Going Further with AOP

1. More on pointcut expressions
2. Types of advice
3. Introductions

Demo project: **`demo-aop-additional`**

1. More on Pointcut Expressions

- Pointcut designators wrt target
- Pointcut designators wrt source
- Accessing arguments via `JoinPoint`
- Argument binding

Pointcut Designators wrt Target

Designator	Description	Wildcards?
`execution`	Matches execution of methods. This is the most commonly used pointcut expression. You can specify the visibility, annotations, return type, package, class name, method name, argument types, and exceptions.	* ..
`target`	Matches calls on beans of the specified type. E.g. target(com.osl.AccountSvcImpl) matches calls on instances of AccountSvcImpl.	(no)
`@target`	Matches calls on beans that have the specified annotation. Typically used for binding. E.g. @target(org.springframework.transaction.annotation.Transactional) matches calls on any beans that are annotated with @Transactional.	(no)
`bean`	Matches calls on beans that have the specified id/name. Useful for vertical slicing. E.g. bean(account*) matches calls to all beans whose id/name starts with account, such as accountController, accountService, accountRepository.	*
`@annotation`	Matches calls on methods that have the specified annotation. E.g. @annotation(org.springframework.transaction.annotation.Transactional) matches calls to any methods that are annotated with @Transactional.	(no)
`args`	Matches calls that receive the specified argument types. Typically used for binding. E.g. args(Serializable, .., String) matches calls that receive a Serializable first argument, a String as the last argument, and anything else in between.	..
`@args`	Matches calls that are called with the specified annotated arguments. E.g. @args(MyAnn1, .., MyAnn2) matches calls that receive a @MyAnn1-annotated first argument, a @MyAnn2-annotated last argument, and anything else in between.	..

Pointcut Designators wrt Source

Designator	Description	Wildcards?
`within`	Matches calls into beans that have the specified type. E.g. within(com.osl..Account*) matches calls into any type that starts with Account, located in the com.osl package (or sub-package).	* ..
`@within`	Matches calls into beans that have the specified annotation. E.g. @within(StartsTransaction) matches calls into beans that are annotated with StartsTransaction.	* ..
`this`	Matches calls into Spring proxy objects that have the specified type. E.g. this(com.osl.AccountSvc) matches calls into proxies that implement AccountSvc.	(no)

Accessing Arguments via `JoinPoint` (1 of 2)

- Consider the following class in our application:

```
@Component
public class BankServiceImpl implements BankService {

    public void doDeposit(int accountID, double amount) { … }
    public void doWithdraw(int accountID, double amount) { … }
    …
}
```

BankServiceImpl.java

- How can we define a pointcut to intercept methods and also access the arguments passed in?

Accessing Arguments via `JoinPoint` (2 of 2)

- One approach is to use a `JoinPoint` to access arguments

```
@Aspect
@Component
public class MyAspect {

    @Before("execution(* BankService.do*(int, double))")
    public void logBizOp(JoinPoint jp) {
      System.out.println("***Method: "    + jp.getSignature().getName() +
                         " account id: "  + jp.getArgs()[0] +
                         " amount: "      + jp.getArgs()[1]);
    }
    …
}
```

MyAspect.java

- This approach isn't very clear, and it's not type-safe

Argument Binding (1 of 2)

- A better approach is to bind arguments in a Pointcut:
  - `this` – gives access to currently-executing object (i.e. proxy)
  - `target` – gives access to the target object (i.e. your object)
  - `args` – gives access to method arguments

- Enables you to write POJO advice methods
  - Alternative to working with `JoinPoint` directly
  - (You can still use `JoinPoint` as well, if you want to)

Argument Binding (2 of 2)

- Example:

```
@Aspect
@Component
public class MyAspect {

    @Before("execution(* BankService.do*(int, double)) && " +
            "target(service) && args(id, amt)")
    public void logBizOp2(BankService service, int id, double amt) {

        // Levy 10% punitive fee on poor customer. Mwa, mwa, mwa...
        service.levyFee(id, amt * 0.10);
    }
    …
}
```

MyAspect.java

- To see the effect, run `App_MoreAspects.java`

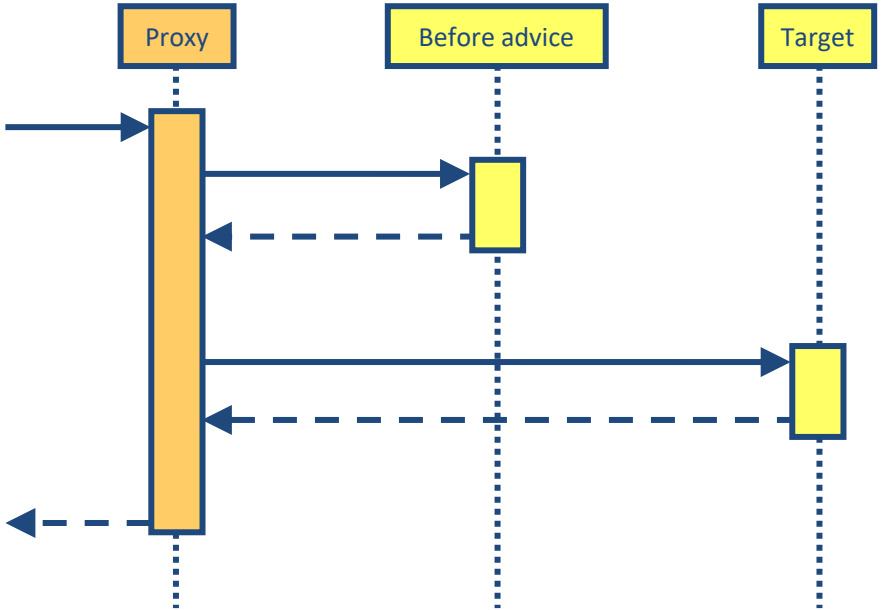
2. Types of Advice

- Before advice
- After returning advice
- After throwing advice
- After advice
- Around advice

Before Advice (1 of 3)

- Annotate advice method with `@Before`
  - Proxy intercepts method call on target object
  - Calls `@Before` advice method
  - Then calls target method (unless advice method threw exception)

Before Advice (2 of 3)



Before Advice (3 of 3)

- Do a security check on every `@Secured` operation
  - Here's an example of a `@Secured` operation

```
@Secured(allowedRoles={"manager","clerk"})
public void chargePenaltyFee() { … }
```

  - This pointcut matches calls to `@Secured`-annotated methods
    - Argument binding makes available the proxy and `@Secured` annotation object

```
@Pointcut("execution(* *(..)) &&
           this(object) &&
           @annotation(secured)")
public void callToSecuredMethod(Object object, Secured secured) {}
```

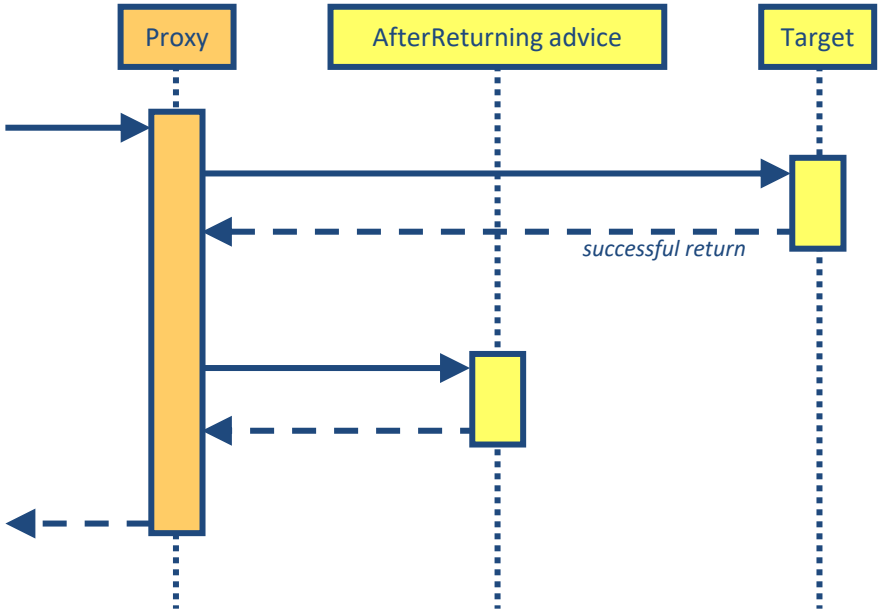
  - This *Before* advice method runs the security check
    - Checks whether the user is in an allowed role

```
@Before("callToSecuredMethod(object, secured)")
public void doSecurityCheck(JoinPoint jp, Object object, Secured secured) {
    checkPermission(jp, object, secured.allowedRoles());
}
```

After Returning Advice (1 of 3)

- Annotate advice method with `@AfterReturning`
  - Proxy calls `@AfterReturning` advice method after the target method returns successfully
  - Gives access to returned object

After Returning Advice (2 of 3)



After Returning Advice (3 of 3)

- Audit all operations that return a `@Secret`-annotated type
  - Here's an example of such an operation

```
public @Secret String getMyPassword() { … }
```

  - This pointcut matches methods with `@Secret`-annotated return

```
@Pointcut("execution((@myannotationspackage.Secret *) *(..))")
public void methodReturningSecretValue() {}
```

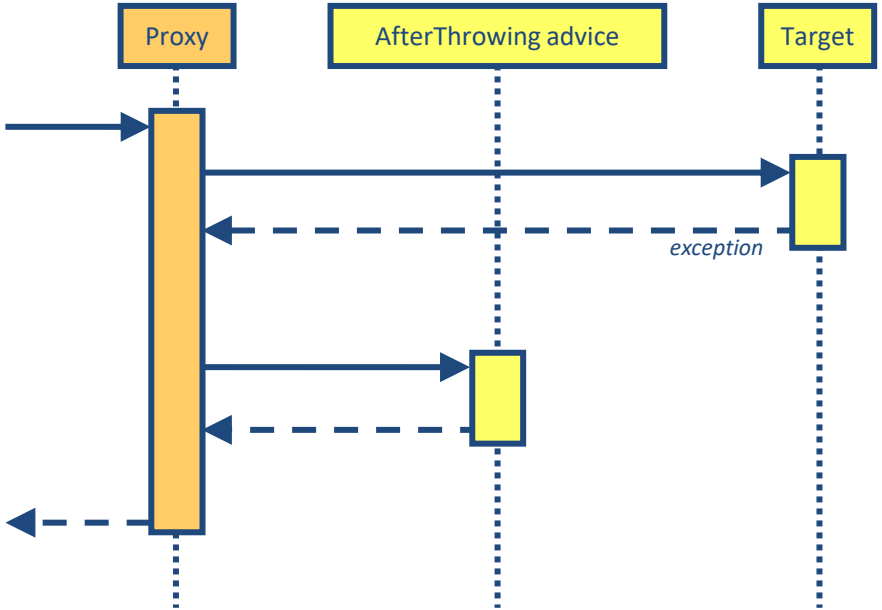
  - This *After Returning* advice method audits the return value
    - `pointcut` – specifies the pointcut
    - `returning` – gives access to the actual returned object

```
@AfterReturning(pointcut="methodReturningSecretValue()",
                returning="retval")
public void auditSecretReturnValue(JoinPoint jp, Object retval) {
    System.out.println("Method: "     + jp.getSignature().getName() +
                       " returning: " + retval);
}
```

After Throwing Advice (1 of 3)

- Annotate advice method with `@AfterThrowing`
  - Proxy calls `@AfterThrowing` advice method if target method threw an exception
  - Gives access to exception object

After Throwing Advice (2 of 3)



After Throwing Advice (3 of 3)

- Log all operations that throw an `IOException`
  - Here's an example of such an operation

```
public void writeSomeDataToFile() throws IOException { … }
```

  - This pointcut matches methods any methods, not very interesting!

```
@Pointcut("execution(*(..))")
public void anyMethod() {}
```

  - This *After Throwing* advice method logs the exception
    - `pointcut` – specifies the pointcut
    - `throwing` – gives access to exception

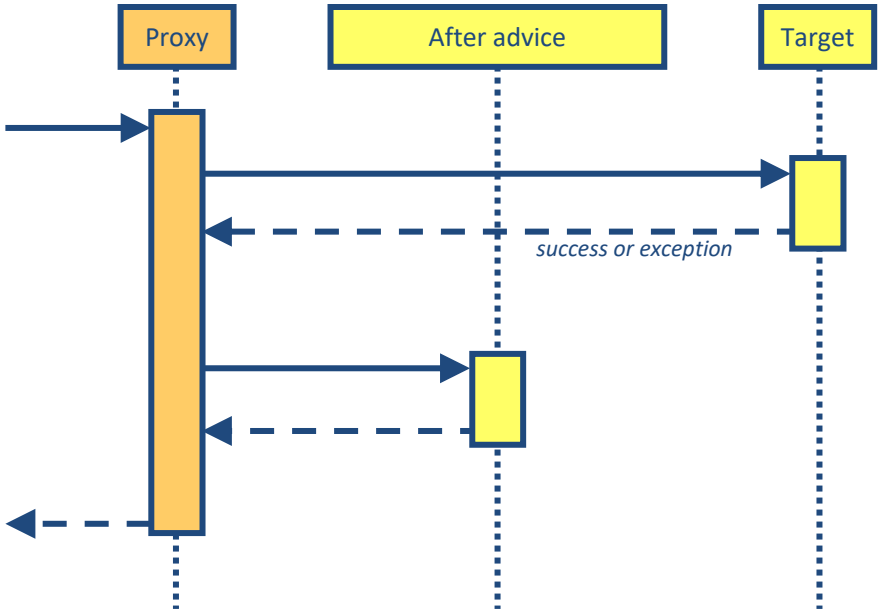
```
@AfterThrowing(pointcut="anyMethod()",
               throwing="ex")
public void logIOException(JoinPoint jp, IOException ex) {
    System.out.println("Method: "     + jp.getSignature().getName() +
                       " exception: " + ex.getMessage());
}
```

Type of arg acts as a filter

After Advice (1 of 2)

- Annotate advice method with `@After`
  - Proxy calls `@After` advice method after target method ends (whether the method returned normally or threw an exception)
  - Note: you cannot access return-object or exception-object

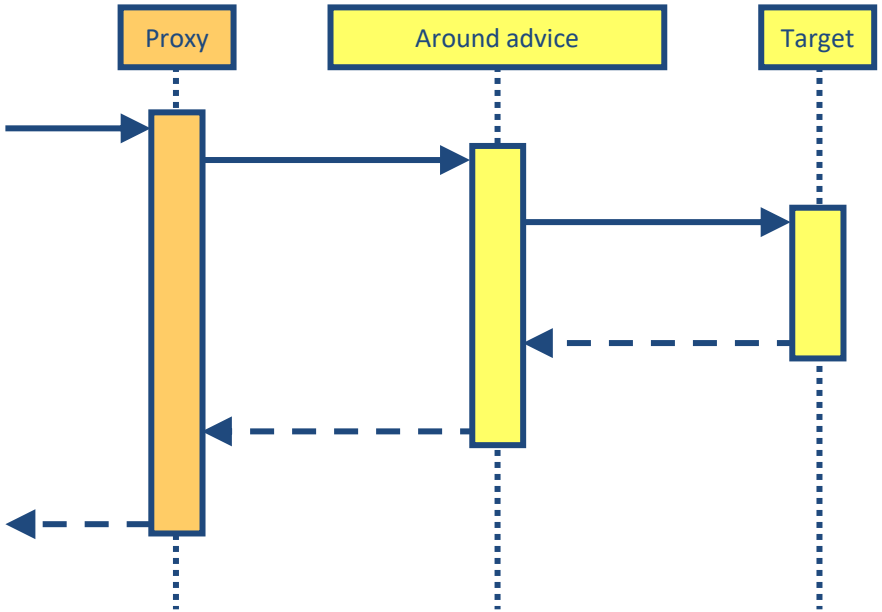
After Advice (2 of 2)



Around Advice (1 of 3)

- Annotate advice method with `@Around`
  - Most complex (and powerful) type of advice
  - Executes around target method
  - Useful for transactions, caching, etc

Around Advice (2 of 3)



Around Advice (3 of 3)

- Time how long operations take
  - Here's an example of an operation

```
public String doSomething(int someParam, String anotherParam) { … }
```

  - This pointcut matches any methods

```
@Pointcut("execution(* *(..))")
public void anyMethod() {}
```

  - This *Around* advice method times the operation
    - The advice method must take a `ProceedingJoinPoint` arg
    - Call its `proceed()` to invoke target method

```
@Around("anyMethod()")
public Object timeMethod(ProceedingJoinPoint pjp) {
    // Start timer.
    Object retval = pjp.proceed();
    // Stop timer.
    return retval;
}
```

3. Introductions

- Introduction ☺
- Defining an introduction interface
- Implementing the introduction mix-in
- Creating the mix-in aspect
- Testing introductions

Introduction ☺

- Introductions are an important part of Spring AOP
  - Enable you to add new functionality to an existing class
  - … by introducing any interface to an existing class

- Useful for adding cross-cutting behaviour
  - Where it would be infeasible for objects to inherit the functionality from a base class
  - … because Java doesn't allow multiple inheritance
  - … and you don't want to implement the behaviour in each class

Defining an Introduction Interface

- Step 1: Define an interface
  - Describes the functionality you want to introduce

- Example:
  - Define a `CallTracker` interface that tracks normal/failing method calls

```
public interface CallTracker {
    void   markNormal();
    void   markFailing();
    int    countNormalCalls();
    int    countFailingCalls();
    String describe();
}
```

CallTracker.java

Implementing the Introduction Mix-In

- Step 2: Implement the introduction
  - This is often referred to as a "mix-in"
- Example - define a `CallTrackerImpl` class

```
@Component
public class CallTrackerImpl implements CallTracker {

    private int normalCalls, failingCalls;

    public void markNormal()  { normalCalls++;  }
    public void markFailing() { failingCalls++; }

    public int countNormalCalls()  { return normalCalls;  }
    public int countFailingCalls() { return failingCalls; }

    public String describe() { return toString(); }

    @Override
    public String toString() {
        return "CallTrackerImpl: " +
               " normal calls="    + normalCalls +
               " failing calls="   + failingCalls;
    }
}
```

CallTrackerImpl.java

Creating the Mix-In Aspect (1 of 2)

- Step 3a: Define an aspect class to apply the mix-in
  - Define a mix-in field and annotate with `@DeclareParents`
    - `value` – The types of target class to apply the mix-in to
    - `defaultImpl` – Concrete implementation type of the mix-in

```
@Aspect
@Component
public class CallTrackerAspect {

    @DeclareParents(value="mypackage.*", defaultImpl=CallTrackerImpl.class)
    public CallTracker mixin;

  // Plus pointcuts and advice methods, see next slide…

}
```

CallTrackerAspect.java

Creating the Mix-In Aspect (2 of 2)

- Step 3b: Define an aspect class to apply the mix-in
  - Define pointcuts and advice methods in the aspect class
    - E.g. a pointcut that matches all calls on beans in our package
    - E.g. `AfterReturning` and `AfterThrowing` advice methods

```
@Aspect
@Component
public class CallTrackerAspect {

    // Mix-in declaration, as per previous slide.

    @Pointcut("execution(* mypackage.*.*(..))")
    private void opCall() {}

    @AfterReturning(pointcut="opCall() && this(theProxyObj)")
    public void trackNormalCall(CallTracker theProxyObj) {
        theProxyObj.markNormal();
    }

    @AfterThrowing(pointcut="opCall() && this(theProxyObj)", throwing="t")
    public void trackFailingCall(CallTracker theProxyObj, Throwable t) {
        theProxyObj.markFailing();
    }
}
```

CallTrackerAspect.java

Testing Introductions

- Step 4: Write a test application to verify introductions work

```
@SpringBootApplication
@EnableAspectJAutoProxy
public class App_Introductions {

  public static void main(String[] args) {

     ApplicationContext context = SpringApplication.run(App_Introductions.class);
     MyService bean = context.getBean(MyService.class);

     try {
          bean.goodOp1();
          bean.goodOp2();
          bean.badOp();

     } catch (Exception ex) {
          System.out.println(ex.getMessage());
     }

     CallTracker t = (CallTracker) bean;
     System.out.println("Bean tracking details: " + t.describe());
  }
}
```

App_Introductions.java

Summary

- More on pointcut expressions
- Types of advice
- Introductions