**34. AOP Aspect-Oriented Programming Overview**.



Programming technique based on concept of an Aspect. Aspect encapsulates cross-cutting logic (cross-cutting concerns). Concerns means logic/functionality. Aspect can be used at multiple locations. Same aspect/class applied based on configuration.

Benefits of AOP:

* Code for Aspect is defined in a single class.
* Business code in your application is cleaner
* Configurable (no need to change code)

Additional AOP Use cases:

* Most common: logging, security, transaction
* 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 loading? Who is the top user?

Disadvantages:

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

**Aspect:** module of code for a cross-cutting concern

**Advice:** What action is taken and when it should applied

**Join Point:** When to apply code during program execution

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

**Advice Types:**

* Before advice
* After finally advice
* After returning advice
* After throwing advice
* Around advice

**Weaving.** Connecting aspects to target objects to create an advised object. There are different types of weaving: Compile-time, load-time or runtime.

There are two leading AOP Frameworks: Spring AOP, AspectJ

Spring AOP:

* Security, transaction, caching
* Uses run-time weaving aspects

Spring AOP Advantages:

* Simpler to use than AspectJ
* Uses proxy pattern
* Can migrate to AspectJ when using @AspectJ 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 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

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

Link: AspectJ in Action. Aspect Oriented Programming with use Cases

**35. AOP Before Advice Type.**

@Before - Run custom code BEFORE the Target Object method call

@AfterReturning – Run custom code After the Target Object method call

@Before use cases:

* Most common: logging, security, transactions
* Audit logging: who, what, when, where
* API Management: how many times has a method been called

Need to download AspectJ JAR file even though we are using Spring AOP (Spring uses AspectJ’s annotations and classes)

Development Process:

1. Create target object
2. Create Spring Java Config class



ComponentScan – scans for components and aspects

1. Create an Aspect with @Before

**36. AOP Pointcut Expressions - Match Methods and Return Types.**

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

Spring AOP uses AspectJ’s pointcut expression language. Execution pointcut applies to execution of methods. Pointcut pattern can have wildcards

execution(modifiers-pattern? return-type-pattern declaring-type-pattern?

method-name-pattern(param-pattern) throws-pattern?)

You can use asterisk (everything) or the star as a wildcard.



**37. AOP Pointcut Expressions - Match Method Parameter Types.**

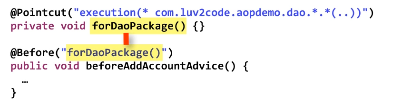
Parameter pattern wildcards:

* () – 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

**38. AOP Pointcut Declarations.**

Development process (reuse pointcut expressions):

1. Create a pointcut declaration
2. Apply pointcut declaration advice(s)



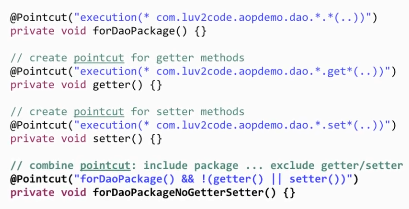
**Combine pointcuts** using logic operators:

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

It works like an if statements. Execution happens only if it evaluates to true.

Development process:

* Create a pointcut declarations
* Combine
* Apply to advices



**39. AOP Ordering Aspects**

Aspect order is undefined.

To control order:

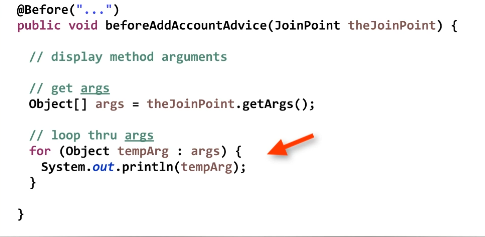
* Refactor: Place advices in separate Aspects
* Control order on Aspects using the @Order annotation (in parentheses you give the precedence number)

Lower numbers have higher precedence. Range: Integer.MIN\_VALUE to Integer.MAX\_VALUE. Negative numbers are allowed. Does not have to be consecutive.

**40. AOP JoinPoints.**

Access and display Method Signature. JoinPoint will give us information about the method that we’re actually executing.





**41 AOP AfterReturning Advice Type.**

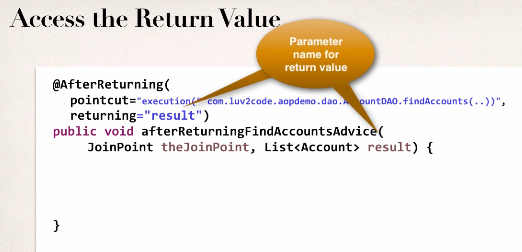
AfterReturning Advice – run after the method completed for successful execution.

Use cases:

* Most common: logging, security, transactions
* Audit logging: who, what, when, where
* Post-processing Data: Post process the data before returning to caller

Use @AfterReturning annotation

Access return value:

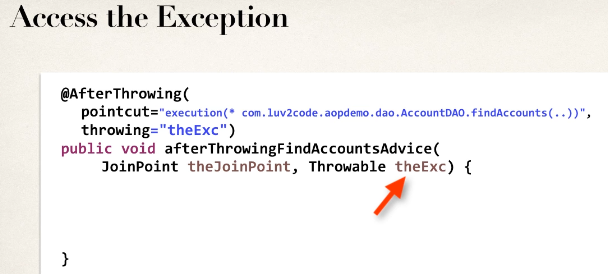


Modify return value (Post process).

**42. AOP AfterThrowing Advice Type.**

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



At this point, we are only intercepting the exception. However, the exception is still propagated.

If you want to stop the exception propagation then use @Around advice.

**43. AOP After (finally) Advice Type.**

@After runs after a method is completed. Regardless of outcome / exceptions (works like “finally” block). After will execute before AfterThrowing.

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

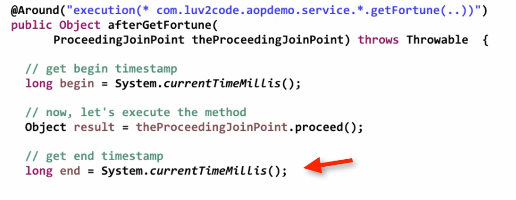
After advice does not have access to the exception. If you need exception, then use @AfterThrowing advice

**44. AOP Around Advice Type** – Before and after target

Use cases:

* Most common: logging, auditing, security
* Pre-processing and post-processing
* Instrumentation / profiling code
* Managing exceptions

When using @Around advice you will get a reference to a ProceedingJoinPoint. This is a handle to the target. Your code can use ProceedingJoinPoint to execute target method



Resolving Print Order Issue:

* **Root cause:**
  + The data is printing to two different output streams
  + Spring is printing to the logger output stream
  + System.out.println is printing to the standard output stream
* **Solution**
  + To have everything in order, you should send to same output stream