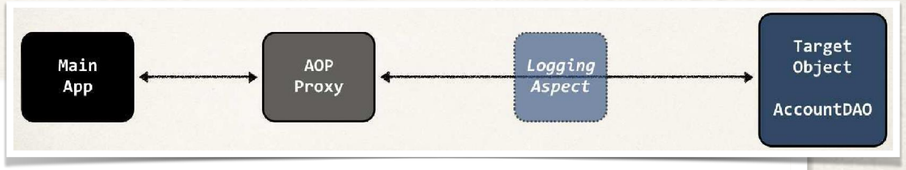
Step 2: Create main app




```
@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();
   }

}
```

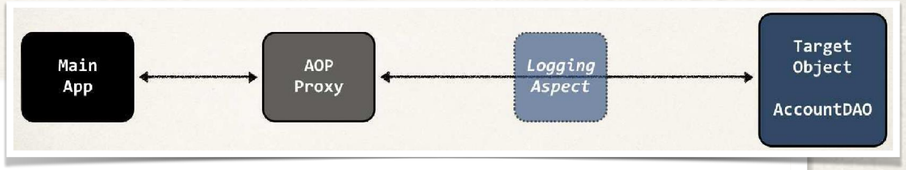
luv2code

Step 3: Create an Aspect with @Before advice



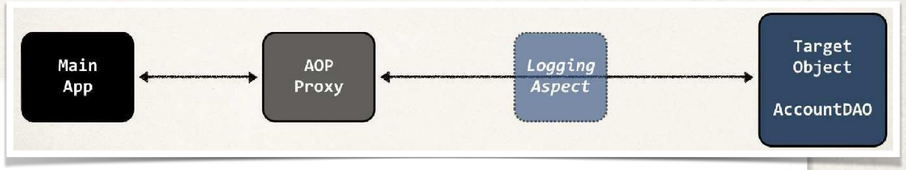

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    ...

}
```

Step 3: Create an Aspect with @Before advice



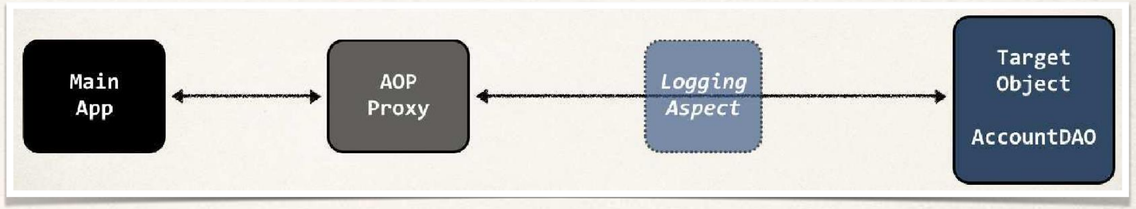

```
@Aspect
@Component
public class MyDemoLoggingAspect {

   @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()");

   }

}
```

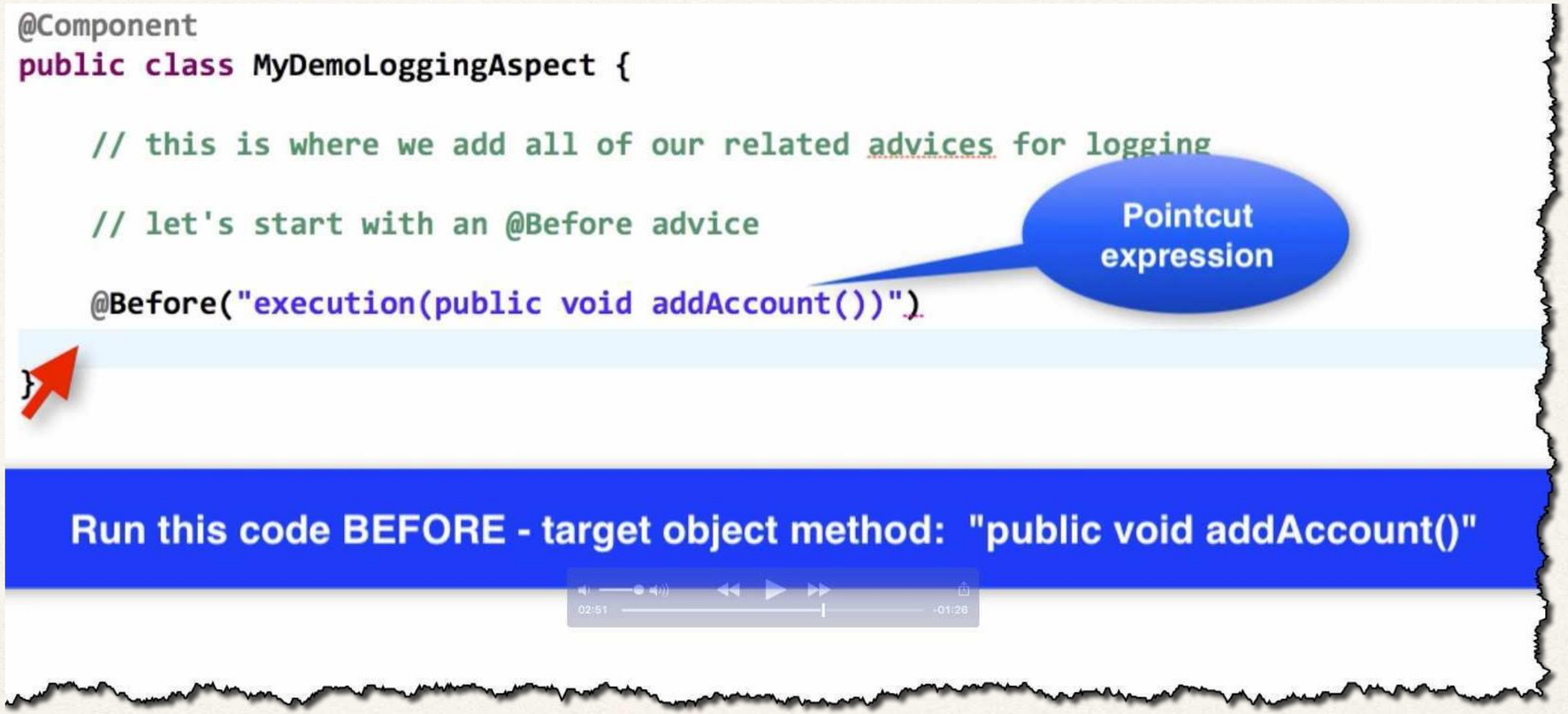
Best Practices: Aspect and Advices



- 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 ...



luv2code

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

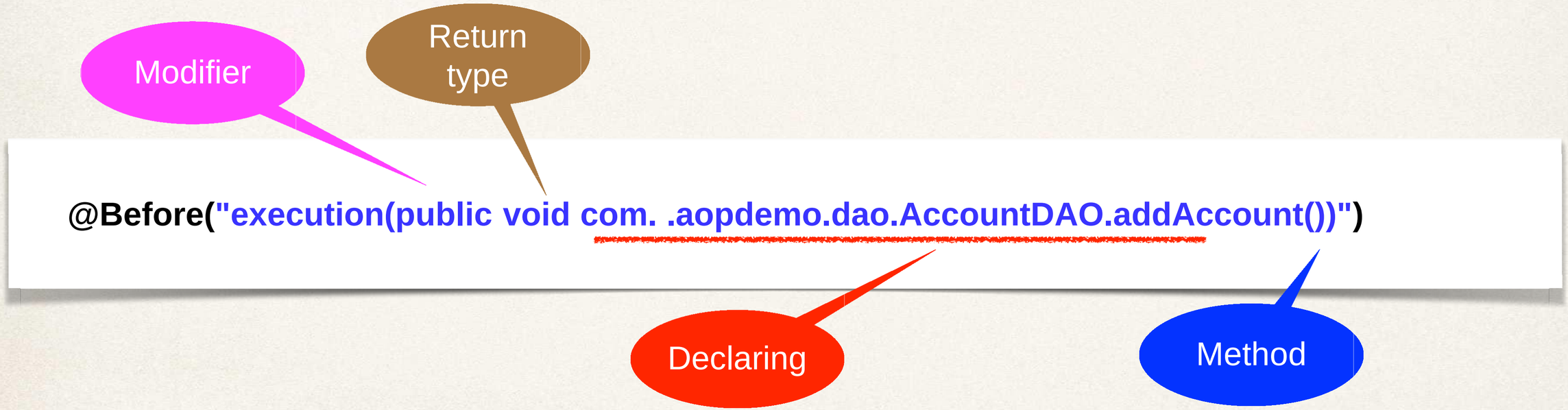
```
execution(modifiers-pattern? return-type-pattern declaring-type-pattern?
          method-name-pattern(param-pattern) throws-pattern?)
```

- The pattern is optional if it has “?”

Pointcut Expression Examples

Match on method names

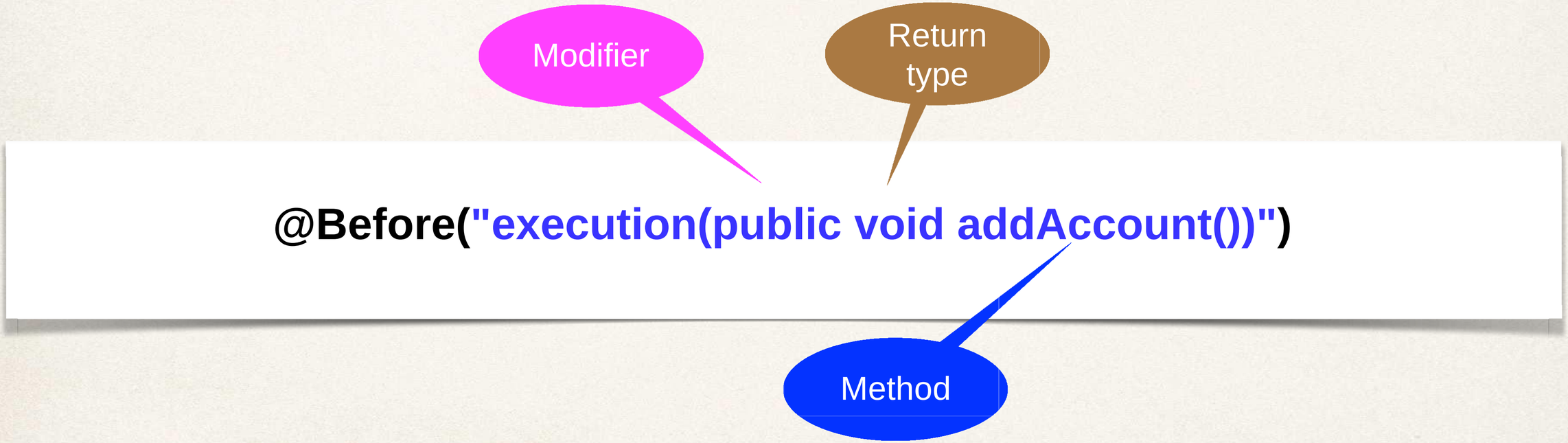
- Match only **addAccount()** method in **AccountDAO** class



Pointcut Expression Examples

Match on method names

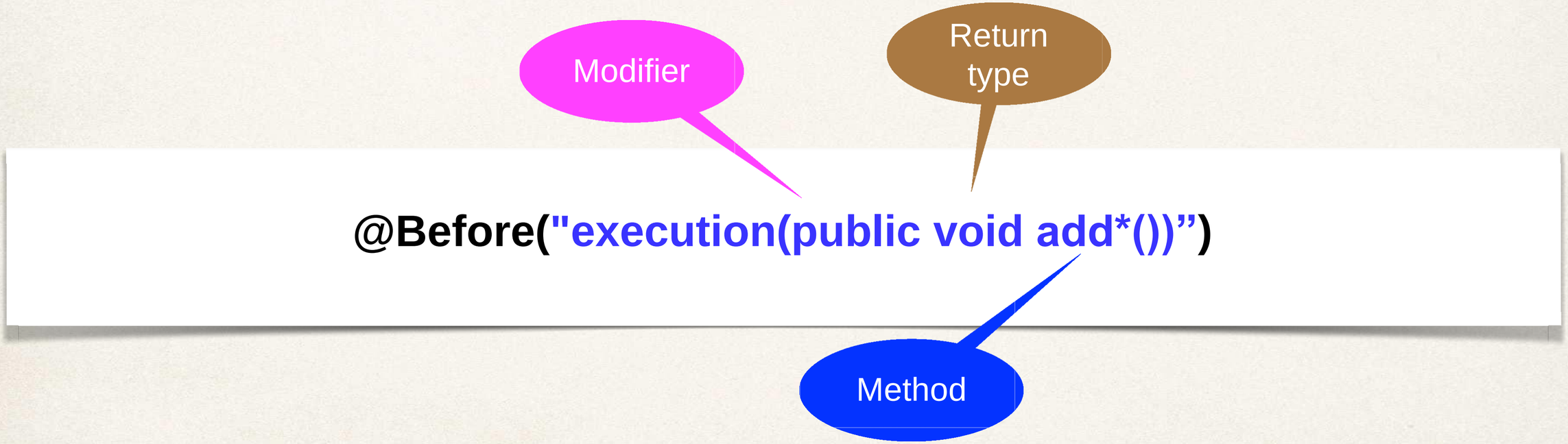
- Match any **addAccount()** method in **any** class



Pointcut Expression Examples

Match on method names (using wildcards)

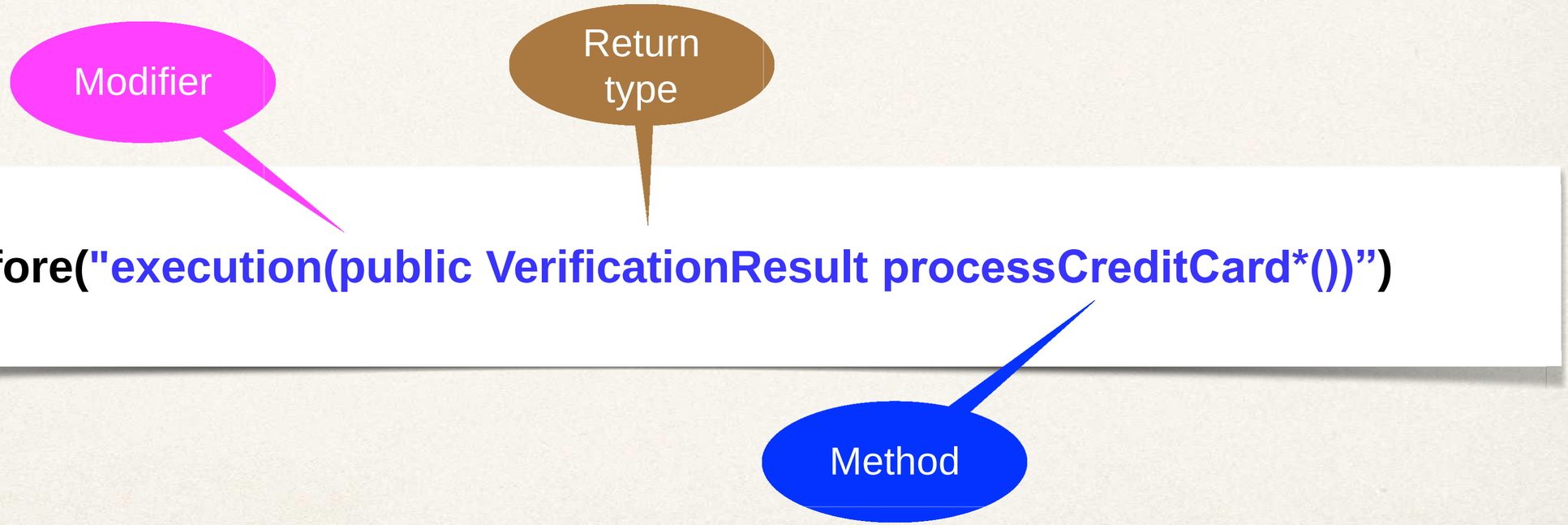
- Match methods **starting** with **add** in any class



Pointcut Expression Examples

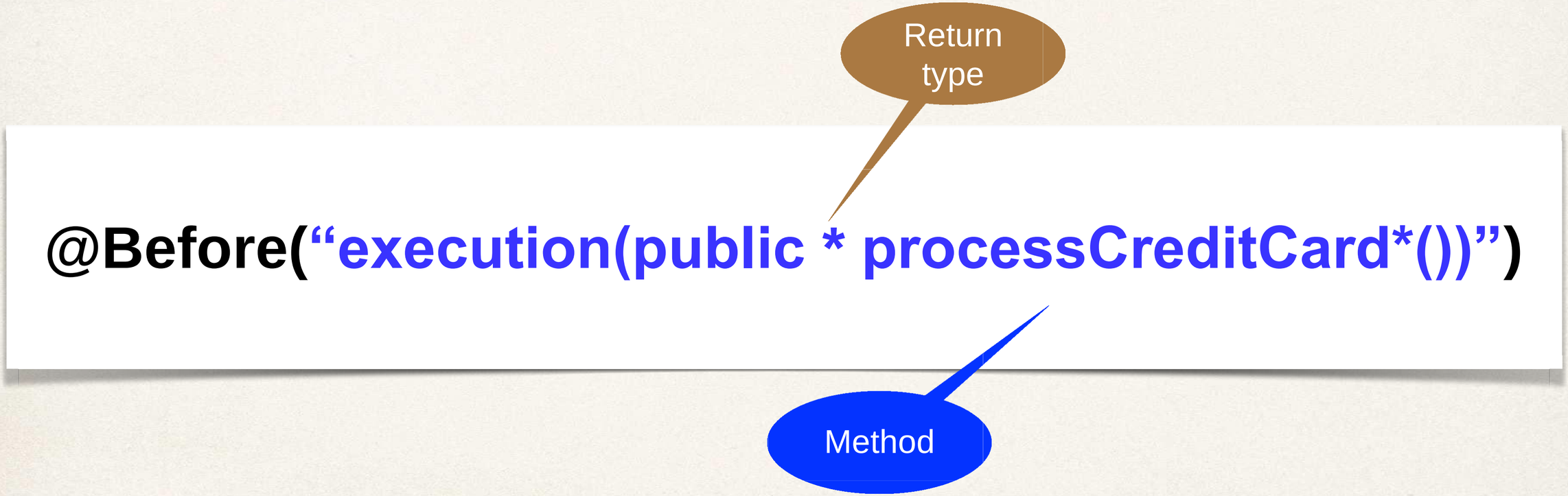
Match on method names (using wildcards)

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



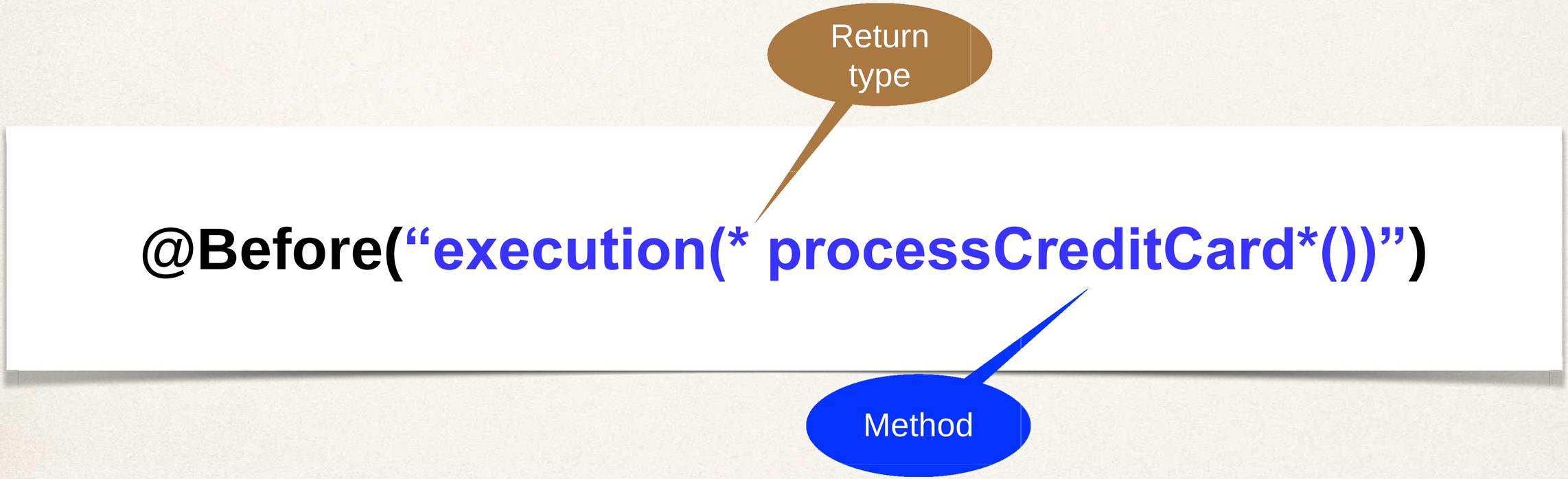
Pointcut Expression Examples

- Use wildcards on return type



Pointcut Expression Examples

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



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

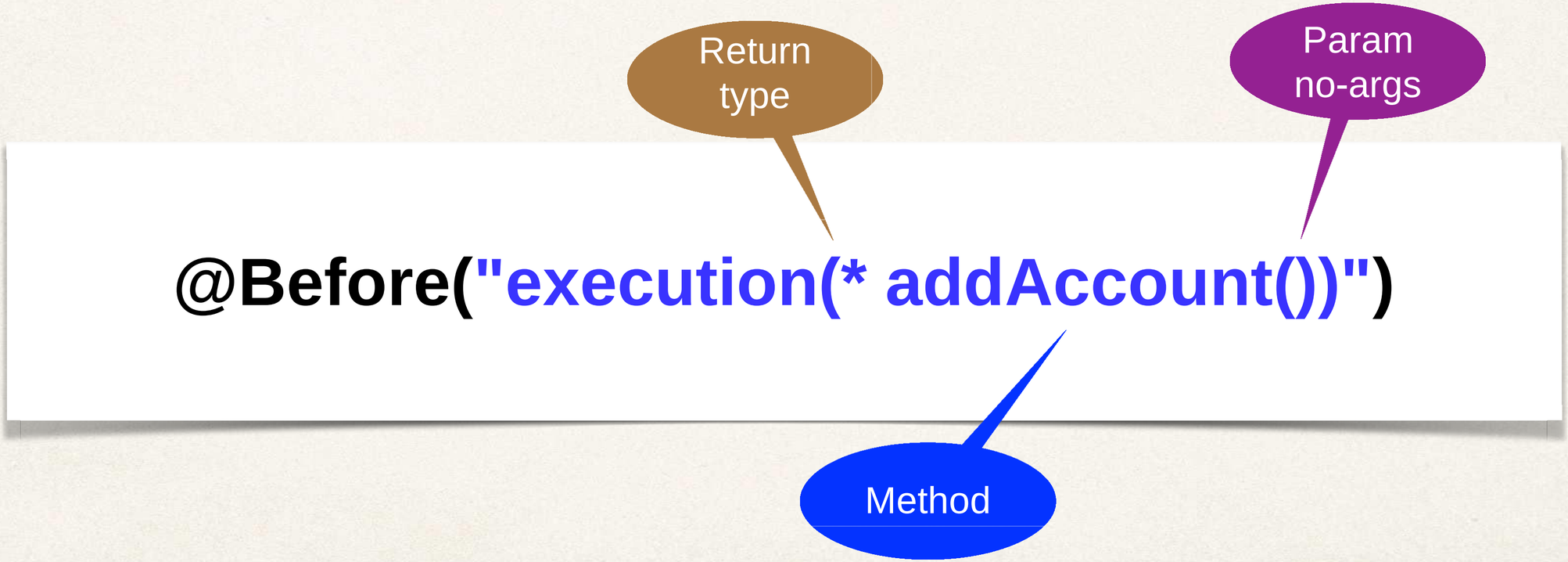
Match on method parameters

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

Pointcut Expression Examples

Match on method parameters

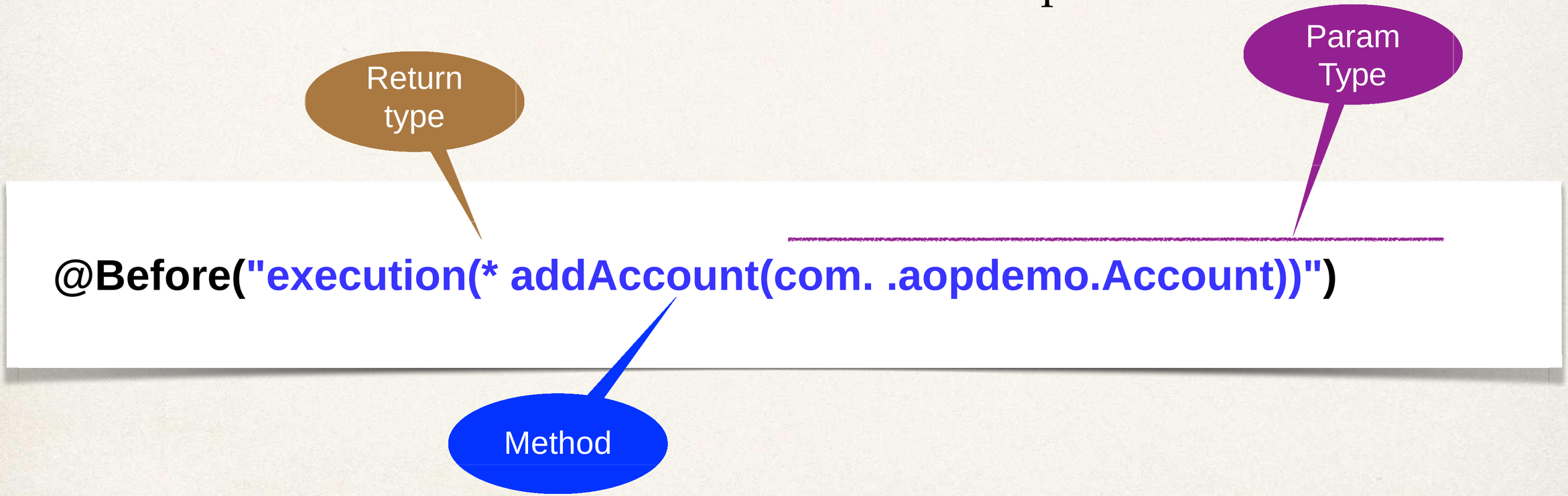
- Match **addAccount** methods with **no arguments**



Pointcut Expression Examples

Match on method parameters

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



Pointcut Expression Examples

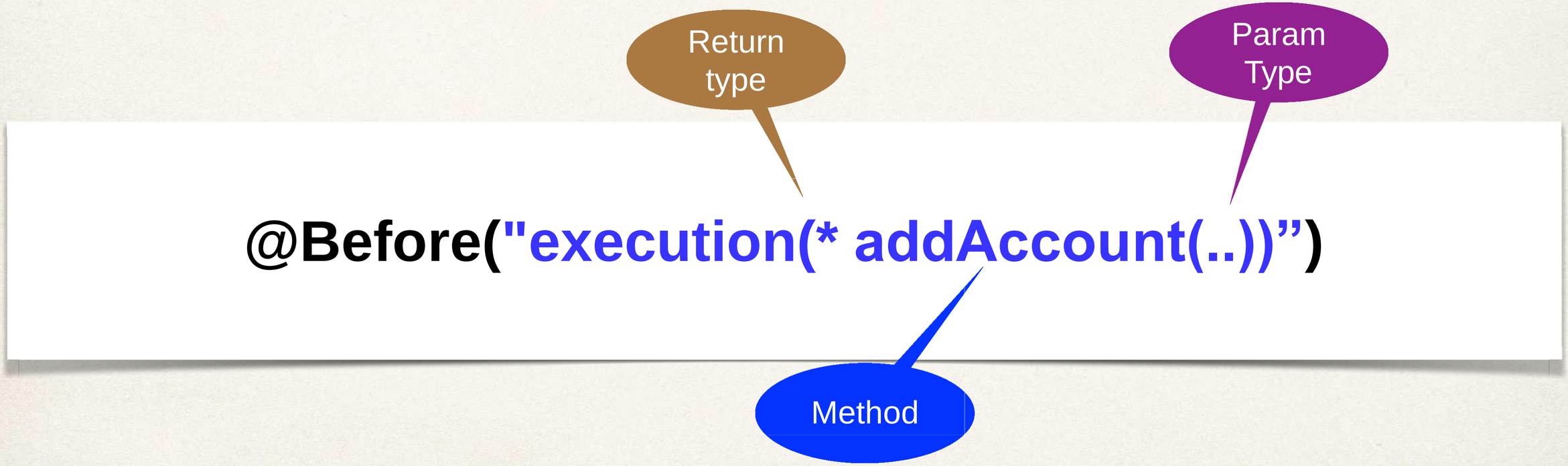
Match on method parameters (using wildcards)

- 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**

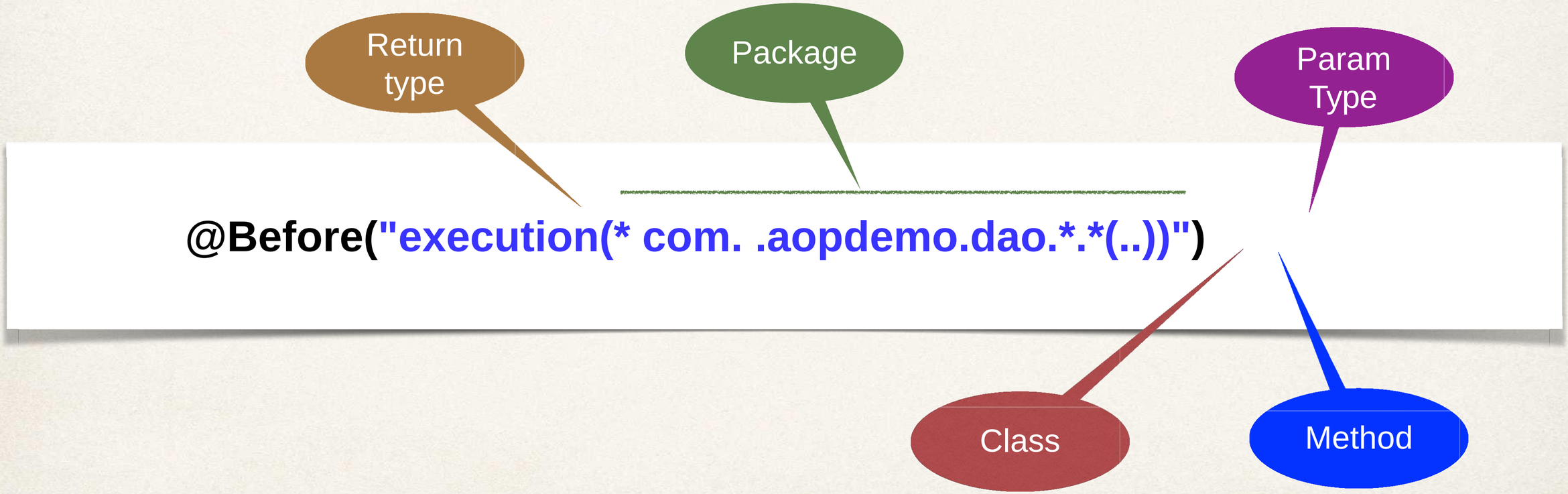


Match on Package

Package - Pointcut Expression Examples

Match on methods in a package

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



luv2code

If you are using IntelliJ Ultimate

You may encounter this error

```
Exception encountered during context initialization - cancelling
refresh attempt:
org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'mbeanExporter' defined in class path
resource [org/springframework/boot/autoconfigure/jmx/
JmxAutoConfiguration.class]
```

Why?

IntelliJ Ultimate loads additional classes for JMX

This conflicts with Spring Boot’s JMX Autoconfiguration

**When using wildcards with AOP,
caution should be taken.**

**If new frameworks are added to your project,
then you may encounter conflicts.**

Recommendation is to:
- narrow your pointcut expressions
- limit them to your project package

In this case, our pointcut expression is too broad.

We can resolve this by:
- narrowing the pointcut expression
- only match within our project package

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

Narrow pointcut expression to our package

Aspect-Oriented Programming (AOP)
Pointcut Declarations

Problem

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

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

Possible Solution

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

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

Possible Solution

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

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

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




}
```

Step 2 - Apply to Multiple Advices

```
@Aspect
@Component
public class MyDemoLoggingAspect {

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

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




}
```

Step 2 - Apply to Multiple Advices

```
@Aspect
@Component
public class MyDemoLoggingAspect {

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

Step 1- Create Pointcut Declaration

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

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

Step 1- Create Pointcut Declaration

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

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

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

Step 2 - Combine Pointcut Declarations

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

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

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

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

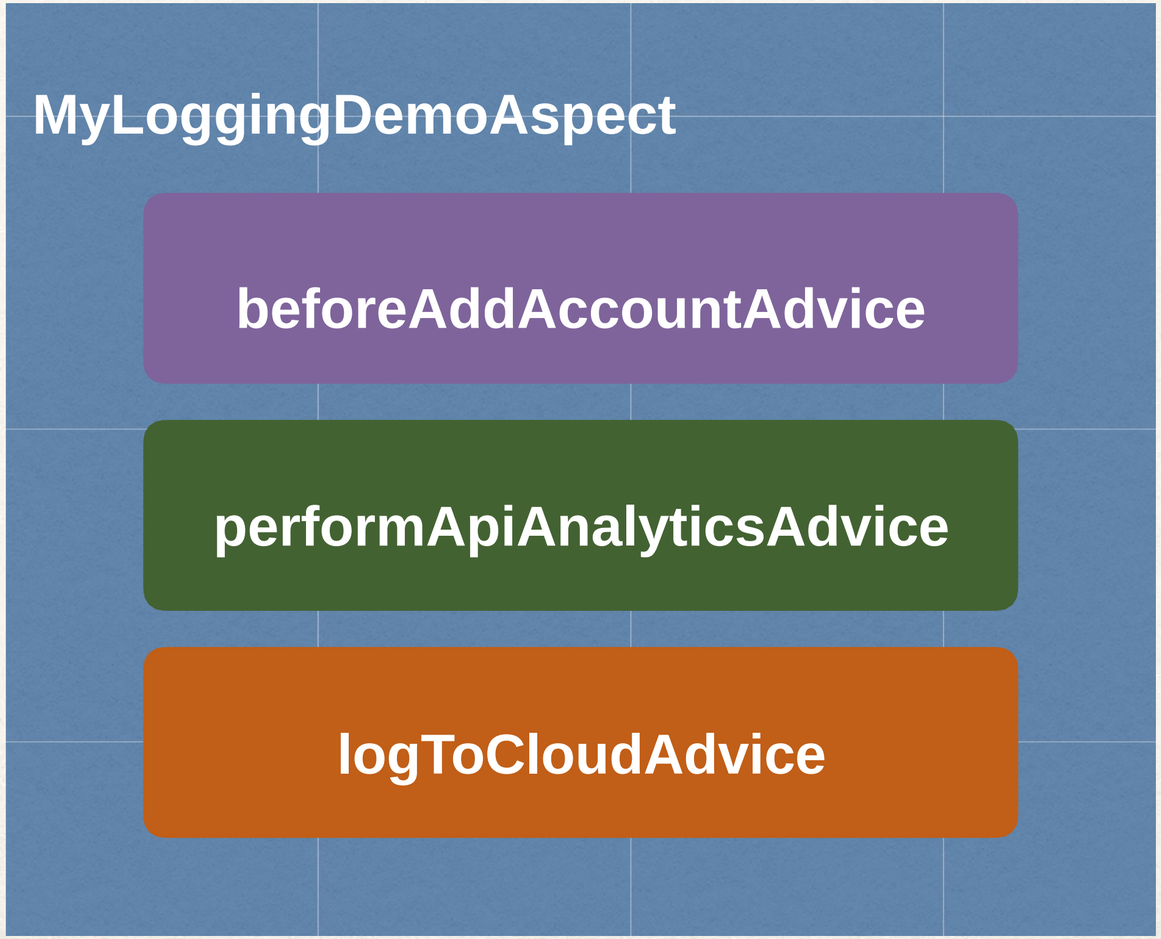
Step 3 - Apply Pointcut Declaration to Advice(s)

```
…

// 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

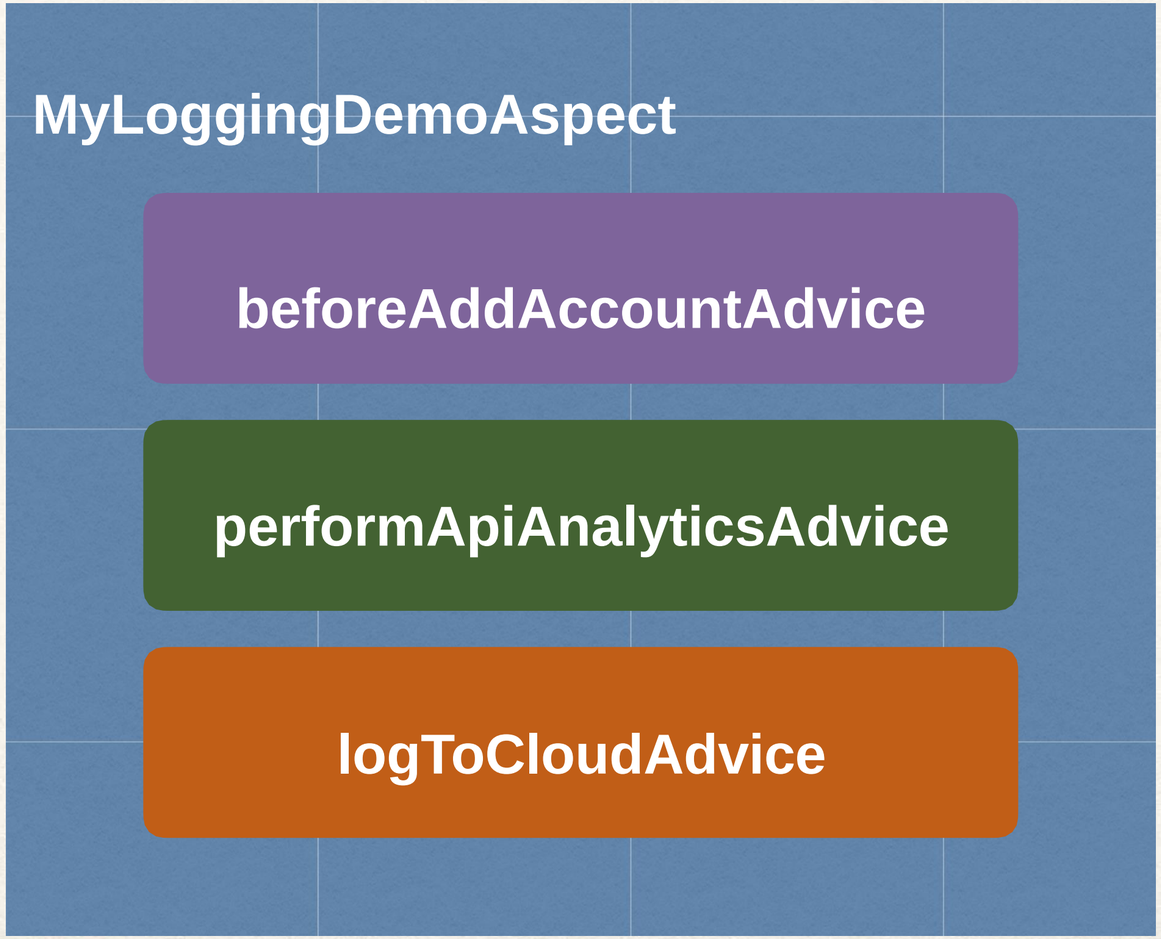
Problem

- 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

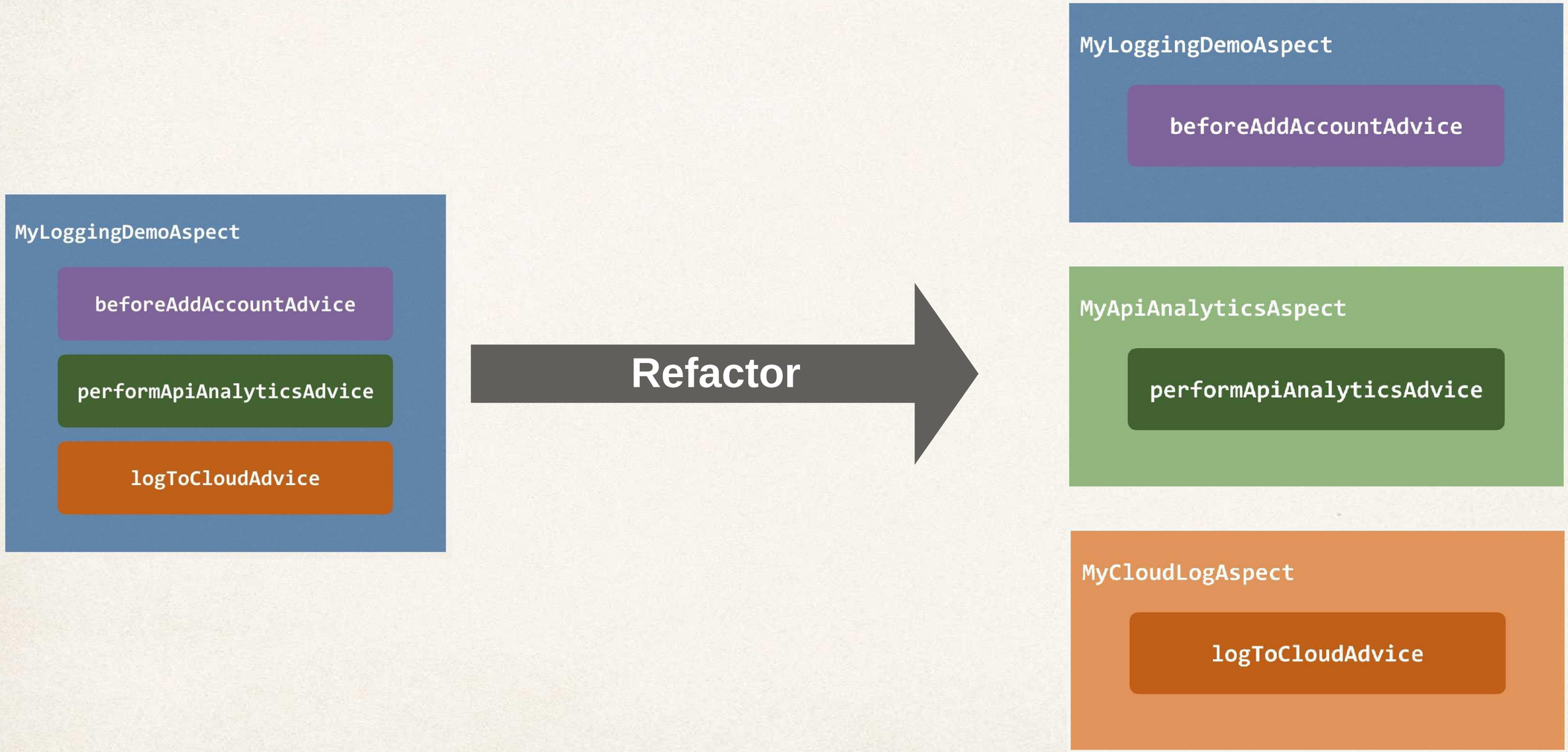
- Refactor: Place advices in separate Aspects
- Control order on Aspects using the @Order annotation
- Guarantees order of when Aspects are applied

Development Process

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

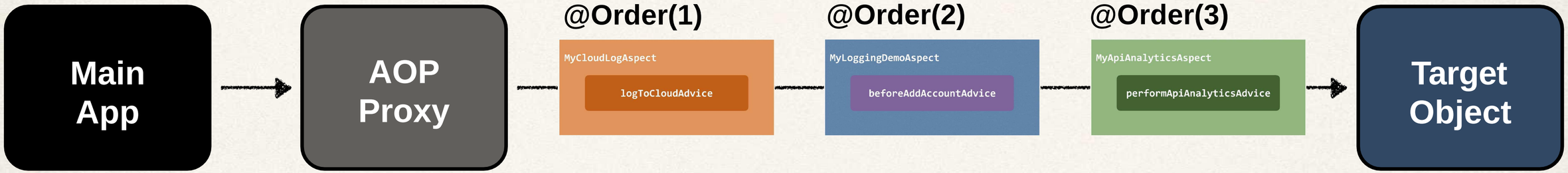
```
@Order(1)
public class MyCloudLogAspect {
  ...
}
```

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

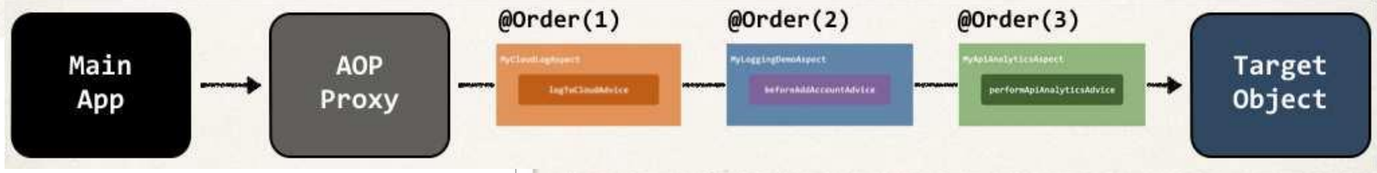
@Order

- We want the following order:

1. MyCloudLogAspect
2. MyLoggingDemoAspect
3. MyApiAnalyticsAspect



@Order annotation




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

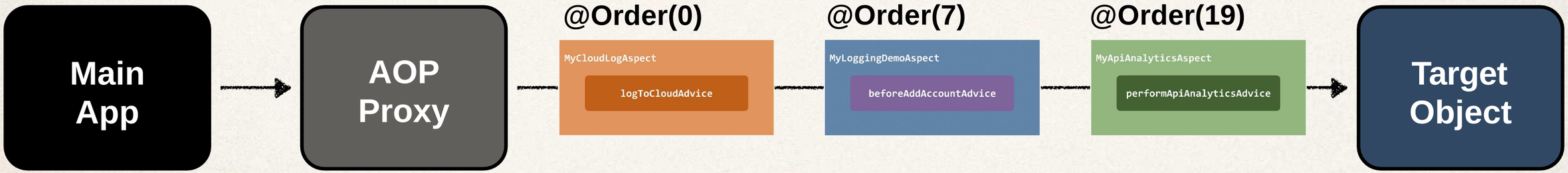
```
@Aspect
@Component
@Order(2)
public class MyLoggingDemoAspect {
 ...
}
```

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

luv2code

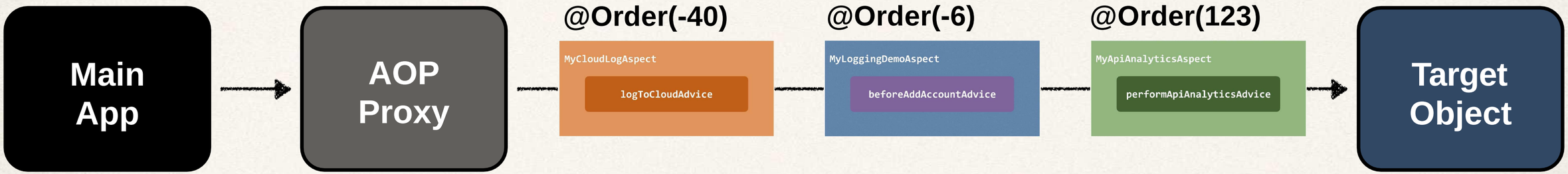
@Order annotation

- 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

luv2code

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

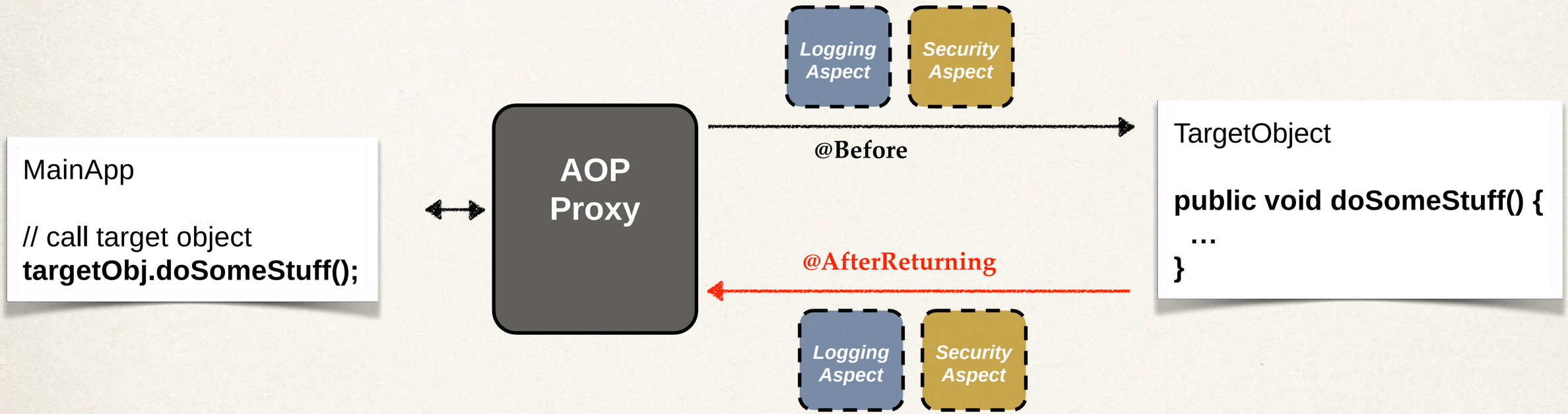
- @AfterReturning
- @AfterThrowing
- @After
- @Around

Aspect-Oriented Programming (AOP)
@AfterReturning Advice

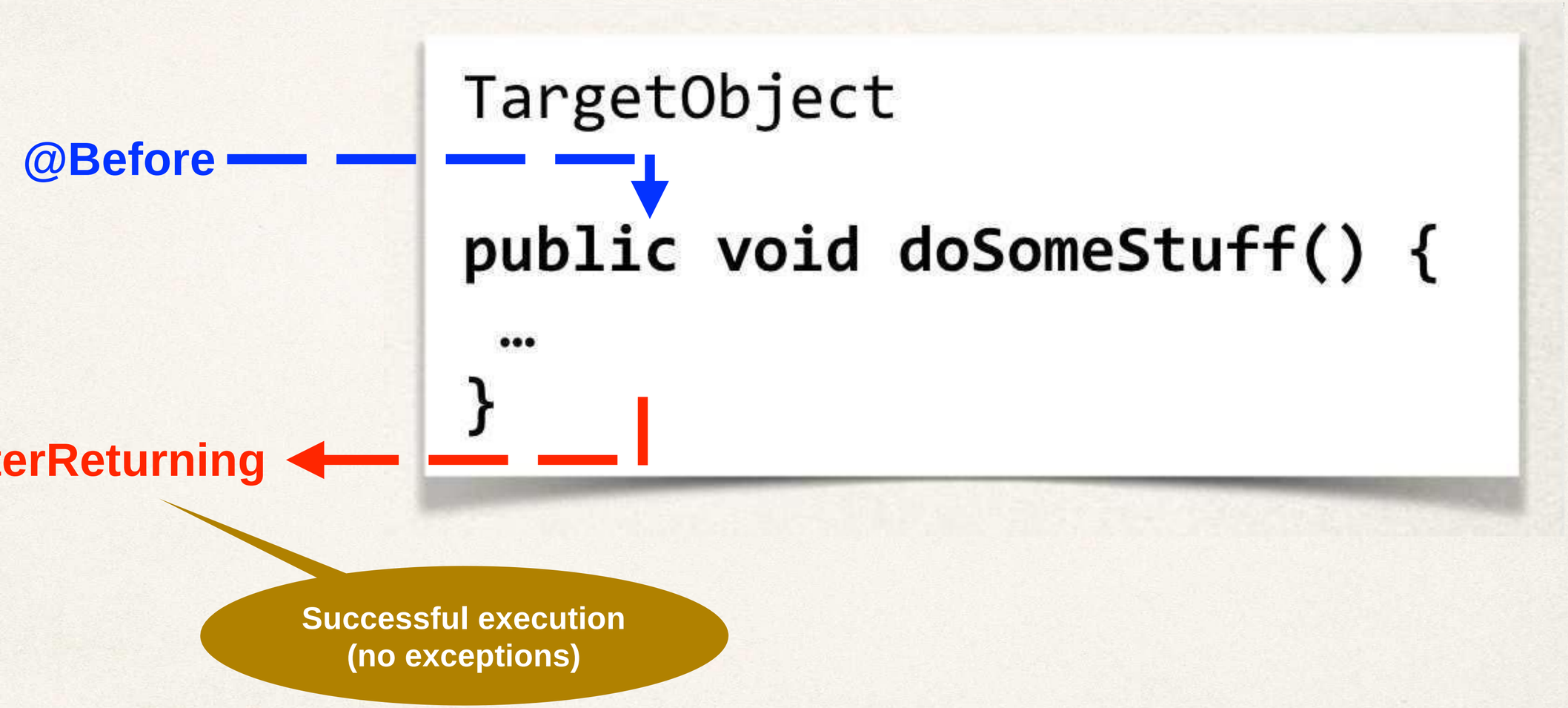
Advice Types

- **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



@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. .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. .aopdemo.dao.AccountDAO.findAccounts(..))",
    returning="result")
public void afterReturningFindAccountsAdvice() {









}
```

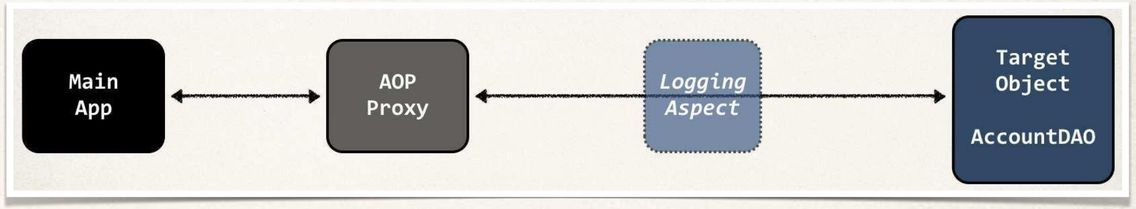
Access the Return Value

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




}
```

Access the Return Value

```
@AfterReturning(
    pointcut="execution(* com. .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

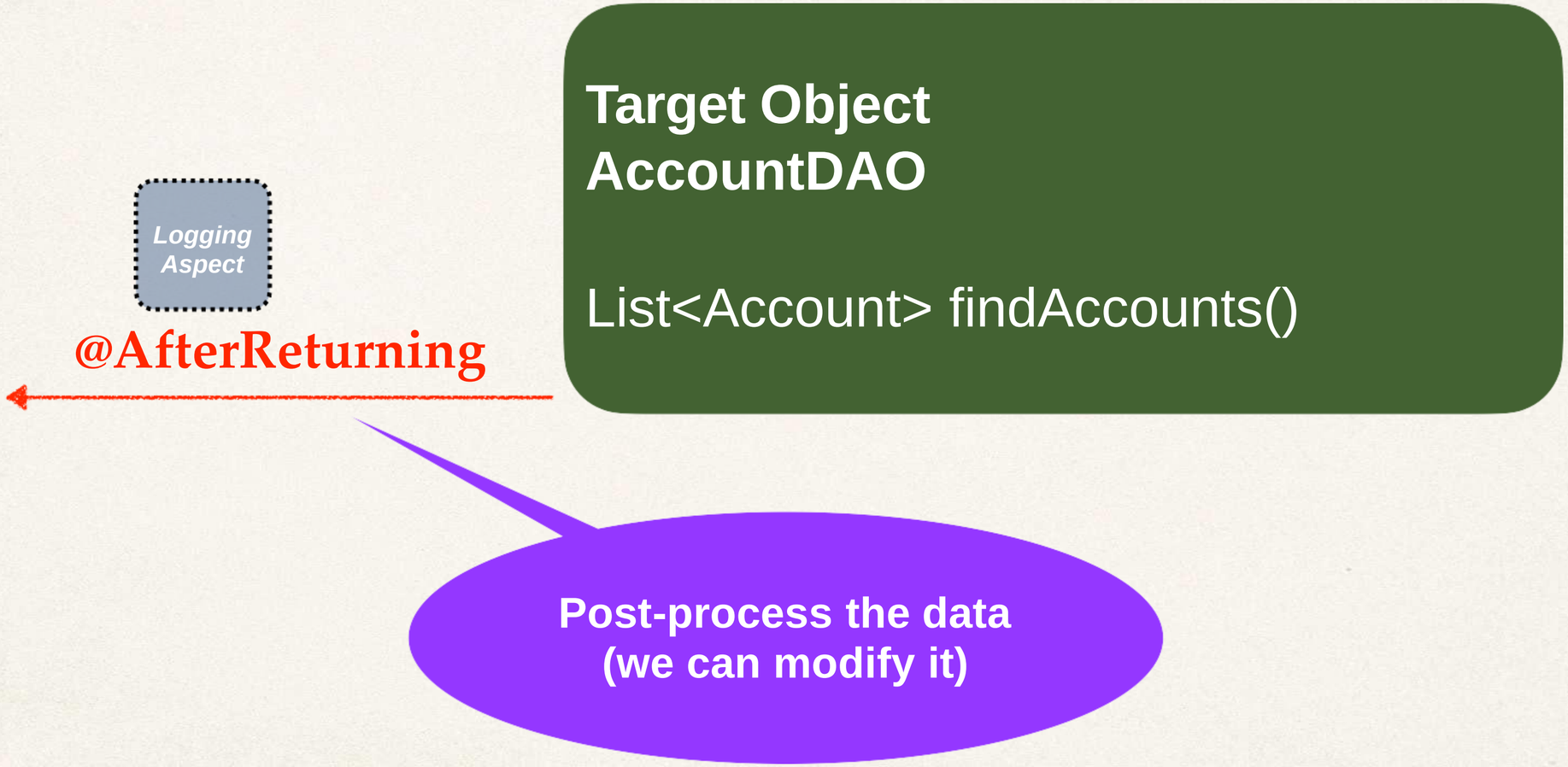
1. Prep Work: Add constructors to Account class
2. Add new method: findAccounts() in AccountDAO
3. Update main app to call the new method: findAccounts()
4. Add @AfterReturning advice

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



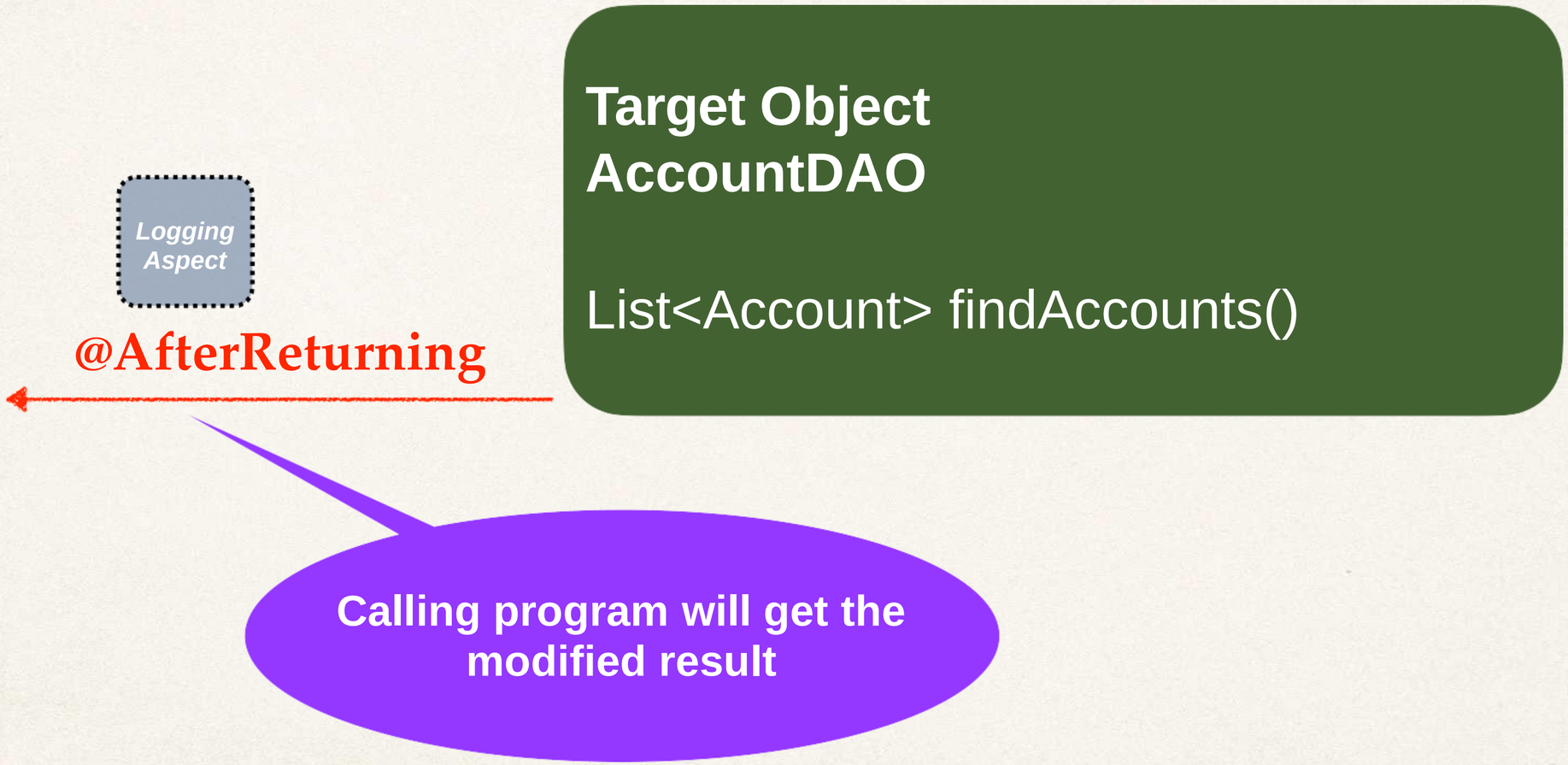
Modify the Return Value

```
@AfterReturning(
    pointcut="execution(* com. .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

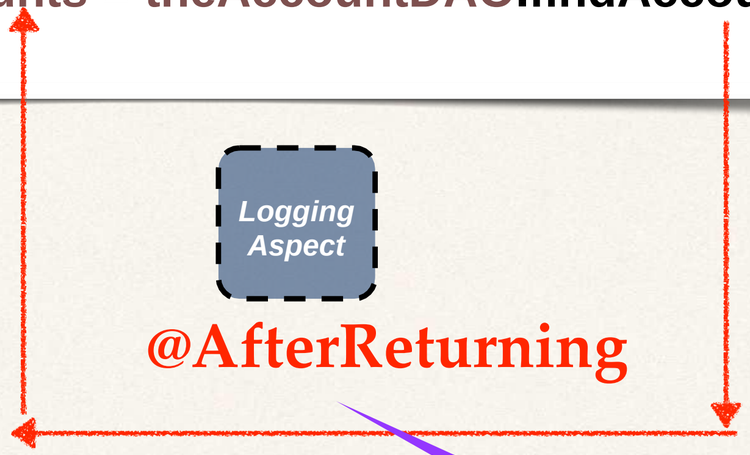
```
@AfterReturning(
    pointcut="execution(* com. .aopdemo.dao.AccountDAO.findAccounts(..))",
    returning="result")
public void afterReturningFindAccountsAdvice(
        JoinPoint theJoinPoint, List<Account> result) {

  // 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();
```



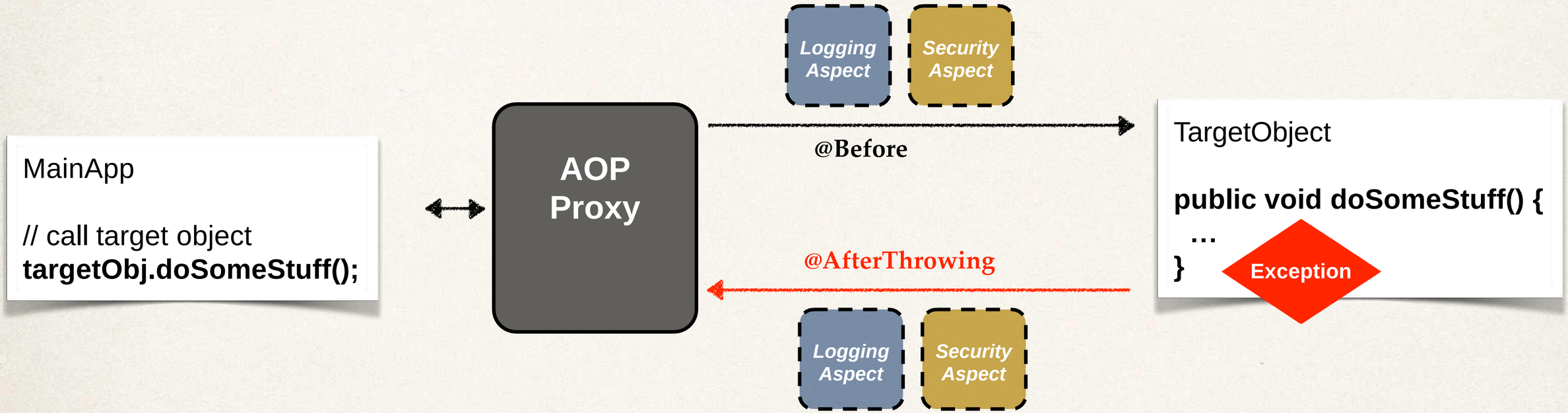
Calling program will get the modified result

Aspect-Oriented Programming (AOP)
@AfterThrowing Advice

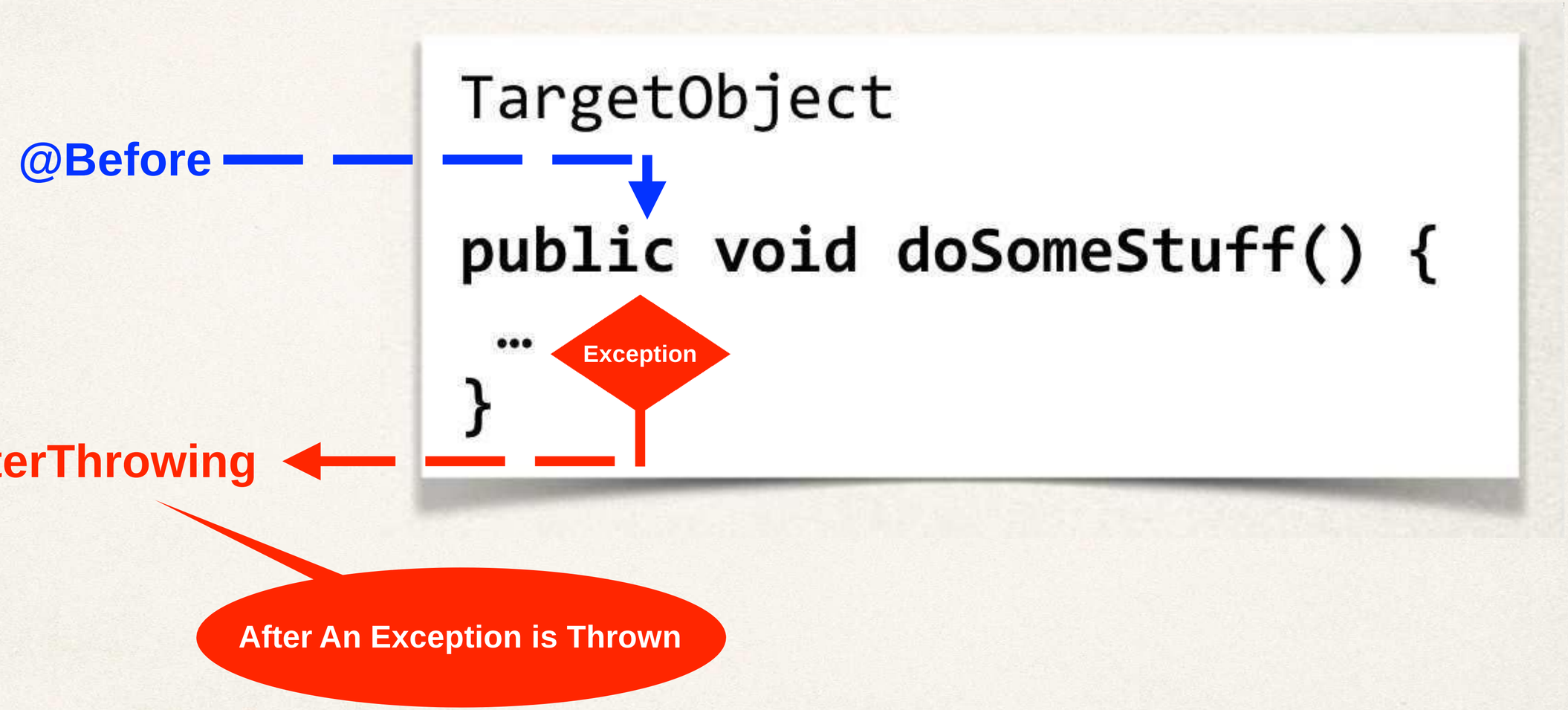
Advice Types

- **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

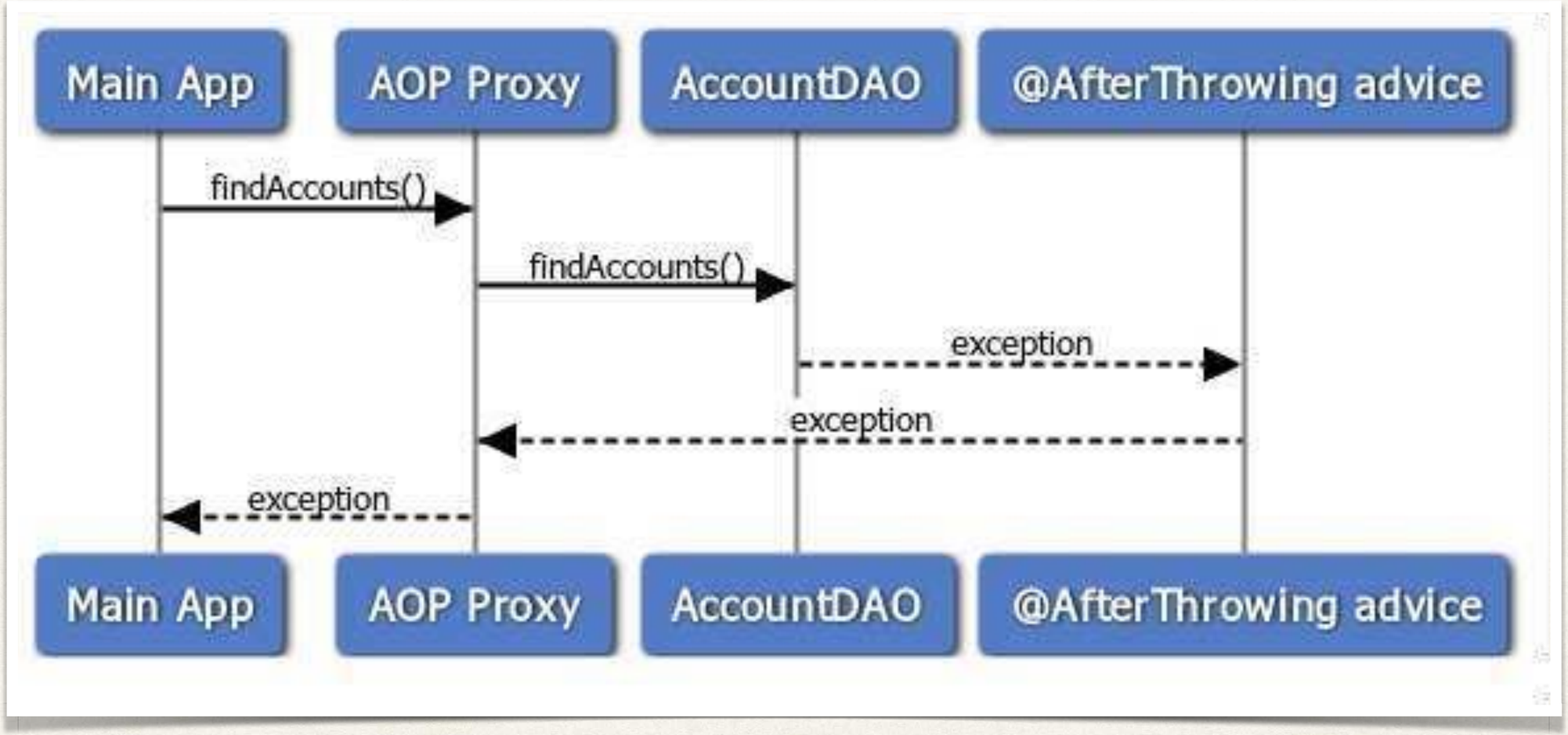
@AfterThrowing Advice - Interaction



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. .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. .aopdemo.dao.AccountDAO.findAccounts(..))",
    throwing="theExc")
public void afterThrowingFindAccountsAdvice() {









}
```

Access the Exception

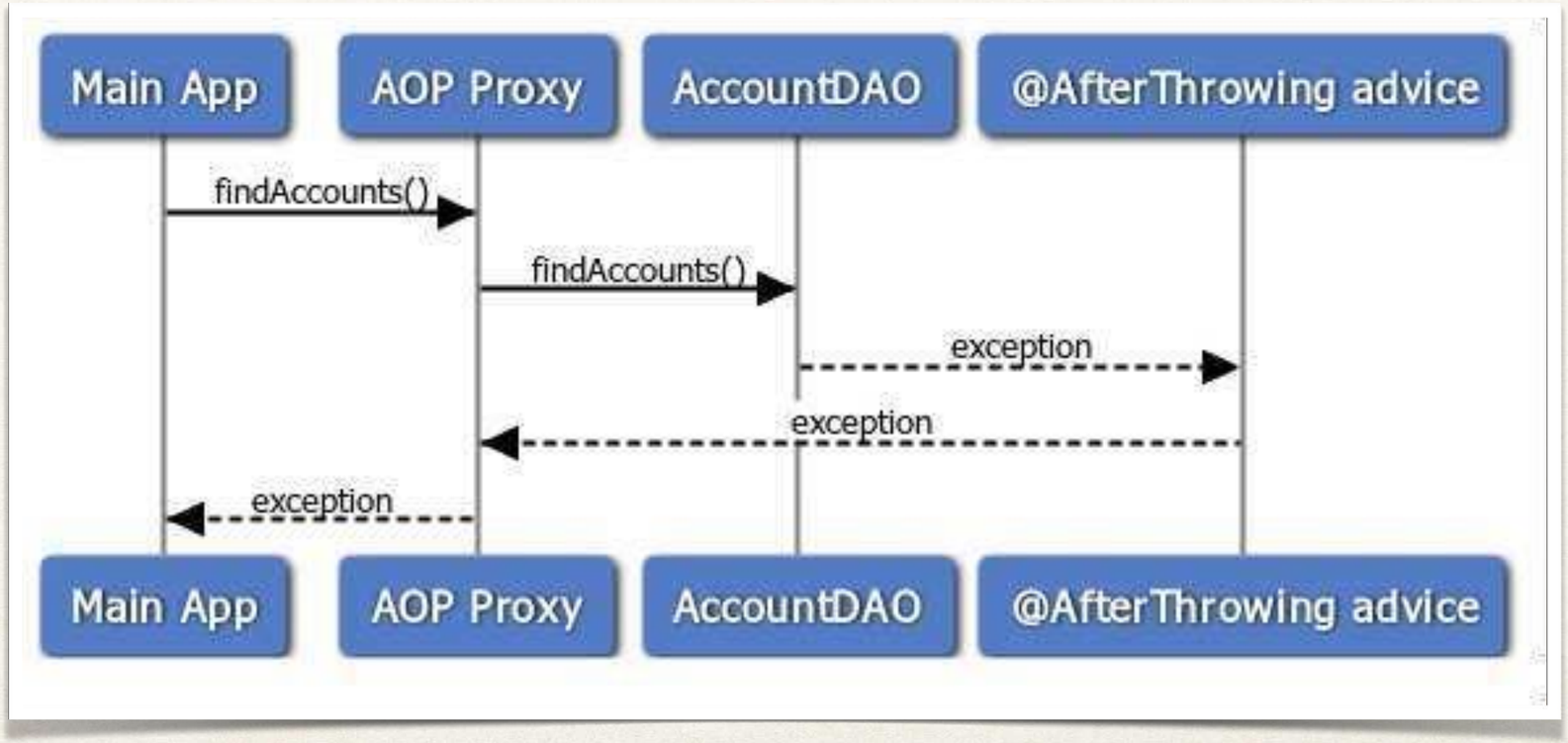
```
@AfterThrowing(
    pointcut="execution(* com. .aopdemo.dao.AccountDAO.findAccounts(..))",
    throwing="theExc")
public void afterThrowingFindAccountsAdvice(
        JoinPoint theJoinPoint, Throwable theExc) {




}
```

Access the Exception

```
@AfterThrowing(
    pointcut="execution(* com. .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

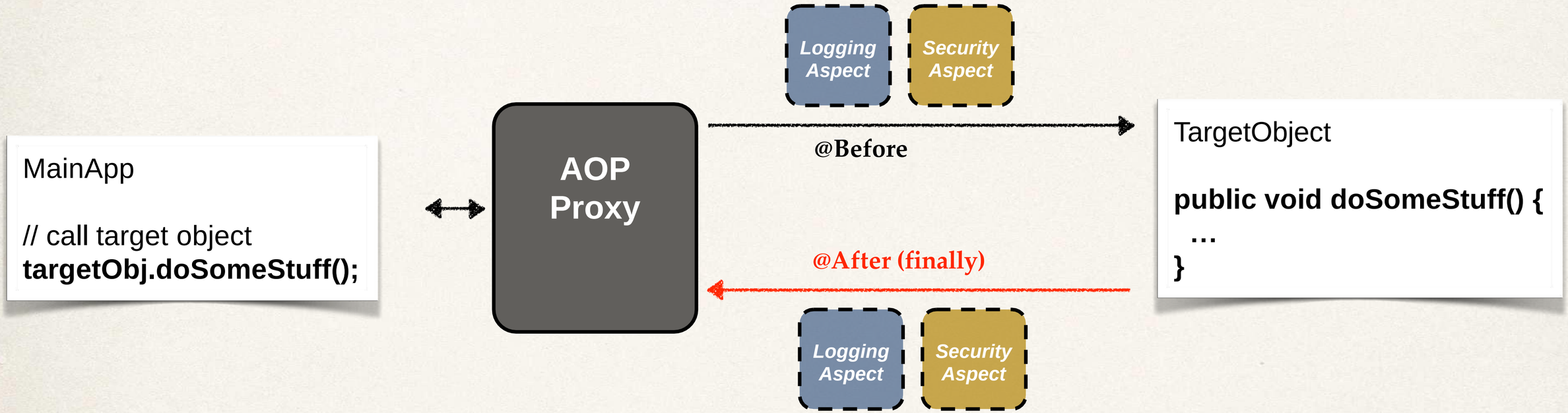
1. In Main App, add a try/catch block for exception handling
2. Modify AccountDAO to simulate throwing an exception
3. Add @AfterThrowing advice

Aspect-Oriented Programming (AOP)
@After Advice

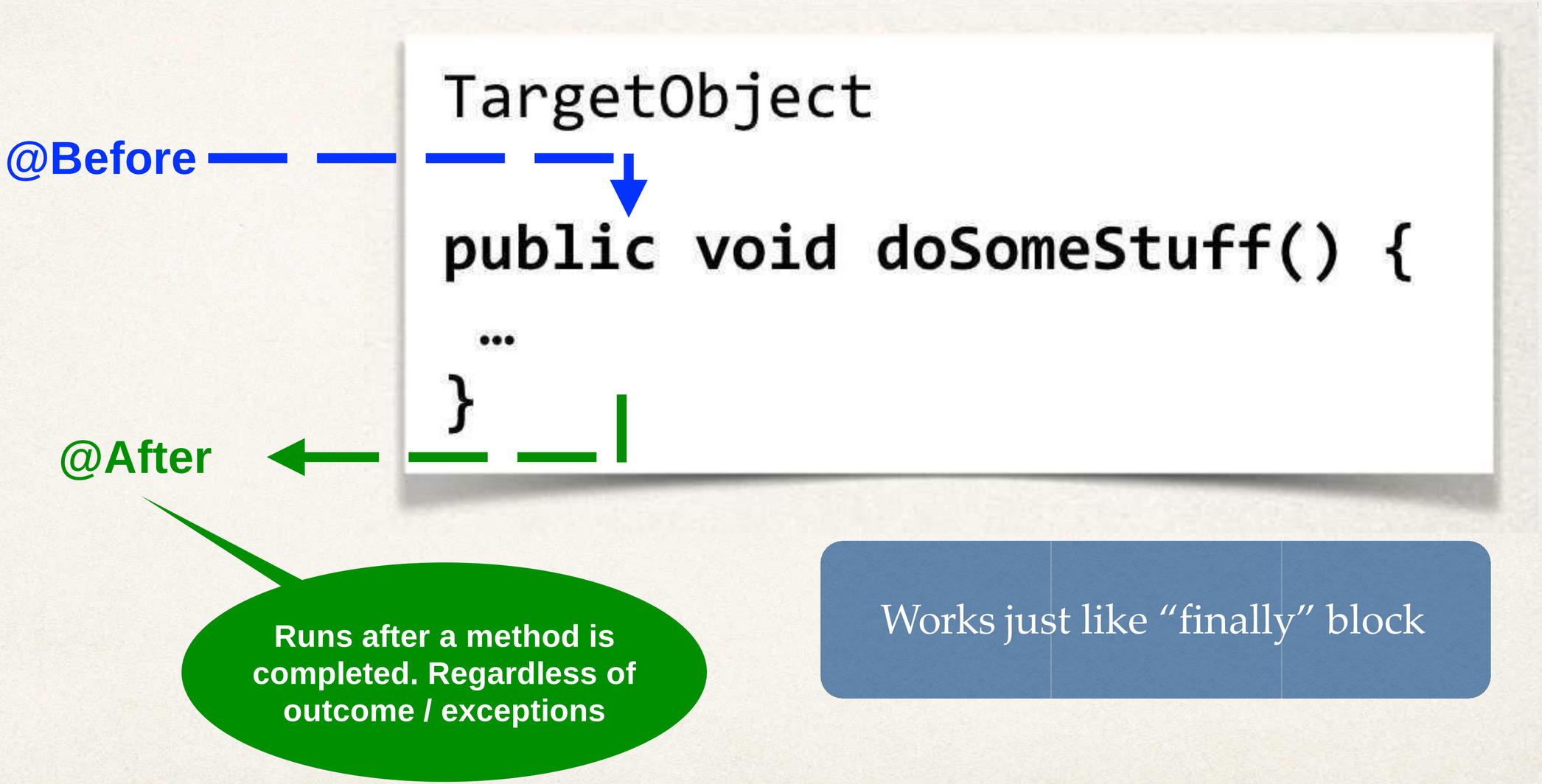
Advice Types

- **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

@After (finally) Advice - Interaction



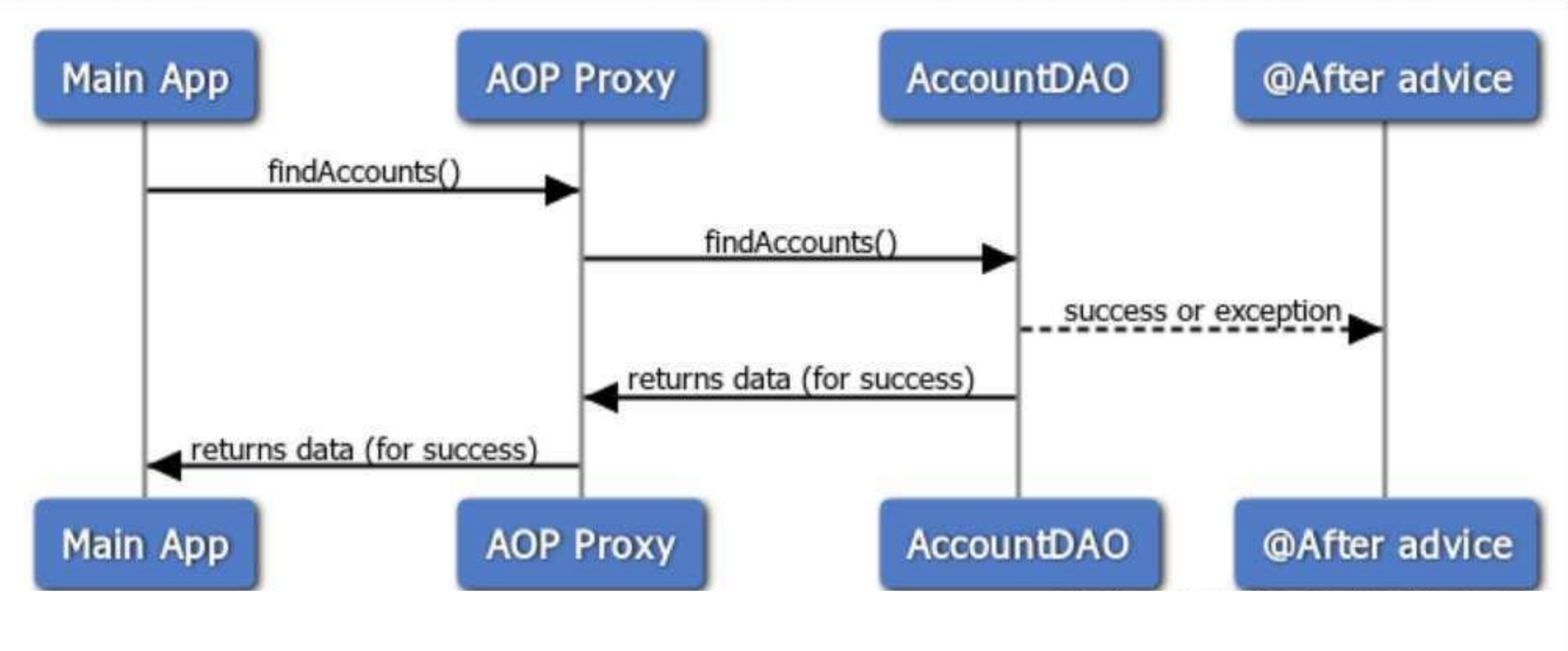
Advice - Interaction



Runs after a method is completed. Regardless of outcome / exceptions

Works just like “finally” block

Sequence Diagram



@After Advice - Use Cases

- Log the exception and/or perform auditing
- Code to run regardless of method outcome
- Encapsulate this functionality in AOP aspect for easy reuse

Example

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

@After Advice

- This advice will run after the method (finally … success/failure)

```
@After("execution(* com. .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 - @After

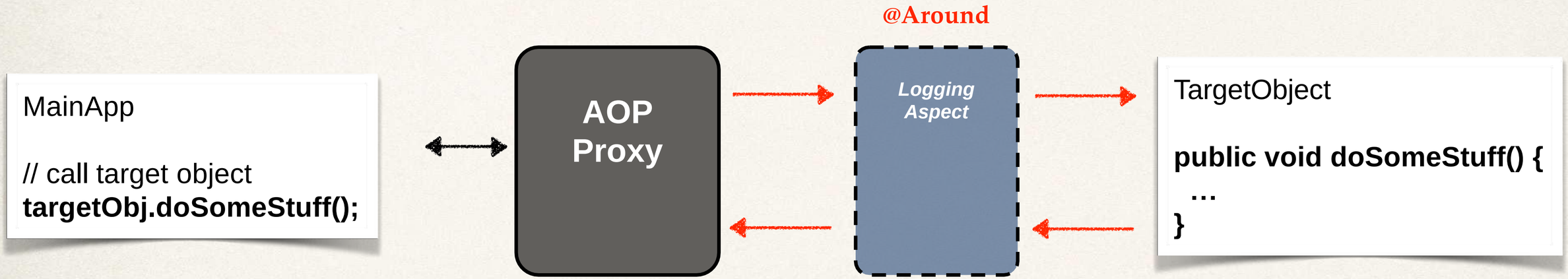
1. Prep work
2. Add @After advice
3. Test for failure/exception case
4. Test for success case

Aspect-Oriented Programming (AOP)
@Around Advice

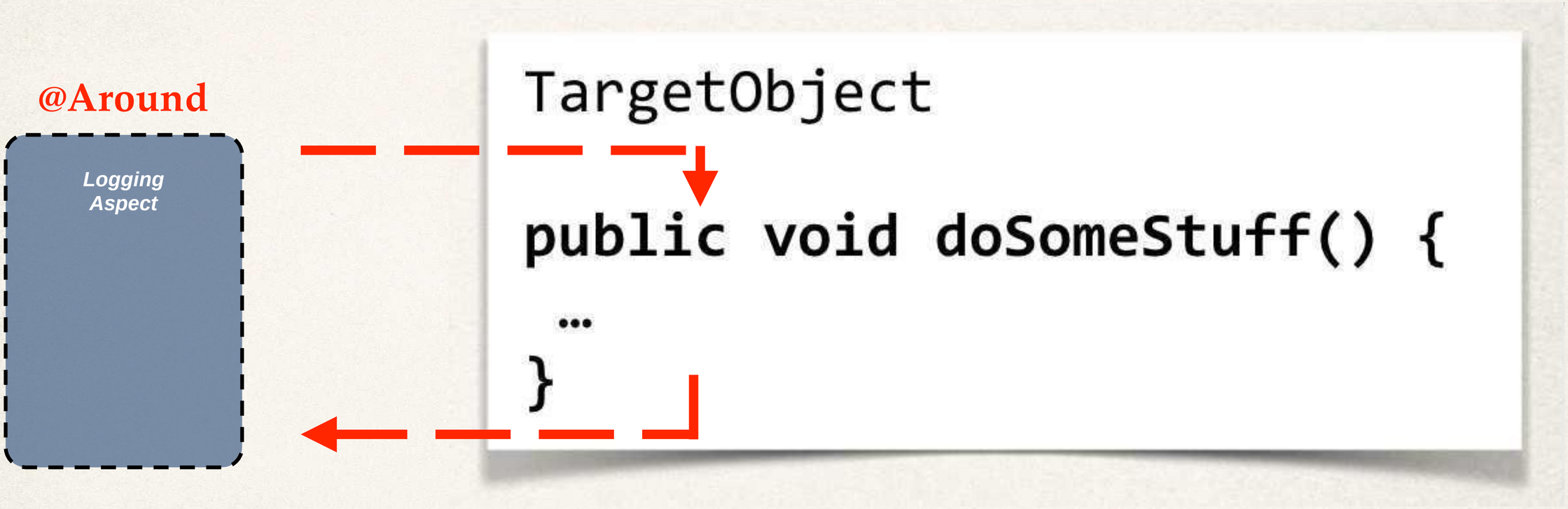
Advice Types

- **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

@Around Advice - Interaction



Advice - Interaction



@Around: Like a combination of @Before and @After

But gives you more fine-grained control

@Around Advice - Use Cases

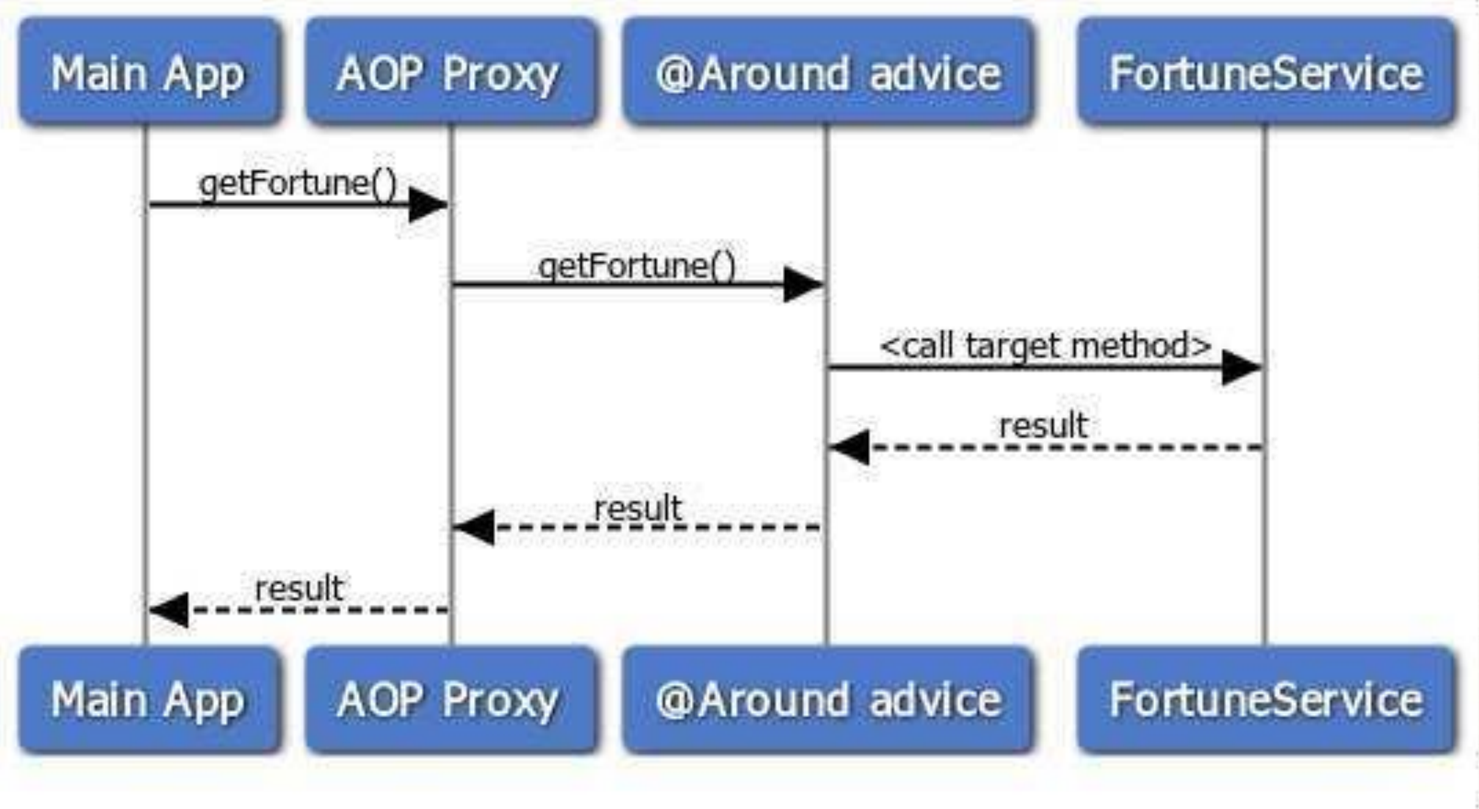
- Most common: logging, auditing, security
- Pre-processing and post-processing data
- Instrumentation / profiling code
  - How long does it take for a section of code to run?
- Managing exceptions
  - Swallow / handle / stop exceptions

Our FortuneService Example - Revisited

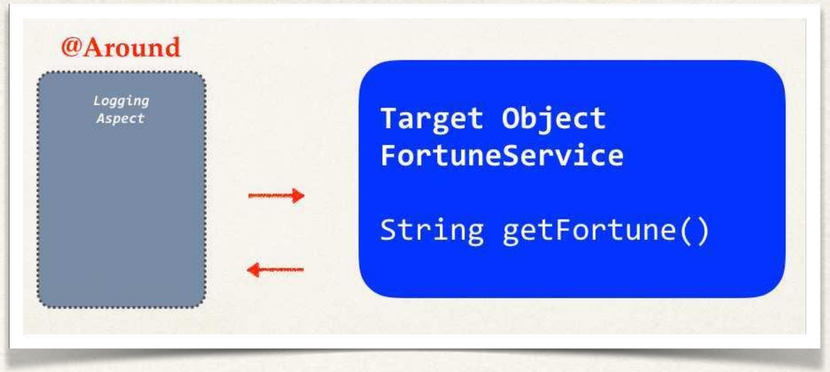
Target Object
FortuneService

String getFortune()

Sequence Diagram



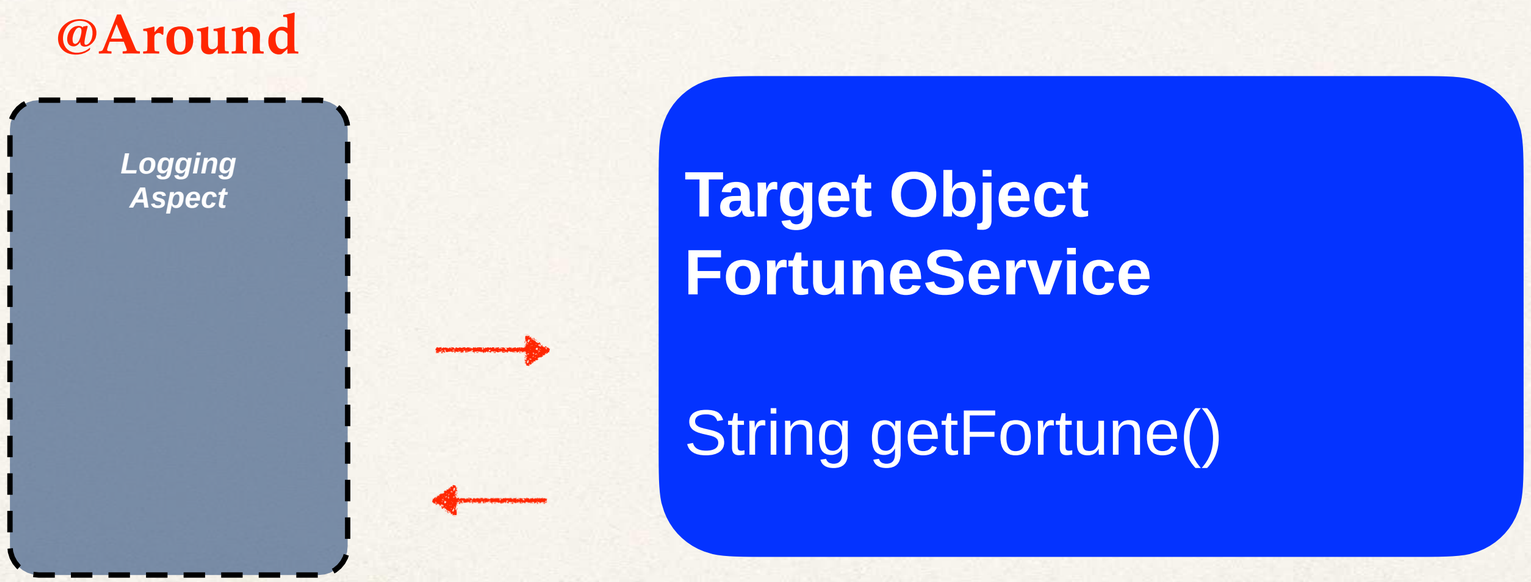
ProceedingJoinPoint



- 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**

Example

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



@Around Advice

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









}
```

@Around Advice

```
@Around("execution(* com. .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;
}
```

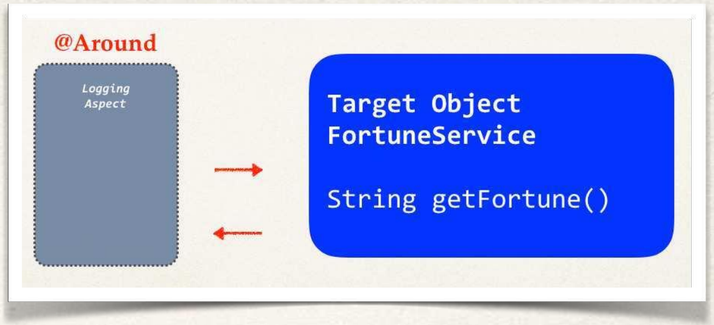
Development Process - @Around

Step-By-Step

1. Create TrafficFortuneService
2. Update main app to call TrafficFortuneService
3. Add @Around advice

Aspect-Oriented Programming (AOP)
@Around Advice - Handle Exception

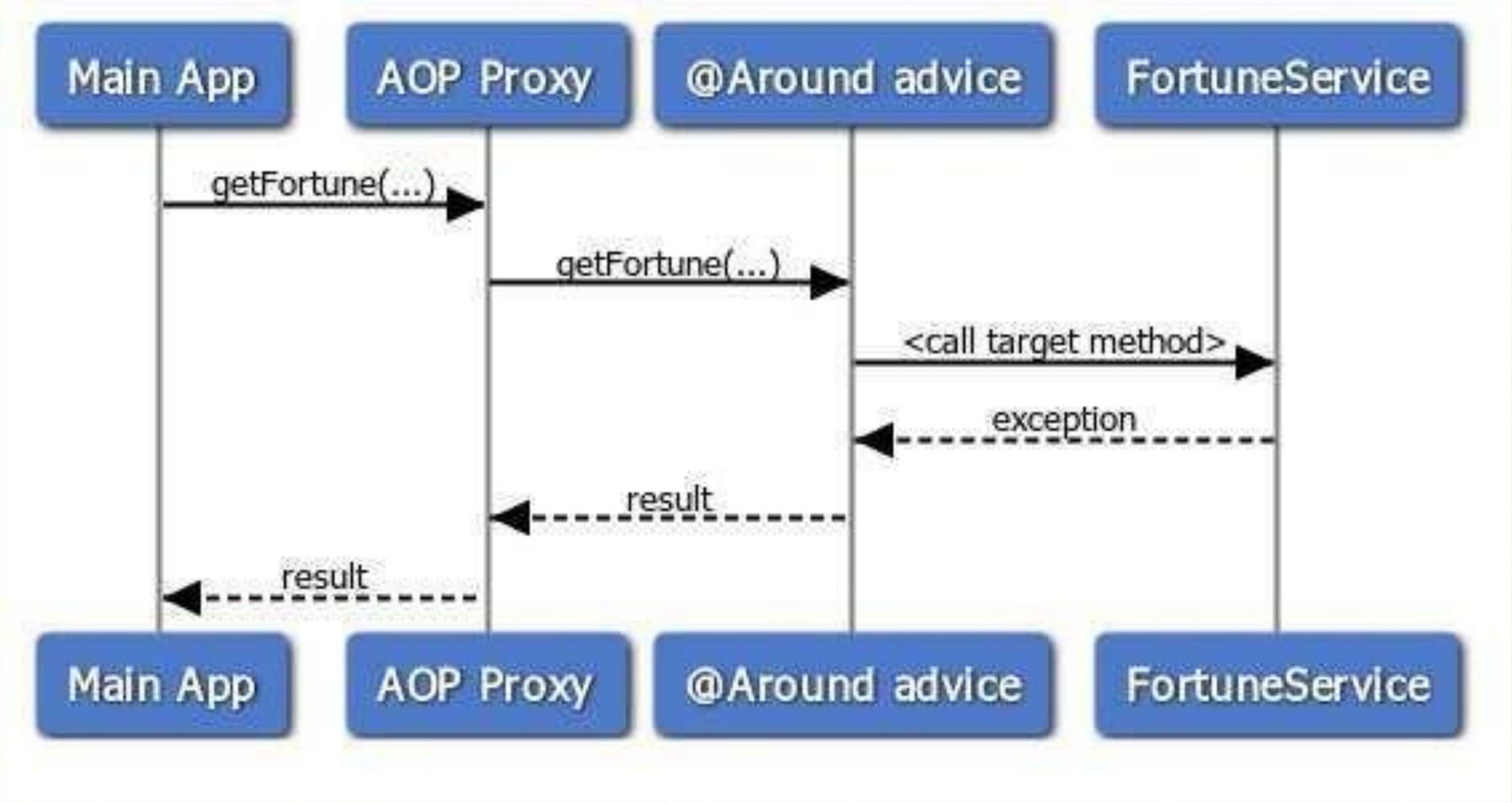
ProceedingJoinPoint - Revisited



- 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



Handle Exception

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









}
```

Handle Exception

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

    Object result = null;









}
```

Handle Exception

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

    Object result = null;

    try {
        // let's execute the method
        result = theProceedingJoinPoint.proceed();

    }









}
```

Handle Exception

```
@Around("execution(* com. .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!";
    }


}
```

Handle Exception

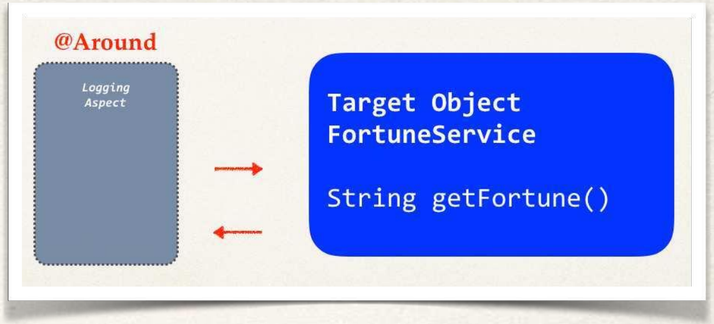
```
@Around("execution(* com. .aopdemo.service.*.getFortune(..))") public
Object afterGetFortune(
        ProceedingJoinPoint theProceedingJoinPoint) throws Throwable  {

    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;
}
```

Development Process - @Around

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

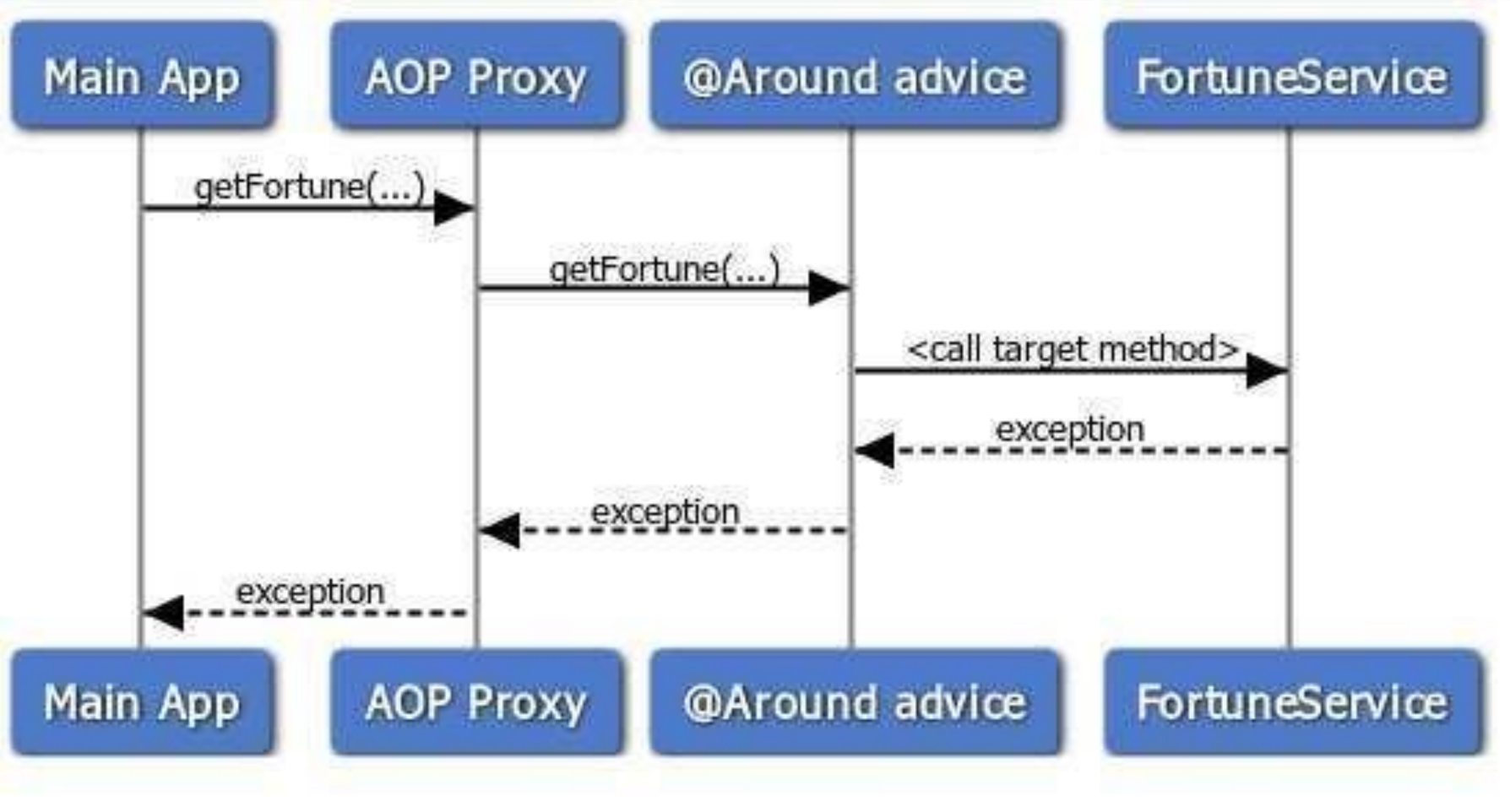
Aspect-Oriented Programming (AOP)
@Around Advice - Rethrow Exception

ProceedingJoinPoint - Revisited



- For an exception thrown from proceeding join point
  - You can handle / swallow /stop the exception
  - **Or you can simply rethrow the exception**

Sequence Diagram



Rethrow Exception

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









}
```

Rethrow Exception

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

    try {
        // let's execute the method
        Object result = theProceedingJoinPoint.proceed();

        return result;
    }









}
```

Rethrow Exception

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

    try {
        // let's execute the method
        Object result = theProceedingJoinPoint.proceed();

        return result;
    }
    catch (Exception exc) {
        // log exception
        System.out.println("@Around advice: We have a problem " + exc);

        // rethrow it
        throw exc;
    }

}
```

FAQ: JoinPoint vs ProceedingJoinPoint

- **When to use JoinPoint vs ProceedingJoinPoint?**

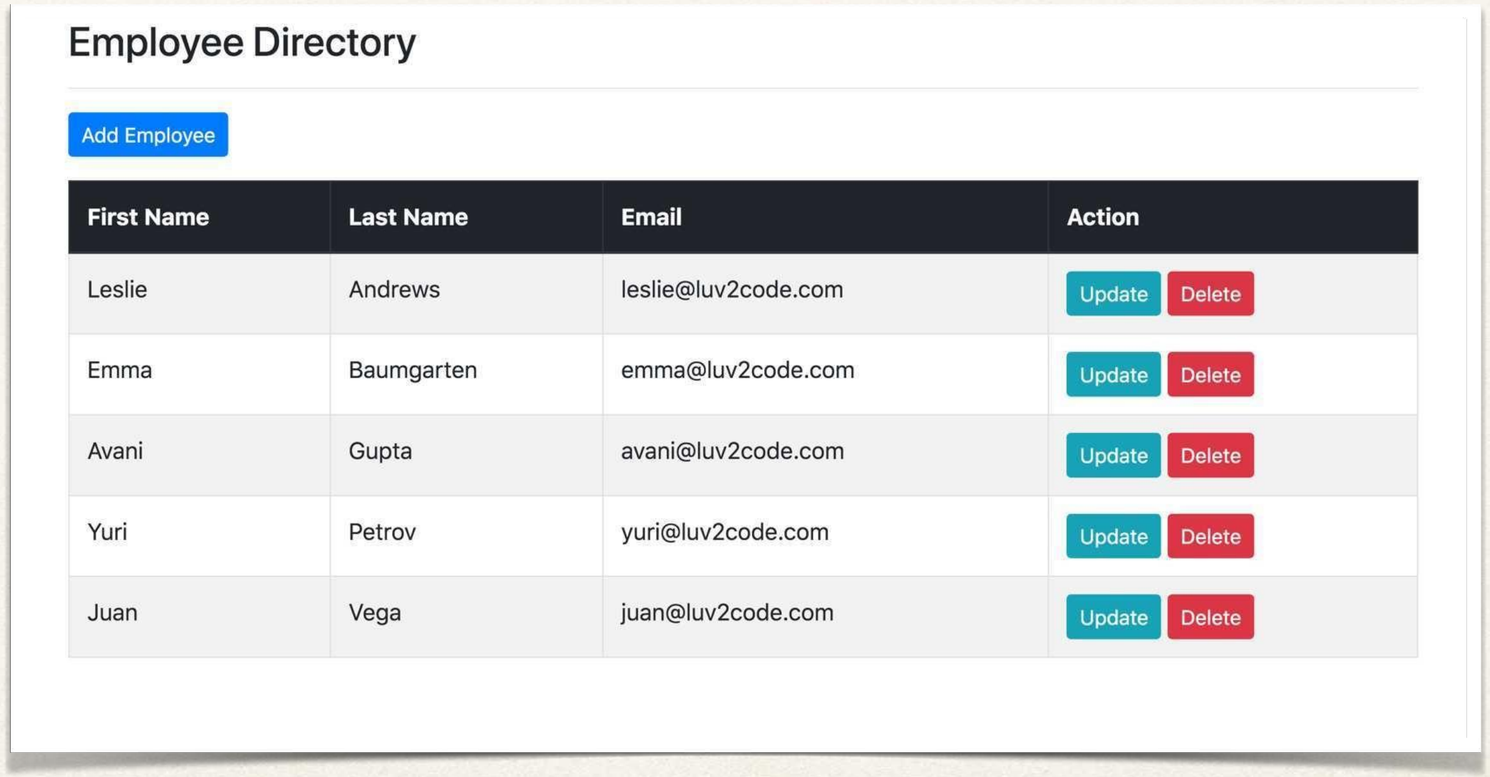
- Use **JoinPoint** with following advice types
  - @Before, @After, @AfterReturning, @AfterThrowing

- Use **ProceedingJoinPoint** with following advice type
  - @Around

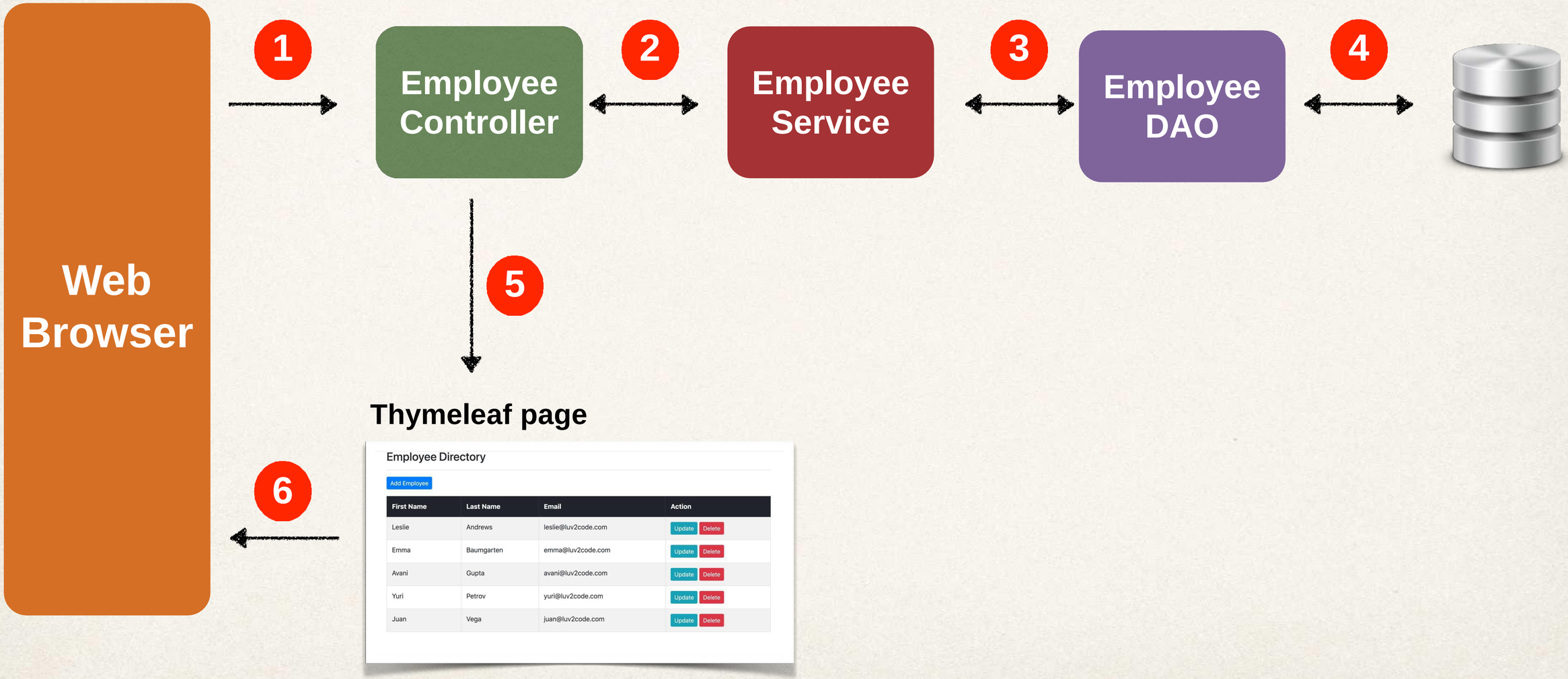
Aspect-Oriented Programming (AOP)
AOP and Spring MVC

Goal

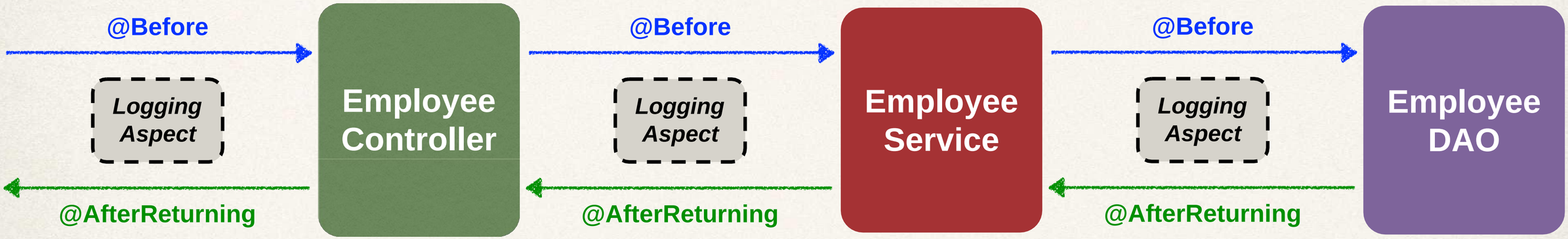
- Add AOP Logging support to our Spring MVC CRUD app



Big Picture


Web Browser
1
Employee Controller
2
Employee Service
3
Employee DAO
4
5
Thymeleaf page
6
Employee Directory
Add Employee
First Name
Last Name
Email
Action
Leslie
Andrews
leslie@luv2code.com
Update
Delete
Emma
Baumgarten
emma@luv2code.com
Update
Delete
Avani
Gupta
avani@luv2code.com
Update
Delete
Yuri
Petrov
yuri@luv2code.com
Update
Delete
Juan
Vega
juan@luv2code.com
Update
Delete

Logging Aspect



Development Process

Step-By-Step

1. Add Spring Boot AOP Starter to Maven pom file
2. Create Aspect
    1. Add logging support
    2. Setup pointcut declarations
    3. Add @Before advice
    4. Add @AfterReturning advice

Step 1: 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>
```

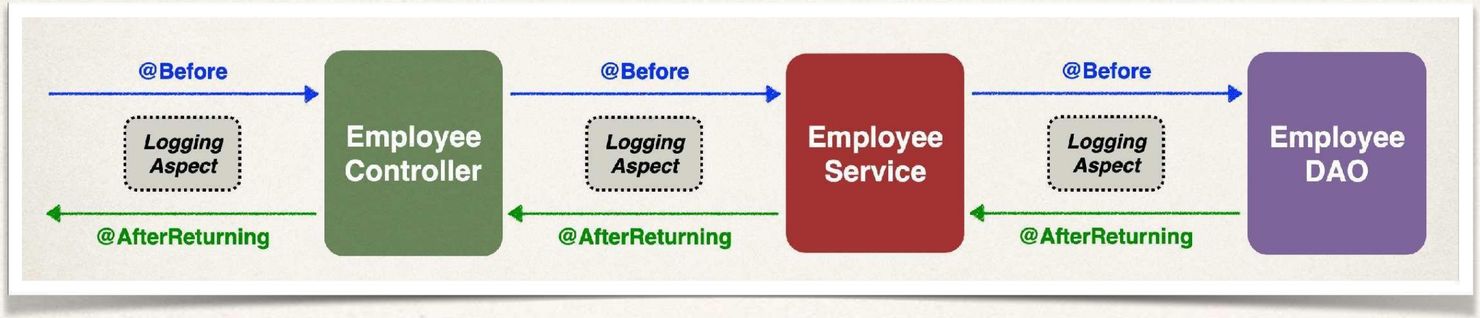
- Since this dependency is part of our pom.xml
  - Spring Boot will **automatically** enable support for AOP

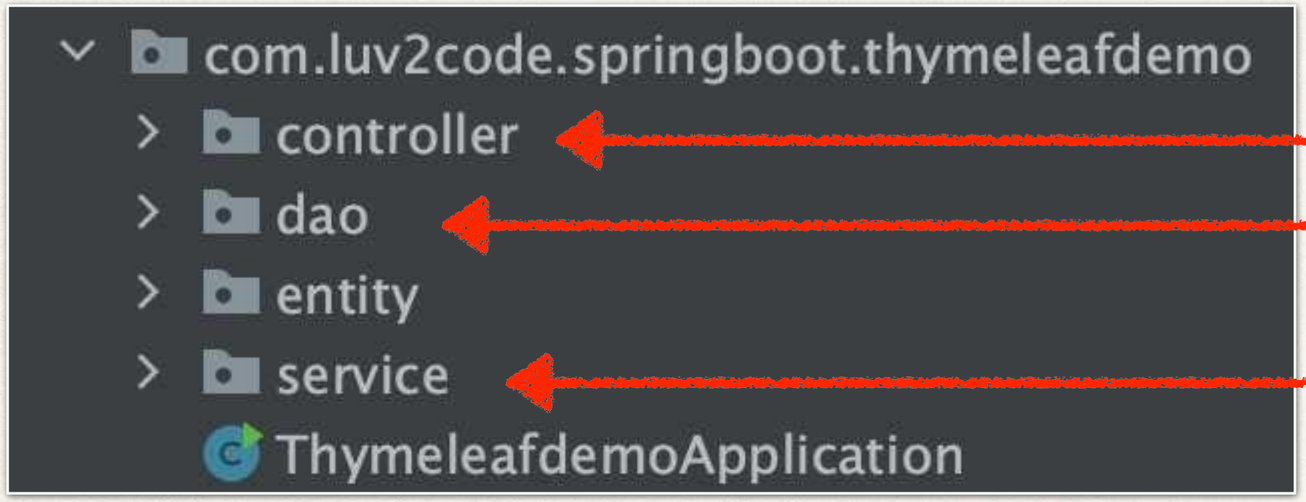
Development Process

Step-By-Step

1. Add Spring Boot AOP Starter to Maven pom file
2. Create Aspect
    1. Add logging support
    2. Setup pointcut declarations
    3. Add @Before advice
    4. Add @AfterReturning advice

Pointcut Declarations





Only match on these three packages