



Introducing Aspect Oriented Programming

Developing Aspects with Spring AOP

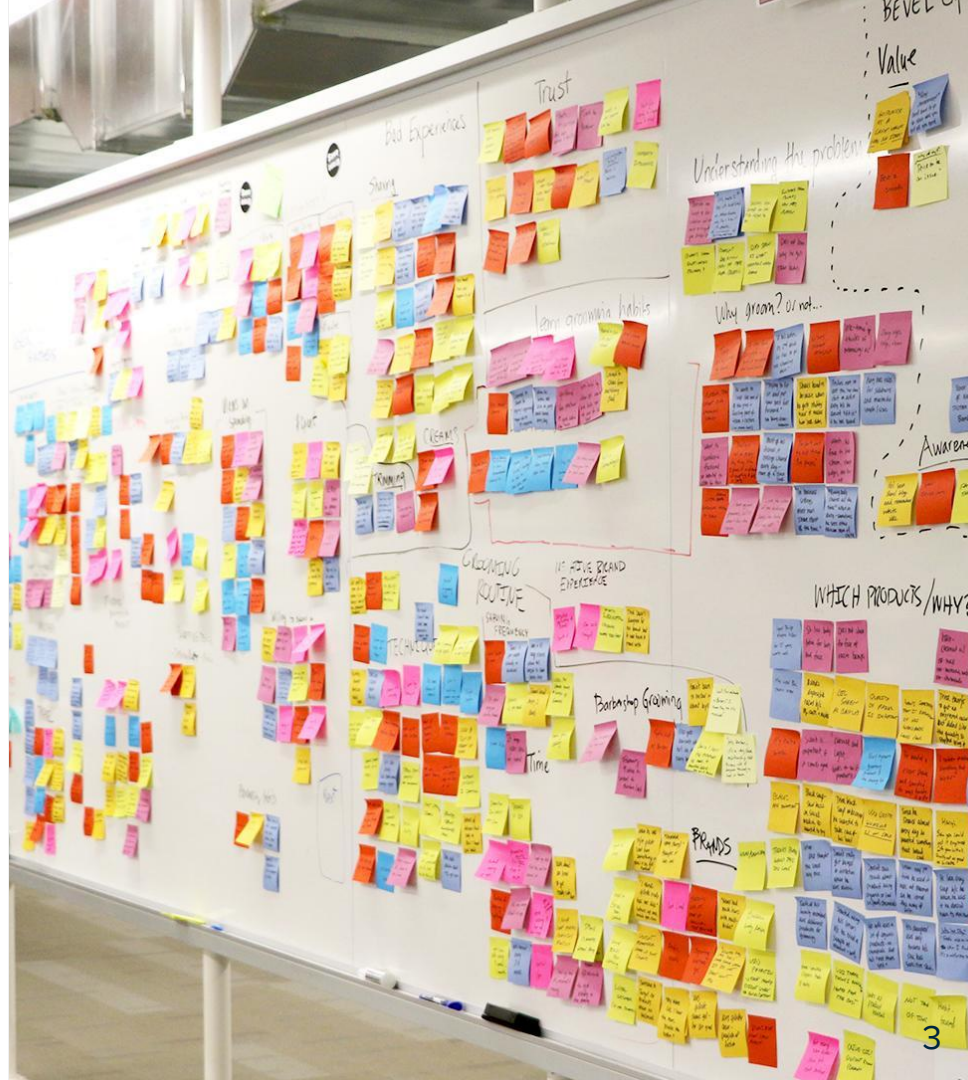
Objectives

After completing this lesson, you should be able to

- Explain the concepts behind AOP and the problem that it solves
- Implement and deploy advices using Spring AOP
- Use AOP Pointcut Expressions
- Explain different types of Advice and when to use them

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics



What Problem Does AOP Solve?

- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns

What are Cross-Cutting Concerns?

- Generic functionality that is needed in many places in your application
- Examples
 - Logging and Tracing
 - Transaction Management
 - Security
 - Caching
 - Error Handling
 - Performance Monitoring
 - Custom Business Rules

An Example Requirement

- Perform a role-based security check before every application method



A sign this requirement is a cross-cutting concern

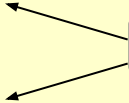
Implementing Cross Cutting Concerns Without Modularization

- Failing to modularize cross-cutting concerns leads to two problems
 - Code tangling
 - A coupling of concerns
 - Code scattering
 - The same concern spread across modules

Symptom #1: Code Tangling

```
public class RewardNetworkImpl implements RewardNetwork {  
    public RewardConfirmation rewardAccountFor(Dining dining) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
    }  
}
```

Mixing of concerns



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

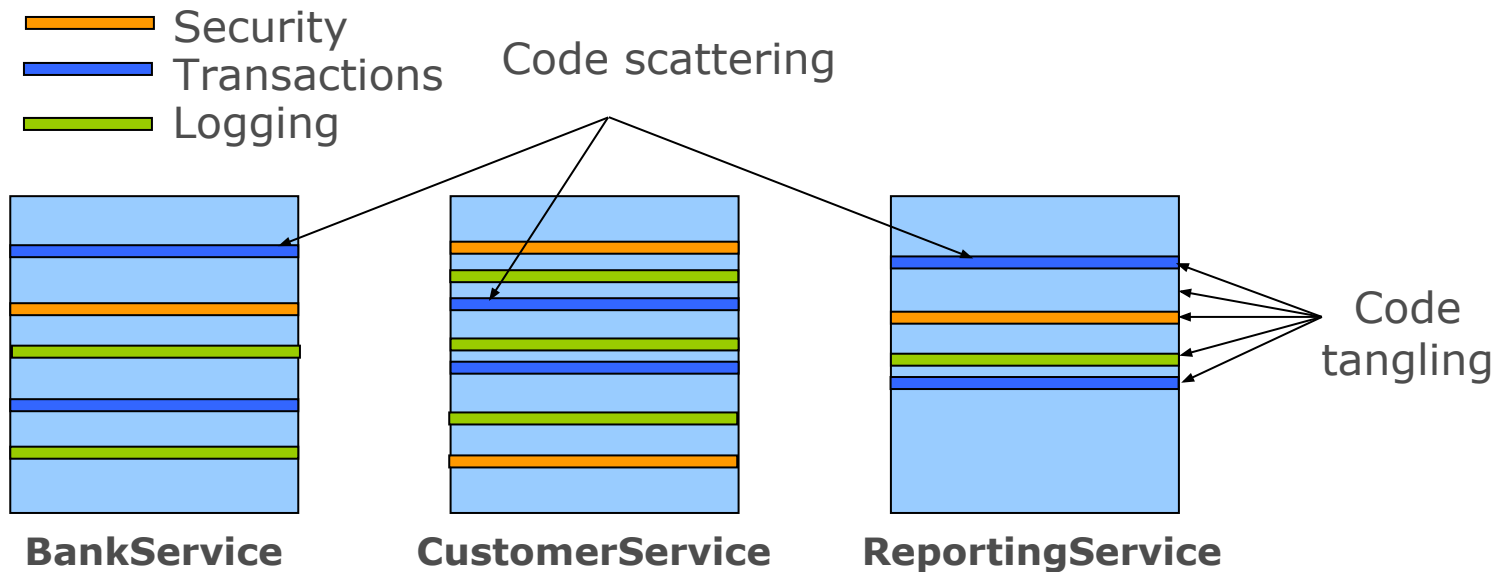

Symptom #2: Code Scattering

```
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();  
        }  
        ...  
    }  
}
```

System Evolution Without Modularization



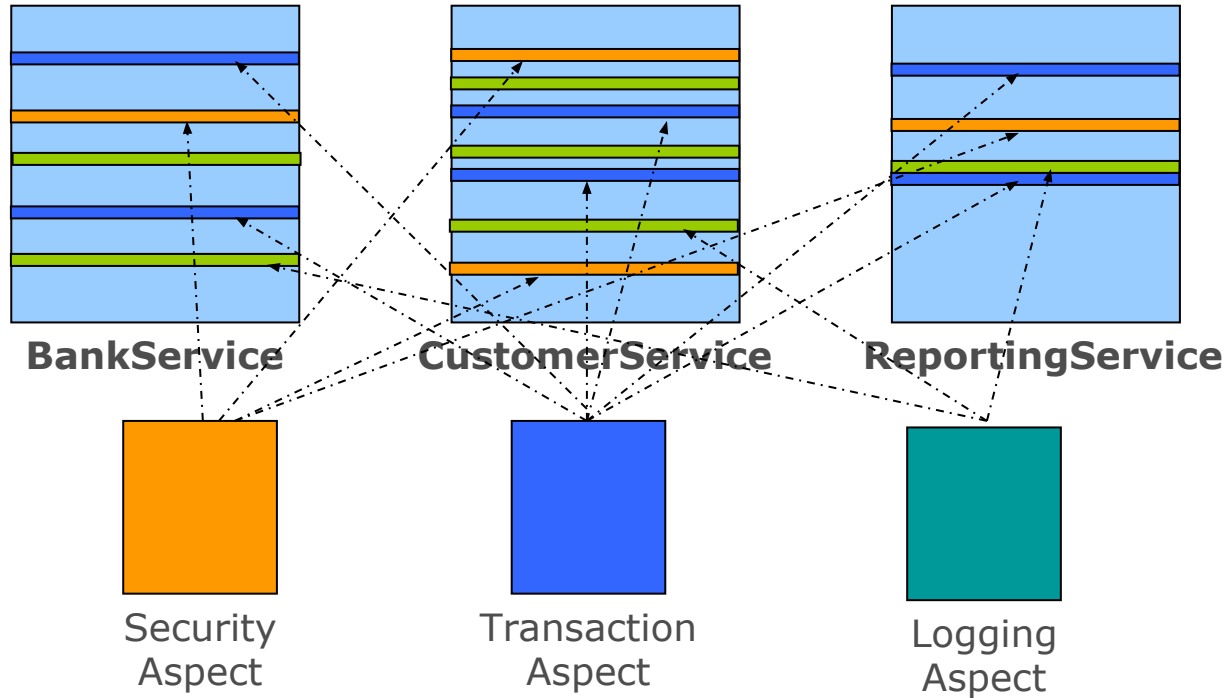
Aspect Oriented Programming (AOP)

- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns
 - To avoid code tangling
 - To eliminate code scattering

How AOP Works

- Implement your mainline application logic
 - Focusing on the core problem
- Write aspects to implement your cross-cutting concerns
 - Spring provides many aspects out-of-the-box
- Weave the aspects into your application
 - Adding the cross-cutting behaviours to the right places

System Evolution: AOP based



Leading AOP Technologies

- AspectJ
 - Original AOP technology (first version in 1995)
 - A full-blown Aspect Oriented Programming language
 - Uses byte code modification for aspect weaving
- Spring AOP
 - Java-based AOP framework with AspectJ integration
 - Uses dynamic proxies for aspect weaving
 - Focuses on using AOP to solve enterprise problems
 - The focus of this session



[Spring Framework Reference – Aspect Oriented Programming](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop)

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop>

Agenda

- What Problem Does AOP Solve?
- **Core AOP Concepts**
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics



Core AOP Concepts

- **Join Point**
 - A point in the execution of a program such as a method call or exception thrown
- **Pointcut**
 - An expression that selects one or more Join Points
- **Advice**
 - Code to be executed at each selected Join Point
- **Aspect**
 - A module that encapsulates pointcuts and advice
- **Weaving**
 - Technique by which aspects are combined with main code

Core AOP Concepts: Proxy

- **Proxy**
 - Someone who stands in place of someone else
 - Such as at an auction or an official meeting
- **AOP Proxy**
 - An “enhanced” class that stands in place of your original
 - With extra behavior (Aspect) added (woven) into it

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- **Quick Start**
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics



AOP Quick Start

- Consider this basic requirement

Log a message every time a property is about to change

- How can you use AOP to meet it?

An Application Object Whose Properties Could Change

```
public class SimpleCache  
    implements Cache {  
    private int cacheSize;  
    private String name;
```

```
    public SimpleCache(String beanName) { name = beanName; }
```

```
    public void setCacheSize(int size) { cacheSize = size; }
```

```
    ...
```

```
    public String toString() { return name; }    // For convenience later  
}
```

```
public interface Cache {  
    public void setCacheSize(int size);  
}
```

Implement the Aspect

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

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

Configure Aspect as a Bean

- Must *enable* use of **@Aspect**
 - Spring uses all beans annotated with **@Aspect** as aspects

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan(basePackages="com.example.aspects")
public class AspectConfig {
    ...
}
```

Configures Spring to recognize and use @Aspect

Include the Aspect Configuration

Include aspect configuration

```
@Configuration
@Import(AspectConfig.class)
public class MainConfig {
    @Bean
    public Cache cacheA() { return new SimpleCache("cacheA"); }

    @Bean
    public Cache cacheB() { return new SimpleCache("cacheB"); }

    @Bean
    public Cache cacheC() { return new SimpleCache("cacheC"); }
}
```

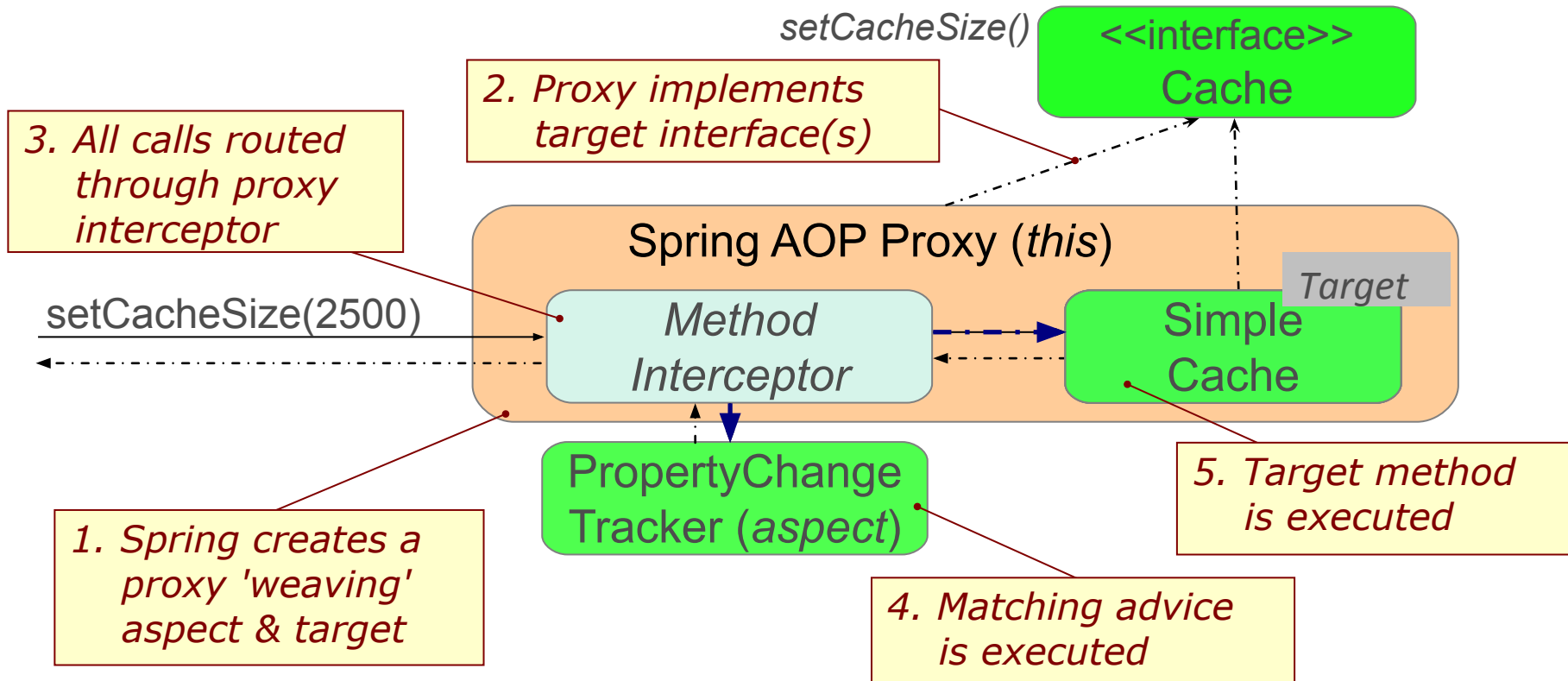
Test the Application

```
ApplicationContext context = SpringApplication.run(MainConfig.class);
```

```
@Autowired @Qualifier("cacheA");  
private Cache cache;  
...  
cache.setCacheSize(2500);
```

INFO: Property about to change...

How Aspects are Applied



Which Setter is Proxied?

```
public class DatabaseCache implements Cache
```

```
{
```

```
    private int cacheSize;  
    private DataSource dataSource;  
    private String name;
```

```
    public SimpleCache(String beanName) { name = beanName; }
```

```
    public void setCacheSize(int size) { cacheSize = size; }
```

```
    public void setDataSource(DataSource ds) { dataSource = ds; }
```

```
    ...
```

```
    public String toString() { return name; }    // For convenience later
```

```
}
```

```
public interface Cache {  
    public void setCacheSize(int size);  
}
```

YES – on **Cache** interface

NO – *not* on **Cache** interface

Tracking Property Changes – With Context

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

    @Before("execution(void set*(*))")
    public void trackChange(JoinPoint point) {
        String methodName = point.getSignature().getName();
        Object newValue = point.getArgs()[0];
        logger.info(methodName + " about to change to " +
            newValue + " on " +
            point.getTarget());
    }
}
```

JoinPoint parameter provides context about the intercepted point

toString() returns bean-name

INFO: setCacheSize about to change to 2500 on cacheA

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- **Defining Pointcuts**
- Implementing Advice
- Lab
- Advanced Topics



Defining Pointcuts

- Spring AOP uses AspectJ's pointcut expression language
 - For selecting where to apply advice
- Complete expression language reference available at
 - <http://www.eclipse.org/aspectj/docs.php>
- Spring AOP supports a practical subset



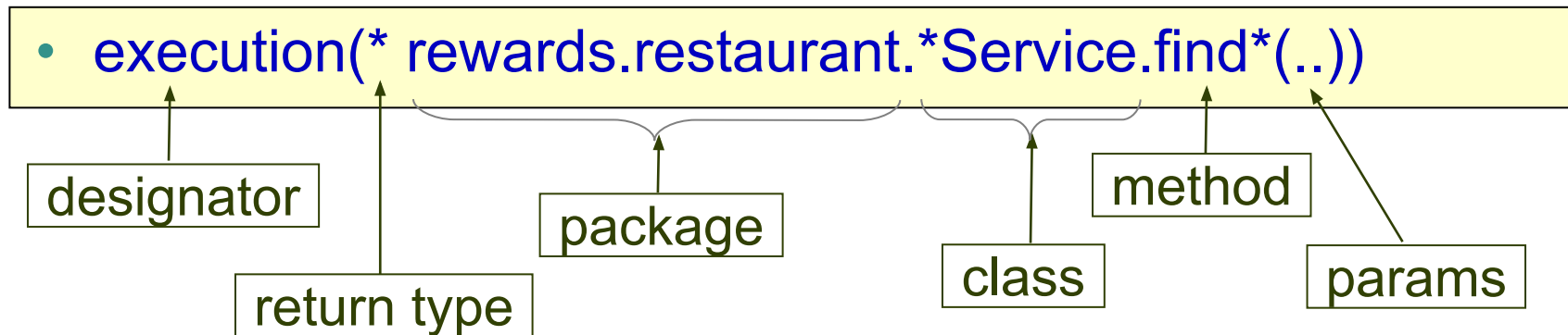
See: [Spring Framework Reference – Declaring a Pointcut](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api-pointcuts)

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api-pointcuts>

Common Pointcut Designator

- `execution(<method pattern>)`
 - The method must match the pattern
- Can chain together to create composite pointcuts
 - `&&` (and), `||` (or), `!` (not)
 - `execution(<pattern1>) || execution(<pattern2>)`
- Method Pattern
 - `[Modifiers] ReturnType [ClassType]
 MethodName (Arguments) [throws ExceptionType]`

Example Expression



Wildcards:

- `*` – matches once (return type, package, class, method name, argument)
- `..` – matches zero or more (arguments or packages)

Execution Expression Examples

Any Class or Package

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

- Any method starting with send that takes a single Dining parameter and has a void return type
- Note use of *fully-qualified classname*

`execution(* send(*))`

- Any method named send that takes a single parameter

`execution(* send(int, ..))`

- Any method named send whose first parameter is an int (the “..” signifies 0 or more parameters may follow)

Execution Expression Examples

Implementations vs Interfaces

- Restrict by *class*

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

- Any void method in the *MessageServiceImpl* class
 - Including any sub-class
- But will be ignored if a different implementation is used

- Restrict by *interface*

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

- Any void *send* method taking one argument, in any object implementing *MessageService*
- More flexible choice – works if implementation changes

Execution Expression Examples

Using Annotations

`execution(@javax.annotation.security.RolesAllowed void send*(..))`

- Any void method whose name starts with “*send*” that is annotated with the *@RolesAllowed* annotation

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

- Ideal technique for your own annotations on your own classes
 - Matches if annotation is present

Execution Expression Examples

Working with Packages

`execution(* rewards.*.restaurant.*.*(..))`

- There is one directory between rewards and restaurant

`execution(* rewards..restaurant.*.*(..))`

- There may be several directories between rewards and restaurant

`execution(* *.restaurant.*.*(..))`

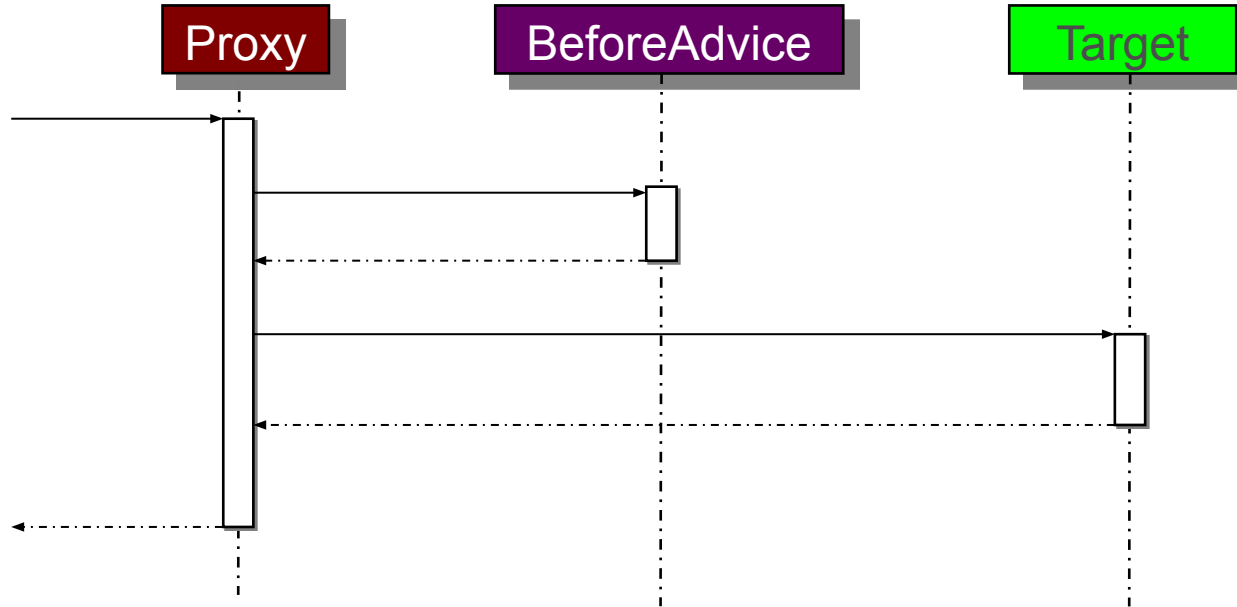
- Any sub-package called restaurant

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- **Implementing Advice**
- Lab
- Advanced Topics



Advice Types: Before



Before Advice Example

- Use *@Before* annotation

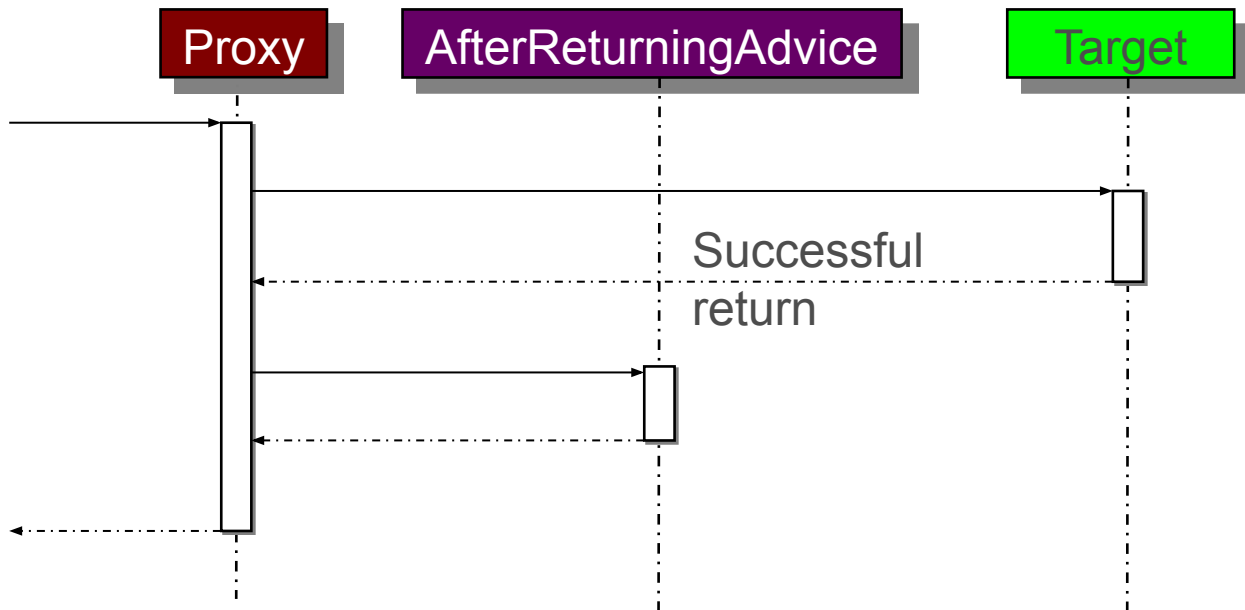
Track calls to all setter methods

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

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

- **Note:** if the advice throws an exception, target will not be called – this is a valid use of a *Before Advice*

Advice Types: After Returning



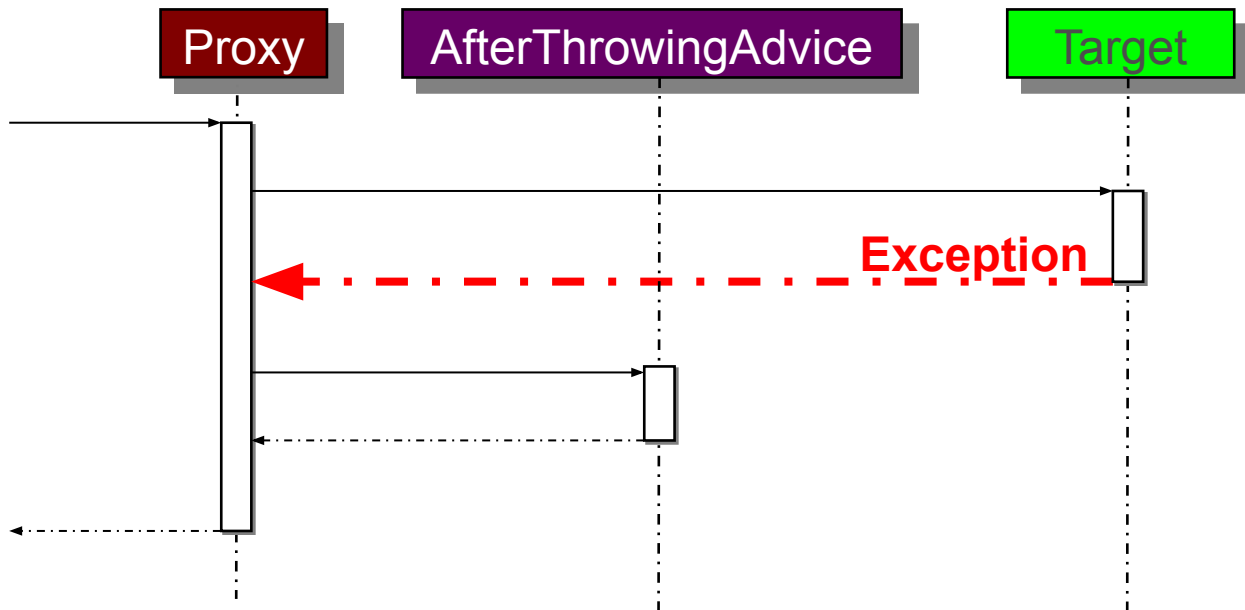
After Returning Advice - Example

- Use `@AfterReturning` annotation with the *returning* attribute

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() );  
}
```


Advice Types: After Throwing



After Throwing Advice - Example

- Use `@AfterThrowing` annotation with the *throwing* attribute
 - Only invokes advice if the right exception type is thrown

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);
}
```

After Throwing Advice - Propagation

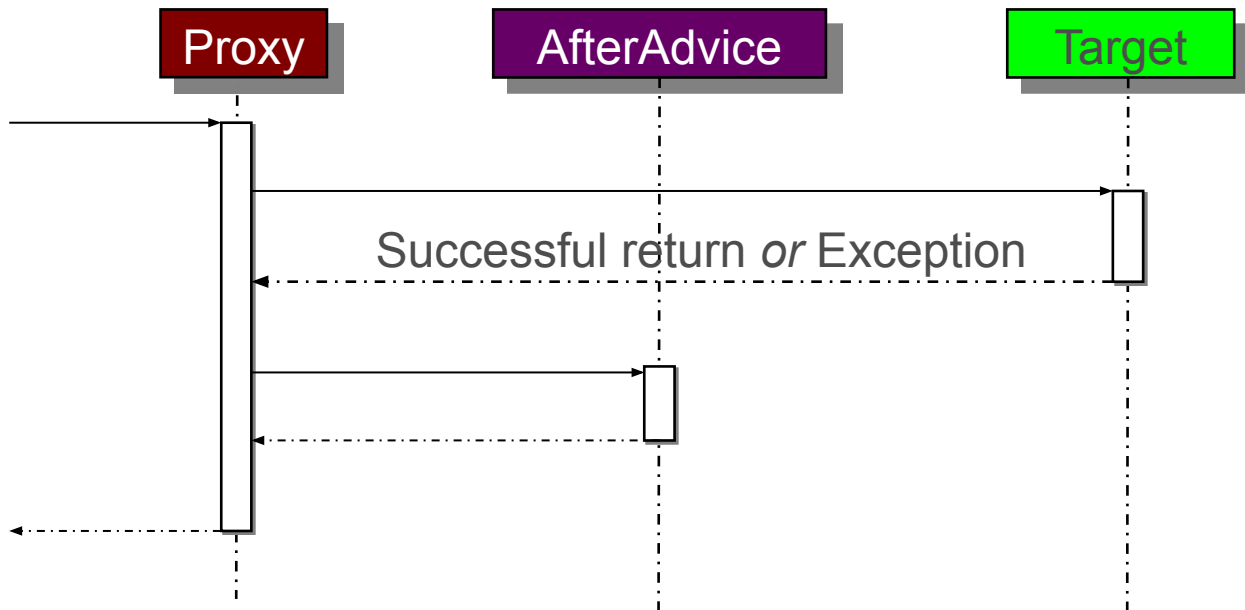
- The @AfterThrowing advice will not stop the exception from propagating
 - However it can throw a different type of exception

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



If you wish to stop the exception from propagating any further, you can use an @Around advice (see later)

Advice Types: After



After Advice Example

- Use *@After* annotation
 - Called regardless of whether an exception has been thrown by the target or not

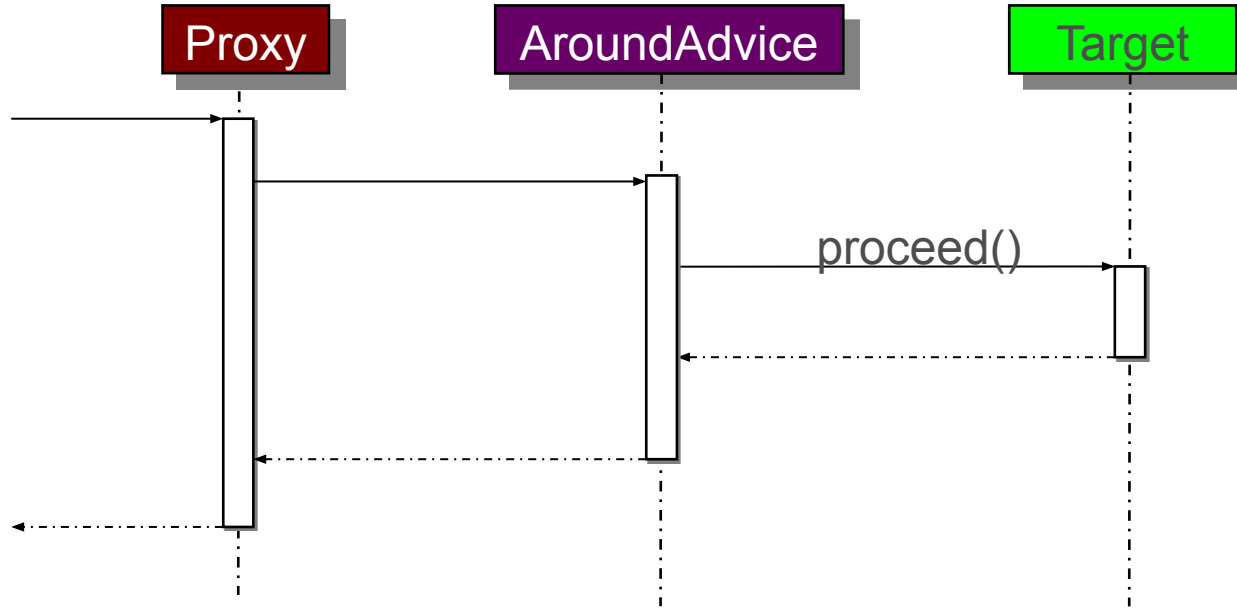
Track calls to all update methods

```
@Aspect
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

Advice Types: Around



Around Advice Example

- Use `@Around` annotation *and* a `ProceedingJoinPoint`
 - Inherits from `JoinPoint` and adds the `proceed()` method

```
@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

Limitations of Spring AOP

- Can only advise *non-private* methods
- Can only apply aspects to *Spring Beans*
- Limitations of weaving with proxies
 - When using proxies, suppose method a() calls method b() on the *same* class/interface
 - advice will *never* be executed for method b()

Summary

AOP

- Aspect Oriented Programming (AOP) *modularizes cross-cutting concerns*
- An aspect is a module (Java class) containing the cross-cutting behavior
 - Annotated with `@Aspect`
 - Behavior is implemented as an “*advice*” method
 - Pointcuts select *joinpoints* (methods) where *advice* applies
 - Five advice types
 - Before, AfterThrowing, AfterReturning, After and Around

A man with a beard and a woman are sitting at a desk in a computer lab, looking at a monitor. The man is pointing at the screen while the woman looks on. The background is slightly blurred, showing other lab equipment and people.

Lab: Introducing Aspect Oriented Programming

**Lab project:
22-aop**

**Anticipated Lab time:
35 Minutes**

Optional Topics: Named pointcuts, context selection, annotations in pointcuts

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Lab
- **Advanced Topics**
 - **Named Pointcuts**
 - Context Selecting Pointcuts
 - Working with Annotations



Named Pointcut Annotation

```
@Aspect
```

```
public class PropertyChangeTracker {
```

```
    private Logger logger = Logger.getLogger(getClass());
```

```
    @Before("serviceMethod() || repositoryMethod()")
```

```
    public void monitor() {
```

```
        logger.info("A business method has been accessed...");
```

```
    }
```

```
    @Pointcut("execution(* rewards.service..*Service.*(..))")
```

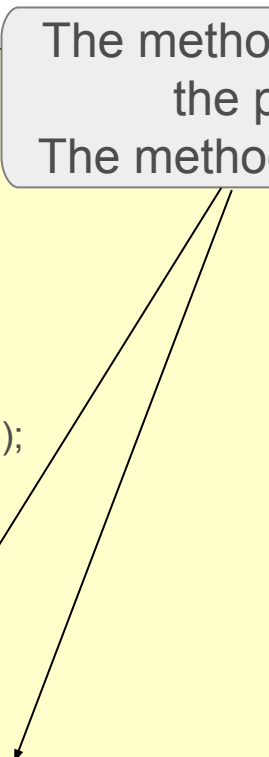
```
    public void serviceMethod() {}
```

```
    @Pointcut("execution(* rewards.repository..*Repository.*(..))")
```

```
    public void repositoryMethod() {}
```

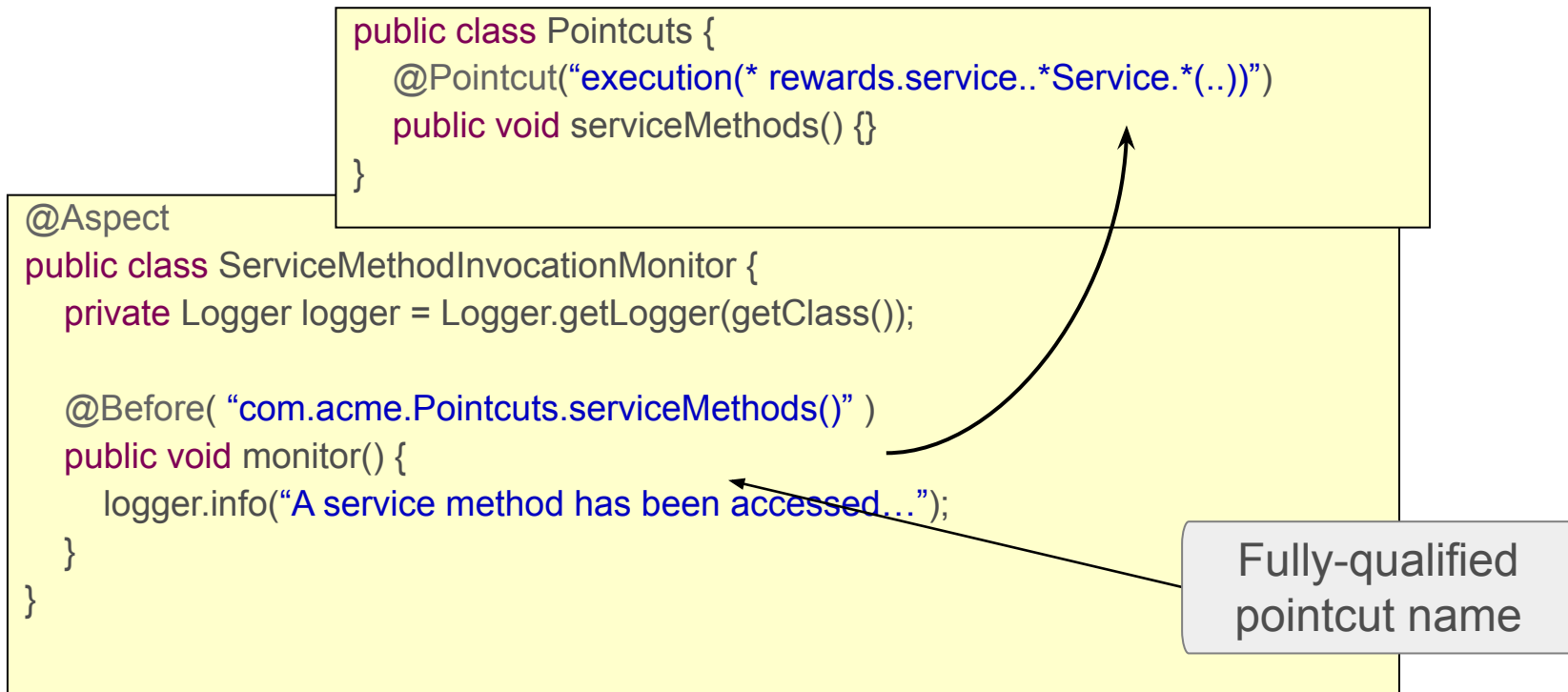
```
}
```

The method *name* becomes
the pointcut ID.
The method is *not* executed.



Named Pointcuts

- Expressions can be externalized



Named Pointcuts - Summary

- Can break one complicated expression into several sub-expressions
- Allow pointcut expression reusability
- Best practice: consider externalizing expressions into one dedicated class
 - When working with many pointcuts
 - When writing complicated expressions

Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Lab
- **Advanced Topics**
 - Named Pointcuts
 - **Context Selecting Pointcuts**
 - Working with Annotations



Context Selecting Pointcuts

- Pointcuts may also select useful join point context
 - The target object
 - Method arguments
 - Annotations associated with the method, target, or arguments
 - The currently executing object (proxy)
- Allows for simple POJO advice methods
 - Alternative to working with a JoinPoint object directly

Context Selecting Example

- Consider this basic requirement

Log a message every time Server is about to start

```
public interface Server {  
    public void start(Map input);  
    public void stop();  
}
```

In the advice, how do we access Server? Map?

Without Context Selection

- All needed info must be obtained from *JoinPoint* object
 - No type-safety guarantees
 - Write advice *defensively*

```
@Before("execution(void example.Server.start(java.util.Map))")
public void logServerStartup(JoinPoint jp) {
    // A 'safe' implementation would also check target type
    Server server = (Server) jp.getTarget();
    // Don't assume args[0] exists
    Object[] args= jp.getArgs();
    Map map = args.length > 0 ? (Map) args[0] : new HashMap();
    logger.info( server + " starting – params: " + map);
}
```

With Context Selection

- Best practice: use context selection
 - Method attributes are bound automatically
 - Types must match or advice skipped

```
@Before("execution(void example.Server.start(java.util.Map))  
    && target(server) && args(input)")  
public void logServerStartup(Server server, Map input) {  
    ...  
}
```

- target(server) selects the target of the execution (your object)
- this(server) would have selected the proxy

Context Selection - Named Pointcut

```
@Before("serverStartMethod(server, input)")
```

```
public void logServerStartup(Server server, Map input) {
```

```
    ...
```

```
}
```

'target' binds the
server starting up

'args' binds the
argument value

```
@Pointcut("execution(void example.Server.start(java.util.Map))  
           && target(server) && args(input)")
```

```
public void serverStartMethod (Server server, Map input) {}
```


Agenda

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Lab
- **Advanced Topics**
 - Named Pointcuts
 - Context Selecting Pointcuts
 - **Working with Annotations**



Pointcut Expression Examples using Annotations

- Can match annotations everywhere
 - annotated methods, methods with annotated arguments, returning annotated objects, on annotated classes
- `execution(@org..transaction.annotation.Transactional * *(..))`
 - Any method marked with the `@Transactional` annotation
- `execution((@example.Sensitive *) *(..))`
 - Any method that returns a type marked as `@Sensitive`



```
@Sensitive
public class MedicalRecord { ... }

public class MedicalService {
    public MedicalRecord lookup(...) { ... }
}
```

AOP and Annotations - Example

- Use of the *annotation()* designator

```
@Around("execution(* *(..)) && @annotation(txn)")
public Object execute(ProceedingJoinPoint jp, Transactional txn) {
    TransactionStatus tx;

    try {
        TransactionDefinition definition = new DefaultTransactionDefinition();
        definition.setTimeout(txn.timeout());
        definition.setReadOnly(txn.readOnly());
        ...
        tx = txnMgr.getTransaction(definition);
        return jp.proceed();
    }
    ... // commit or rollback
}
```

No need for @Transactional in *execution* expression – the *@annotation* matches it instead

AOP and Annotations – Named pointcuts

- Same example using a named-pointcut

```
@Around("transactionalMethod(txn)")  
public Object execute(ProceedingJoinPoint jp, Transactional txn) {  
    ...  
}
```

```
@Pointcut("execution(* *(..)) && @annotation(txn)")  
public void transactionalMethod(Transactional txn) {}
```


Advanced Topics Summary

- Topics covered were:
 - Named Pointcuts
 - Context-Selecting Pointcuts
 - Working with Annotations