5. Aspect Oriented Programming

1. What Is Cross-Cutting Concern & Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

What is Cross-Cutting Concern & Aspect Oriented Programming?

Cross-Cutting Concern

- *Generic functionality that is needed in many places in your application.*
- Logging, tracing, caching, error handling etc.

Aspect Oriented Programming

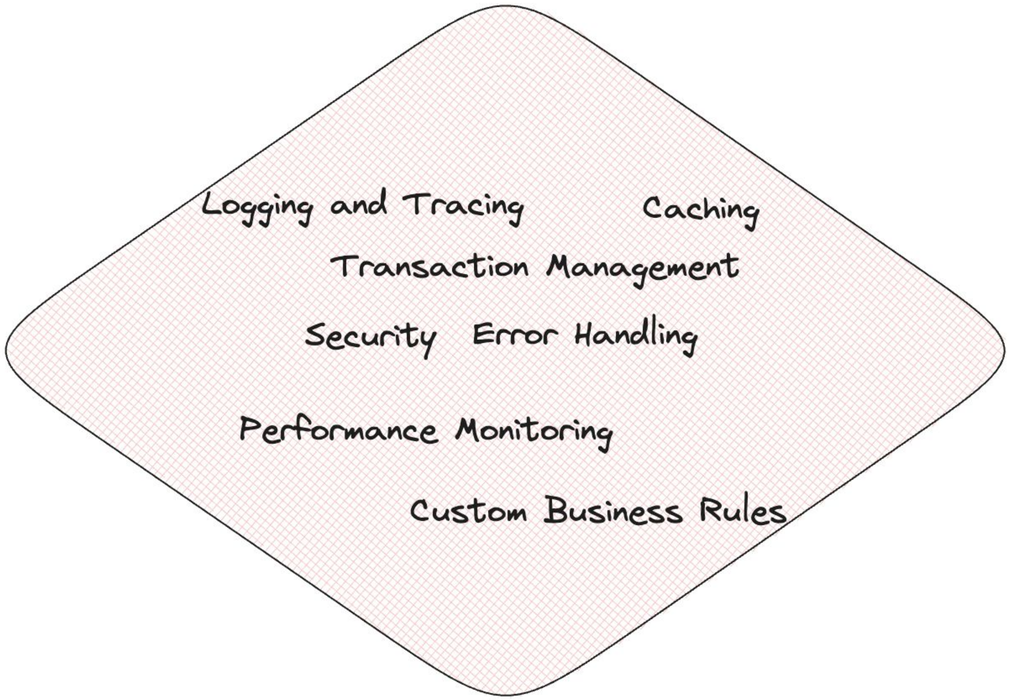
- *A programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.*
- enables modularization of cross-cutting concerns.
- makes code clean and simpler.
- is shortly AOP

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

What Problem Does AOP Solve?

- AOP separates core business functionality.
- Allows us to be able to divide a bigger problem into smaller pieces
- Provide separation of concerns principle.
- Enables modularization of *cross-cutting concerns*



An Example of Cross Cutting Concern

Perform a role-based security check before every application method

A sign this requirement is a cross-cutting-concern

2 PROBLEMS:

Code Complexity

Code Duplication

First Problem: *Code Complexity*

```
public class RewardNetworkImpl implements RewardNetwork {
   public RewardConfirmation rewardAccountFor(Dining dining) {

      // Security-related code
      if (!hasPermission(SecurityContext.getPrincipal()) {
         throw new AccessDeniedException();
      }

      // Application code
      Account a = accountRepository.findByCreditCard(…
      Restaurant r = restaurantRepository.findByMerchantNumber(…
      MonetaryAmount amt = r.calculateBenefitFor(account, dining);
      …
   }
}
```

Mixing of concerns

- Security and Business logic together
- Hard to test

Second Problem: *Code Duplication*

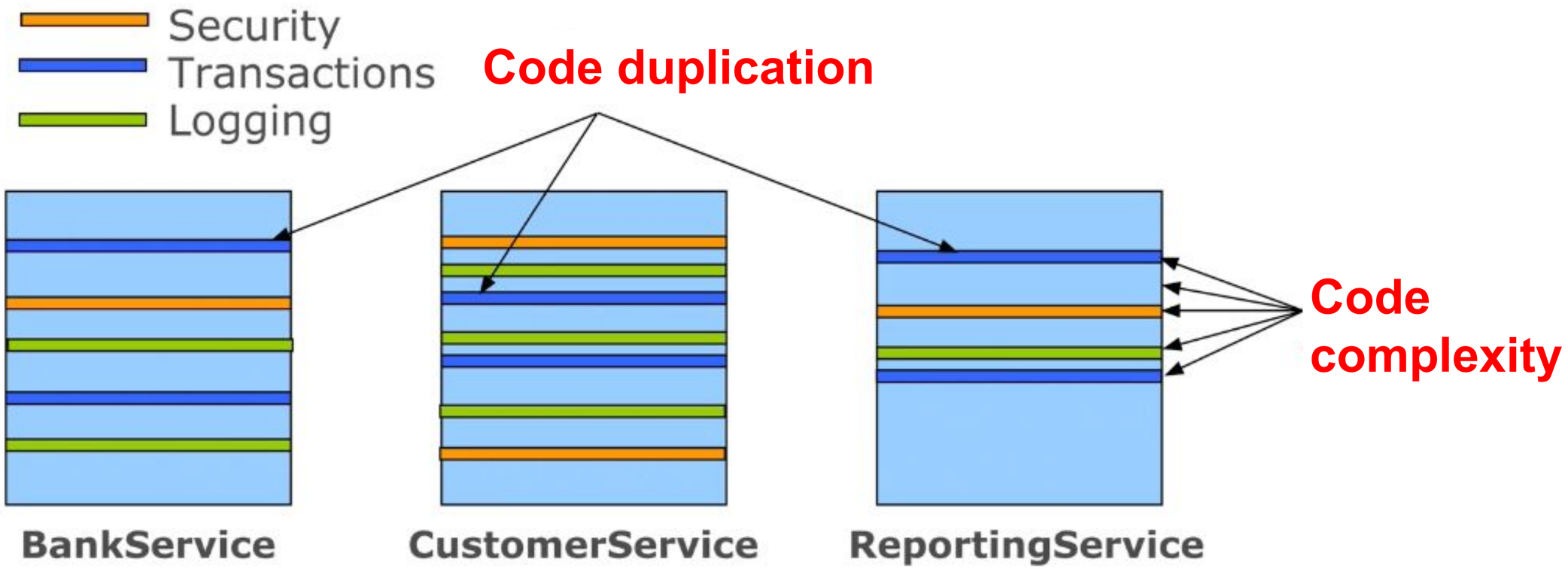
```
public class JpaAccountManager implements AccountManager {
    public Account getAccountForEditing(Long id) {
        if (!hasPermission(SecurityContext.getPrincipal()) {
            throw new AccessDeniedException();
        }
        ...
```

Duplication

```
public class JpaMerchantReportingService
        implements MerchantReportingService {
    public List<DiningSummary> findDinings(String merchantNumber,
                                           DateInterval interval) {
        if (!hasPermission(SecurityContext.getPrincipal()) {
            throw new AccessDeniedException();
        }
        ...
```

- Copy and paste the same code
- Must change it every where

System Design Without Modularization



System Design Solution

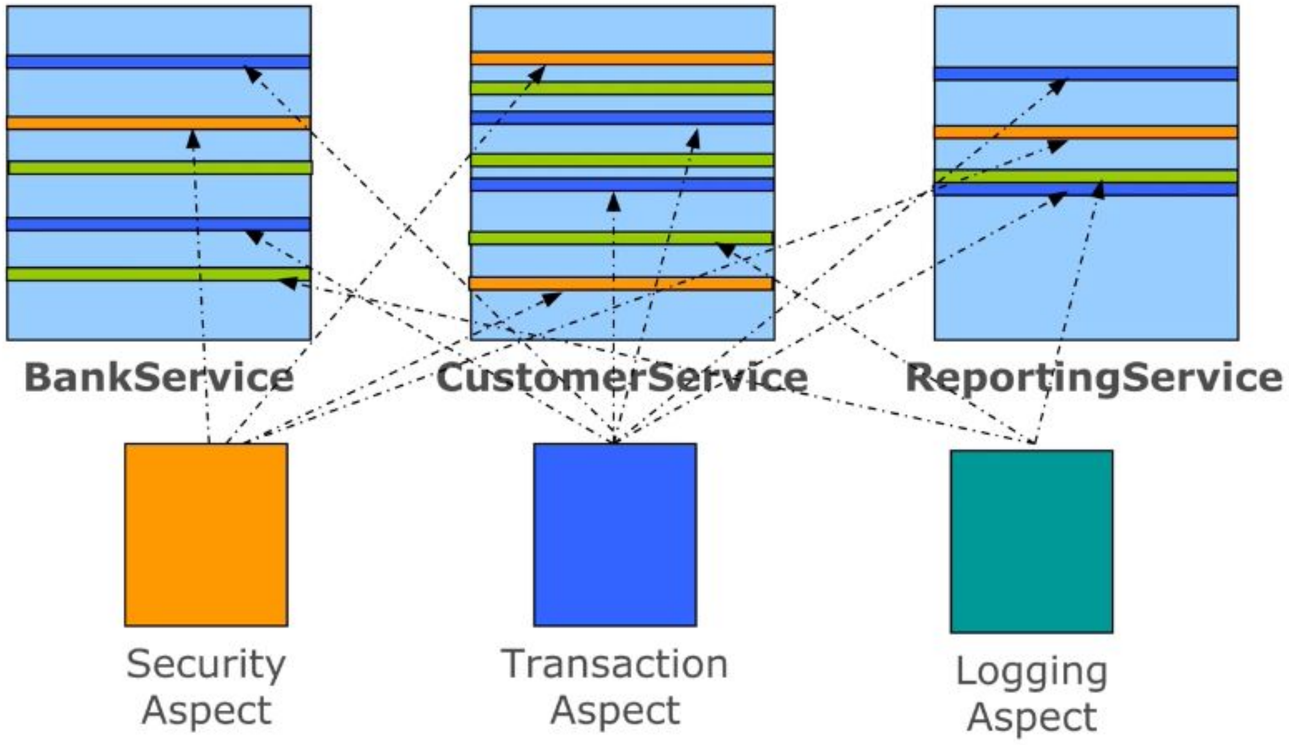
Implement your main application logic

Write Aspects to implement cross cutting concerns

Use aspects in your application

System Design With AOP

🔔 Now, main logic and concerns are *independent*

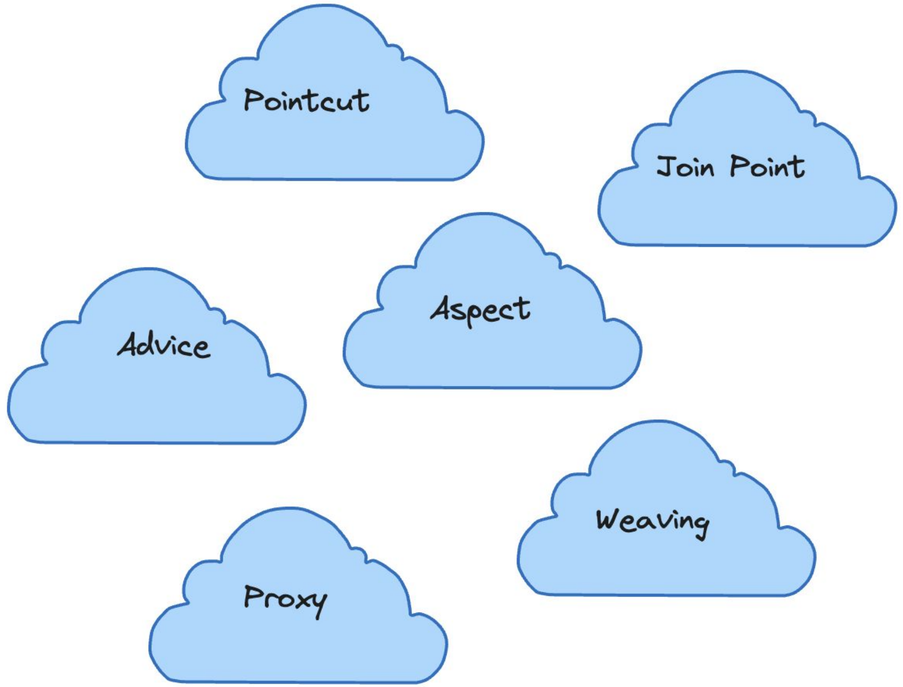


5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

AOP Concepts

- **JoinPoint:** A point in the execution of a program such as a method call or exception thrown.
- **Pointcut**: An expression that selects one or more JoinPoint.
- **Advice**: A specific code executed at a certain join point.
- **Aspect**: A module that encapsulates Pointcuts + Advices
- **Weaving**: Technique by which aspects are combined with main code
- **Proxy:** An “enhanced” class that stands in place of your original, with extra behavior



5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

Pointcut Expressions

🔔 *Recall*: An expression that selects one or more JoinPoint. (*where to apply Advice*)

execution(*<method pattern>*)

- Method must match the pattern.

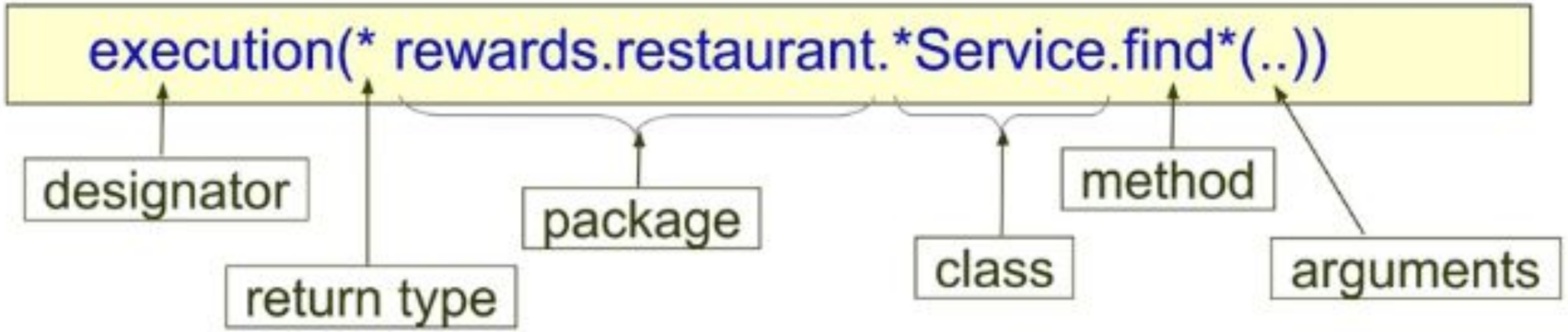
Method Pattern

- [Modifiers] ReturnType* [Class Type] MethodName(Arguments)* [throw ExceptionType]
- ReturnType and MethodName(Arguments) are mandatory.

Can chain together to create composite pointcuts

- and: **&&**
- or: **||**
- not: **!**
- execution(*<method pattern 1>*) || execution(*<method pattern 2>*)

Example Expression



Any method
--> starts with find with zero or more arguments
--> ends with Service
--> in a package with rewards.restaurant
--> any return type

- ✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]
- ✔ * : matches once (ReturnType, Package, class, MethodName, Argument)
- ✔ .. : matches zero or more (Argument, Package)

Example Expression: *Any Class or Package*

```
execution(void send*(rewards.Dining))
```

Any method
--> starts with send with argument type rewards.Dining
--> returns void

❗ Use fully-qualifed class name.

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Any Class or Package*

execution(* send(*))

Any method
--> which name is send and takes a single parameter
--> any return type

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Any Class or Package*

execution(* send(int, ..))

Any method
--> which name is send and takes a first argument is an int type and zero or more argument
--> any return type

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Implementation vs Interfaces*

```
execution(void example.MessageServiceImpl.*(..))
```

Any method
--> in example package and MessageServiceImpl class with zero or more argument
--> returns void

- -> Including any sub-class with same implementation

```
execution(void example.MessageService.send(*))
```

Any method
--> which name is send and taking one argument, in any object implementing MessageService
--> returns void

More flexible choice - works if implementation changes

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Using Annotations*

execution(@javax.annotation.security.RolesAllowed void send*(..)) --> Ideal technique for your own annotations on your own classes

Any method
--> which name starts with send, takes zero or more argument
--> returns void
--> with annotated with @RolesAllowed annotation

```
public interface Mailer {
    @RolesAllowed("USER")
    public void sendMessage(String text);
}
```

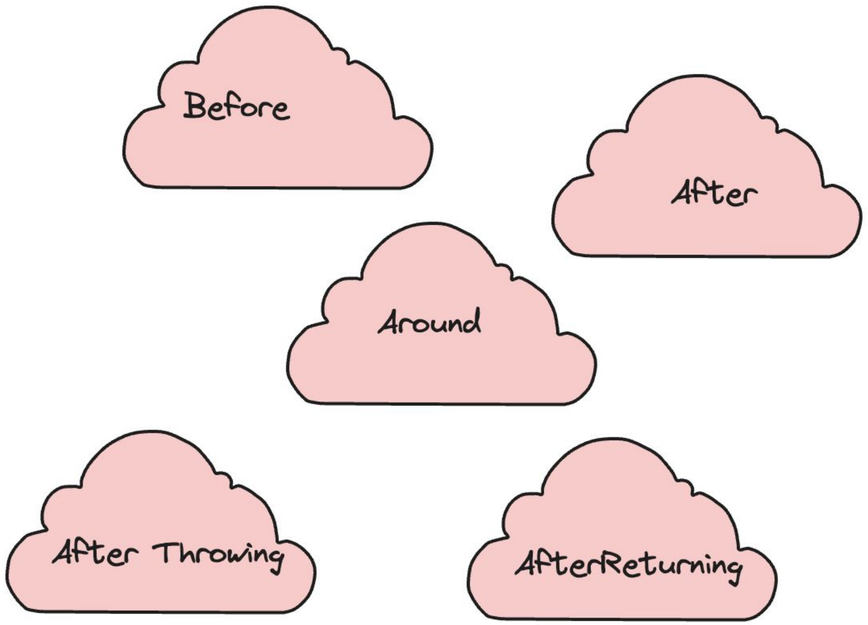
- ✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]
- ✔ * : matches once (ReturnType, Package, class, MethodName, Argument)
- ✔ .. : matches zero or more (Argument, Package)

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

Advice Types

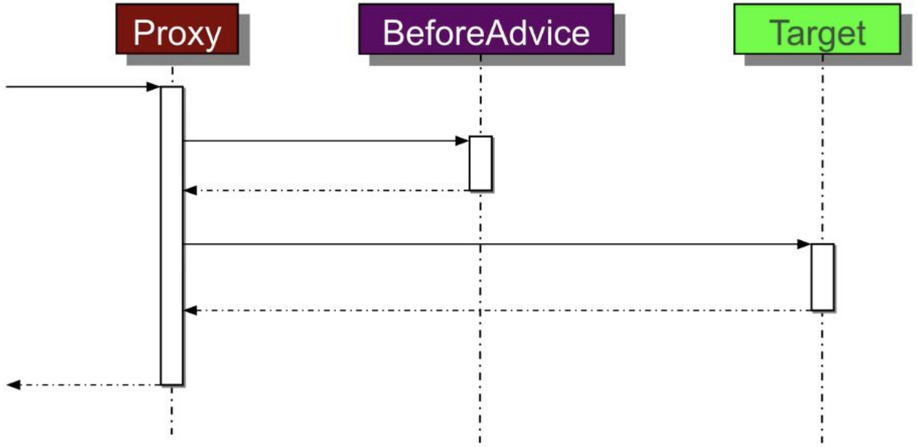
🔔 *Advice*: Advice is a specific code executed at a certain join point.



Advice Types: *@Before*

@Before → org.aspectj.lang.annotation package.

- It runs *before* a join point.
- Proxy is responsible for advice and target execution.
- It can't prevent execution flow proceeding to the join point unless it throws an exception.



Before Advice Example

Requirement

Track calls to all setter methods

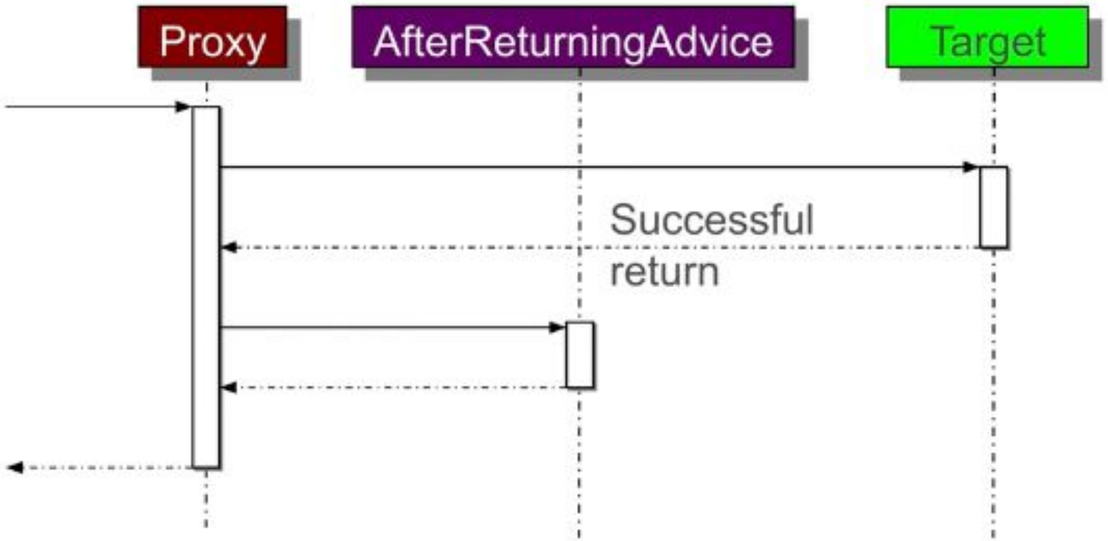
```
@Aspect
@Component
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before "execution(void set*(*))")
    public void trackChange() {
        logger.info("Property about to change…");
    }
}
```

- ❑ Find all set methods with only one argument and void type.
- ❑ Before set method execution, add info log and then call set method.
- ❑ If there is an exception on this trackChange() method, setter methods are not called.

Advice Types: *@AfterReturning*

@AfterReturning → org.aspectj.lang.annotation package.

- ❑ It runs *after* a join point and completes normally
- ❑ Proxy is responsible for advice and target execution.
- ❑ If a method throws an exception this advice does not run. It waits <u>successful</u> return.



AfterReturning Advice Example

Requirement

Audit all operations in the *service* package that return a *Reward* object

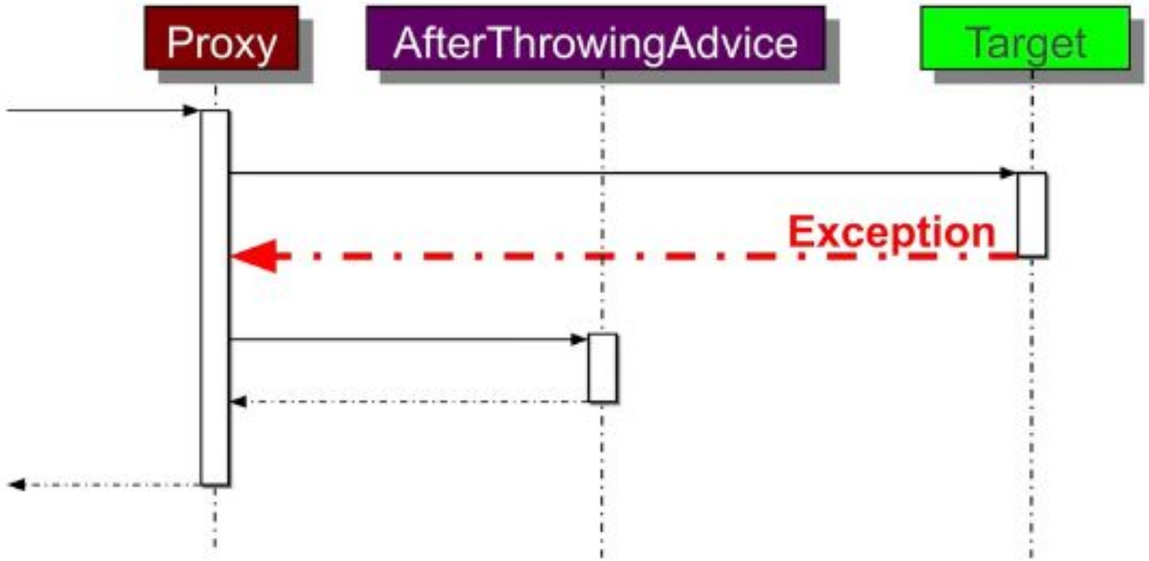
```
@AfterReturning(value="execution(* service..*.*(..))",
                returning="reward")
public void audit(JoinPoint jp, Reward reward) {
    auditService.logEvent(jp.getSignature() +
        " returns the following reward object :"  + reward.toString() );
}
```

- Find all methods with zero or more arguments in any class in any package in service and also returning any type.
- Execute method
- If this method returns Reward object, advice will invoke and logEvent via auditService.
- You can see value and returning parts in the example.

Advice Types: *@AfterThrowing*

@AfterThrowing → org.aspectj.lang.annotation package.

- ❑ It runs if a method throws an exception.
- ❑ Proxy is responsible for advice and target execution.
- ❑ If a method does not throw an exception this advice does not run. It waits exception.
- ❑ 🔔 Very useful for exception handling



AfterThrowing Advice Example

Requirement

Send an email every time a Repository class throws an exception of type DataAccessException

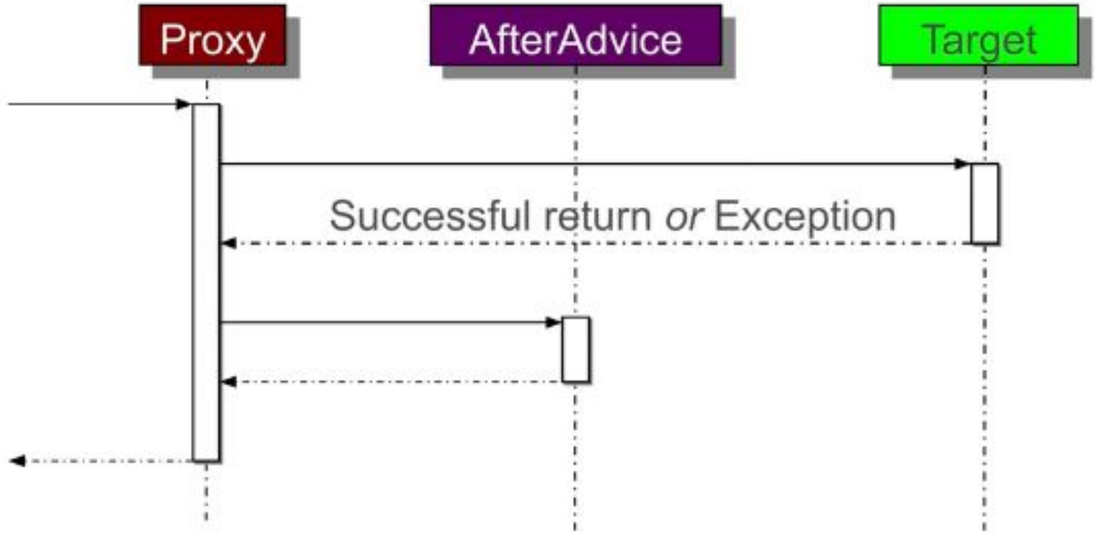
```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
    mailService.emailFailure("Exception in repository", jp, e);
}
```

- Find all methods with zero or more arguments in Repository class in any package and also returning any type.
- Execute method.
- If this method throw DataAccessException, advice will invoke and mail will send.
- You can see value and throwing parts in the example.
- Any child exception of DataAccessException will match the expression.

Advice Types: *@After*

@After → org.aspectj.lang.annotation package.

- It runs regardless of how a join point exits whether by normal or exceptional return.
- Proxy is responsible for advice and target execution.



After Advice Example

Requirement

Track calls to all update methods

```
@Aspect
@Component
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @After "execution(void update*(..))")
    public void trackUpdate() {
        logger.info("An update has been attempted …");
    }
}
```

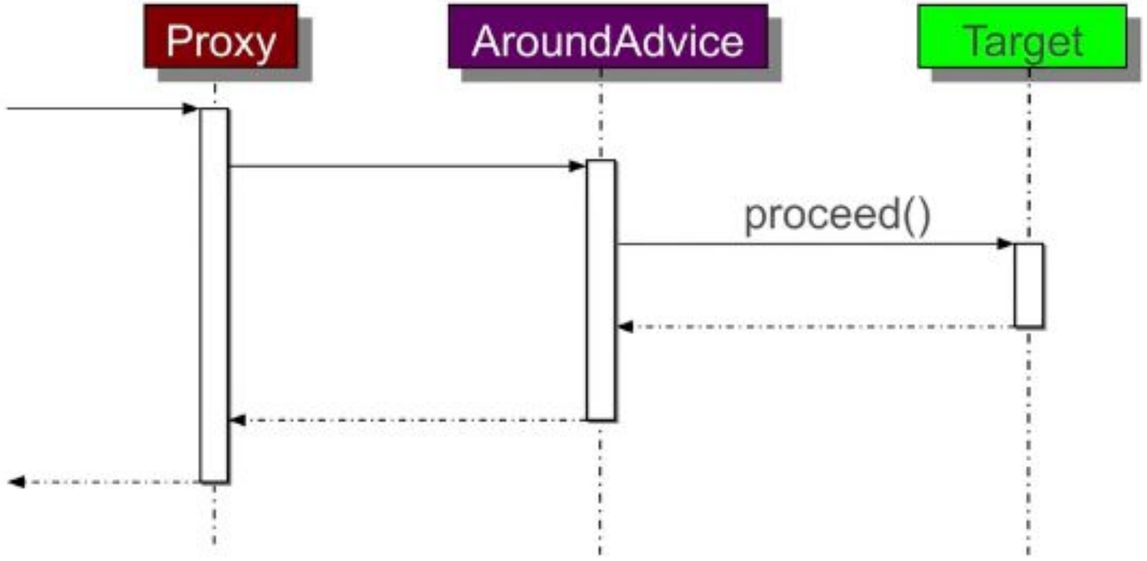
We don't know how the method terminated

- ❑ Find any methods with zero or more arguments and started with update and also returning void.
- ❑ Execute method. It does not matter return value exception or success
- ❑ And then add info log.

Advice Types: *@Around*

@Around → org.aspectj.lang.annotation package.

- It runs before and after a join point.
- It is like a combination of before and after returning aspects.
- Proxy is responsible for only advice execution. Target will execute by advice.
- Most *powerful* and also most *dangerous* advice type.



Around Advice Example

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point) throws Throwable {
    Object value = cacheStore.get(CacheUtils.toKey(point));

    if (value != null) return value;

    value = point.proceed();
    cacheStore.put(CacheUtils.toKey(point), value);
    return value;
}
```

Value exists? If so just return it

Proceed *only* if not already cached

Cache values returned by *cacheable* services

<- - - Requirement

- ❑ Find any methods with zero or more arguments in rewards.service package and also returning any type and annotated with Cacheable.
- ❑ put cache new value
- ❑ Execute method.
- ❑ If cache is found, return value otherwise put cache new value

Limitations of Spring AOP

- Can only advise *non-private* methods.
- Can only apply aspects to Spring Beans.
- Assume method a() calls method b() on the same class/interface
  - Advice will *never* be executed for method b()
  - Advice will *never* be executed for inner-classes

Summary

AOP modularizes *cross-cutting* concerns

An aspect is a module containing cross-cutting behaviour.

- ❑ Annotated with **@Aspect**
- ❑ Behavior is implemented as an “advice” method.
- ❑ Pointcuts select joinpoints(methods)
- ❑ Five advice types
  - ❑ Before, AfterThrowing, AfterReturning, After and Around

AOP is very useful for security, logging, error handling, caching

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Homework: *Aspect Oriented Programming*

Homework

What you will learn:

1. How to write an aspect and weave it into your application
2. Spring AOP using annotations
3. Writing pointcut expressions

Requirements:

REQUIREMENT 1: Create a simple logging aspect for repository find methods.

REQUIREMENT 2: Implement an @Around Advice which logs the time spent in each of your repository update methods.

REQUIREMENT 3: Implement an @AfterThrowing Advice which catch exceptions on database operations.

Todos:

1. Fork project from github : https://github.com/gulumseraslann/spring-training
2. Switch branch to feature/aspect-oriented-programming
3. Create a new branch from this branch, your new branch name should be feature/aspect-oriented-programming-homework
4. There are **13 TODOs** in the project files. Look at these TODOs
5. Please try to do each TODO
6. Please make sure tests are success.
7. Please add the changes and push the solution code in your github repository.

Homework

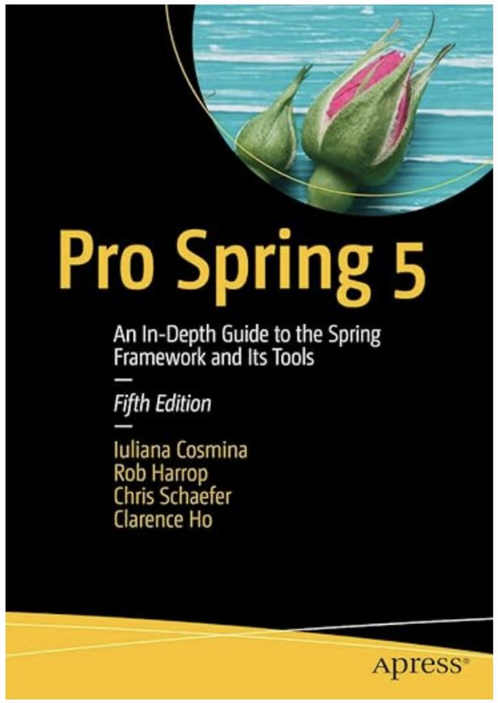
TO-DO:

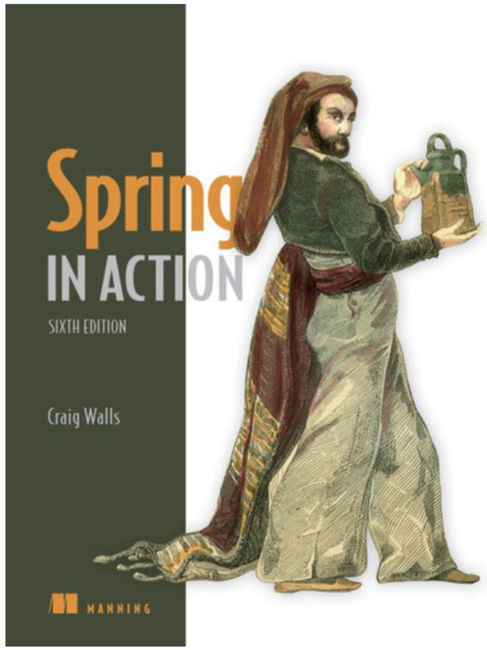
```
com.trendyol.bootcamp.spring.ch05 13 items
  aspect 7 items
    DBExceptionHandlingAspect.java 2 items
      (17, 5) // TODO-10 : Use AOP to log an exception.
      (28, 5) // TODO-11 : Annotate this class as a Spring-managed bean.
    LoggingAspect.java 4 items
      (11, 5)   //  TODO-02: Use AOP to log a message before
                //          any repository's find...() method is invoked.
      (33, 5) // TODO-03: Write Pointcut Expression
      (46, 8) // TODO-07: Use AOP to time update...() methods.
      (57, 8) //  TODO-08: Add the logic to proceed with the target method invocation.
    DBExceptionHandlingAspectTests.java 1 item
  config 1 item
    AspectsConfig.java 1 item
      (8, 4) // TODO-04: Update Aspect related configuration
  service 2 items
    RewardNetworkTests.java 2 items
      (68, 6) // TODO-06: Run this test. It should pass AND you should see TWO lines of
      (73, 6) // TODO-09: Save all your work, and change the expected matches value above from 2 to 4.
  SystemTestConfig.java 1 item
    (13, 4) *  TODO-05:  Make  this  configuration  include  the  aspect  configuration.
  TestConstants.java 2 items
    (5, 5) // TODO-00: In this lab, you are going to exercise the following:
    (10, 5) // TODO-01: Enable checking of console output in our Tests.
```

Final


component
pointcut
annotation
aspect
advice
service
inversion of control
dependency injection
configuration
bean
spring boot
dependency

Book Recommendation





Q & A

THANK YOU

trendyol
learning