ASPECT ORIENTED PROGRAMMING

SPRING - AOP

NEELAM AGARWAL

OUTLINE

- Introduction to AOP
- AOP terminologies
- Spring AOP architecture
- @AspectJ support
- Schema-based AOP

ASPECT ORIENTED PROGRAMMING

NEELAM AGARWAL

ASPECT ORIENTED PROGRAMMING

- *Aspect-Oriented Programming* (*AOP*) complements OOP
- Unit of modularity in OOP is *class*
- Unit of modularity in AOP is *aspect*
- To modularize concerns such as transaction management that would otherwise cut across multiple objects.
- Such concerns are called as ***crosscutting*** concerns

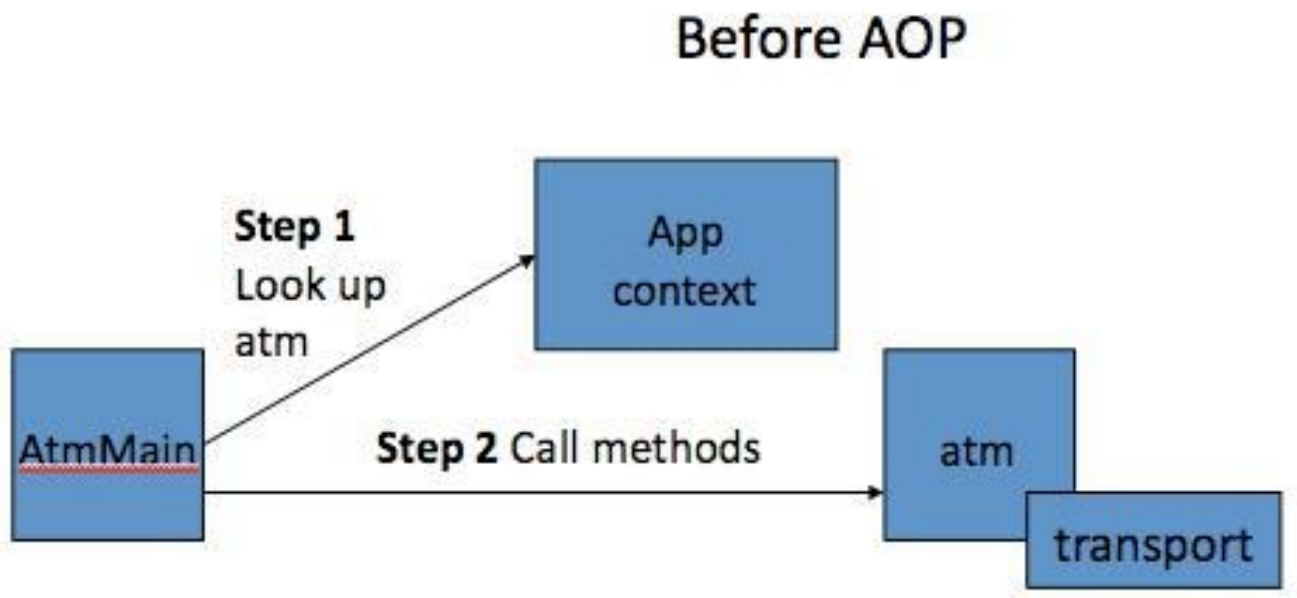
WHERE AOP IS USED?

- To provide declarative enterprise services, especially as a replacement for EJB declarative services.
  - Ex : Declarative transaction management
- To allow users to implement custom aspects, complementing their use of OOP with AOP

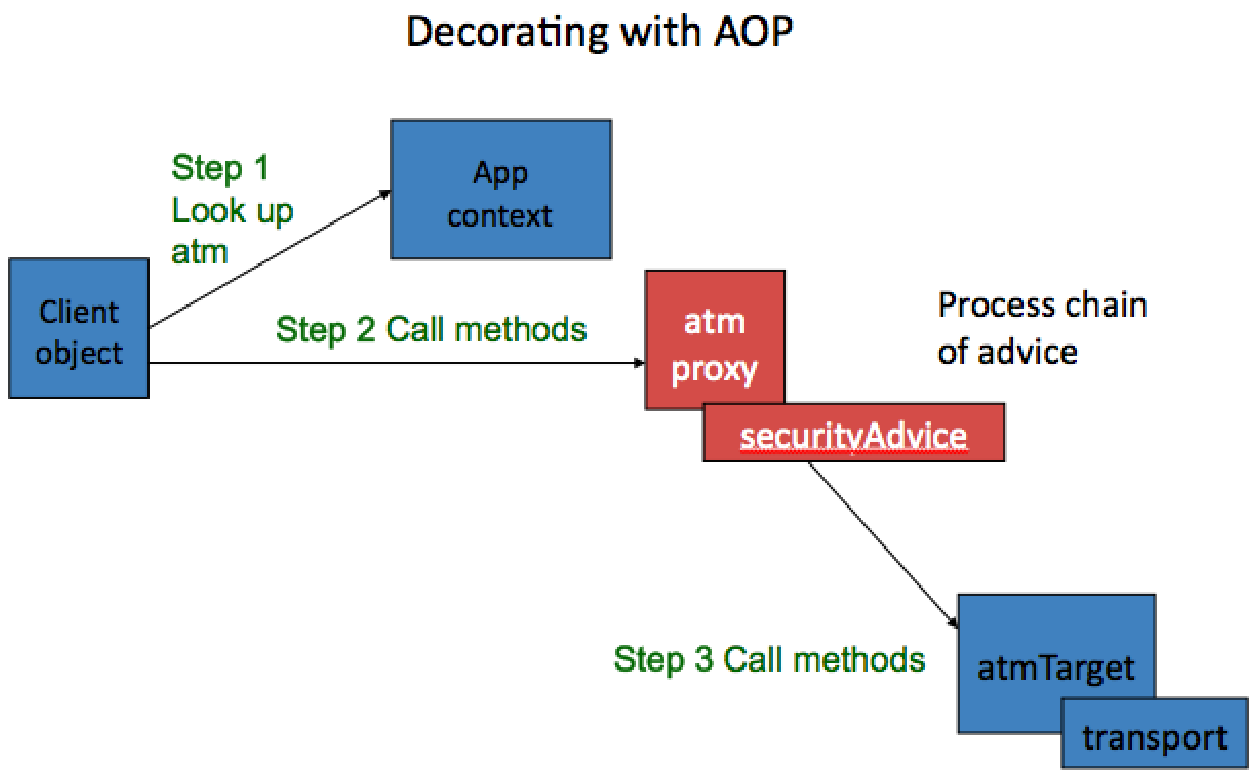
SEPARATION OF CONCERN

- Advantages
  - Focus on the concerns at one place
  - Easier to add and remove concerns with affecting the other parts of the code
  - Much easier to understand the concerns
  - Concerns can be implemented more efficiently as they are segregated from the main code

BEFORE AOP


Before AOP
Step 1 Look up atm
App context
AtmMain
Step 2 Call methods
atm
transport

AFTER AOP



Decorating with AOP
Step 1
Look up
atm
App
context
Client
object
Step 2 Call methods
atm
proxy
securityAdvice
Process chain
of advice
Step 3 Call methods
atmTarget
transport

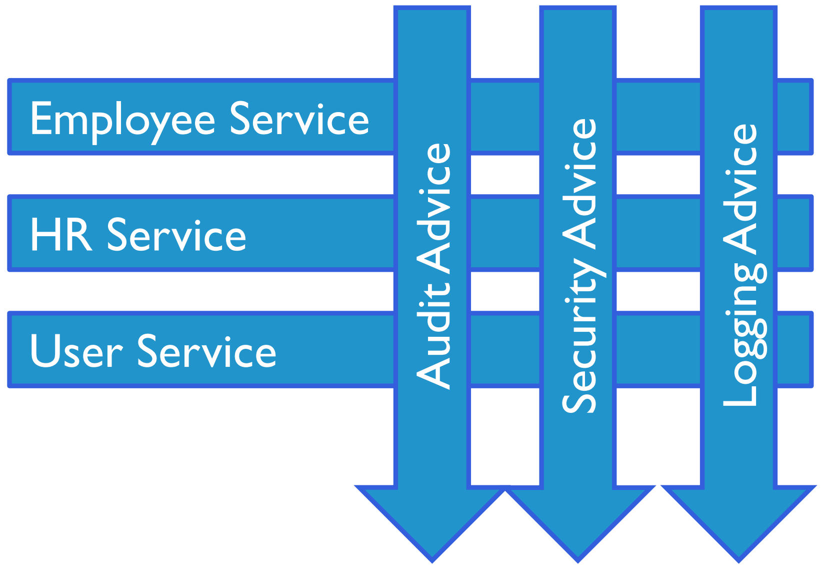
AOP TERMINOLOGIES

- Aspect
- Join point
- Advice
- Pointcut
- Target Object
- AOP Proxy
- Weaving

AOP TERMINOLOGIES

- Cross-cutting concern
  - Aspects of a program which affect (crosscut) other core concerns

  - Examples
    - Logging
    - Security
    - Auditing
    - Locking
    - Event handling
    - Transaction management



AOP TERMINOLOGIES

- Aspects
  - A modularization of a concern that cuts across multiple classes
  - It is collection of advice, pointcuts.

    Ex : Transaction management
  - 2 ways to implement
    - @AspectJ style
    - Schema-based approach

- Joinpoint
  - Point during the execution of a program, such as a method invocation or exception handling
  - In Spring AOP, a join point is always method execution

AOP TERMINOLOGIES

- Advice
  - Action taken by the AOP framework at a particular joinpoint
  - It contains the actual code that you want to execute.
  - AOP Frameworks model an advice as an interceptor, maintaining a chain of interceptors around the join point

- Point cut
  - A predicate that matches the join points
  - An expression that selects set of joinpoints specifying when an advice should fire
  - Ex: Execution of a method with a certain name

AOP TERMINOLOGIES

- Target Object
  - Object being advised by one or more aspects.
  - Also known as the *Advised* object
  - Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object
- AOP Proxy
  - Object created by the AOP framework in order to implement the aspect contracts
  - In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- Weaving
  - Linking aspects with other application types to create an advised object
  - Can be done at compile time, load time or runtime
  - Spring AOP performs weaving at runtime

AOP IMPLEMENTATION

- AspectJ – http://www.eclipse.org/aspectj
- Spring AOP -
- Jboss AOP – http://jboss.org/jbossaop

TYPES OF ADVICE

- Before Advice
  - Advice that executes before a method execution
  - Does not have the capability to prevent execution flow proceeding to the join point

- After returning Advice
  - Advice that executes after a method execution completes normally, without throwing an exception

- After throwing Advice
  - Advice to be executed if a method exits by throwing an exception

- After (finally) Advice
  - Advice to be executed regardless whether a method returns normally or throw an exception

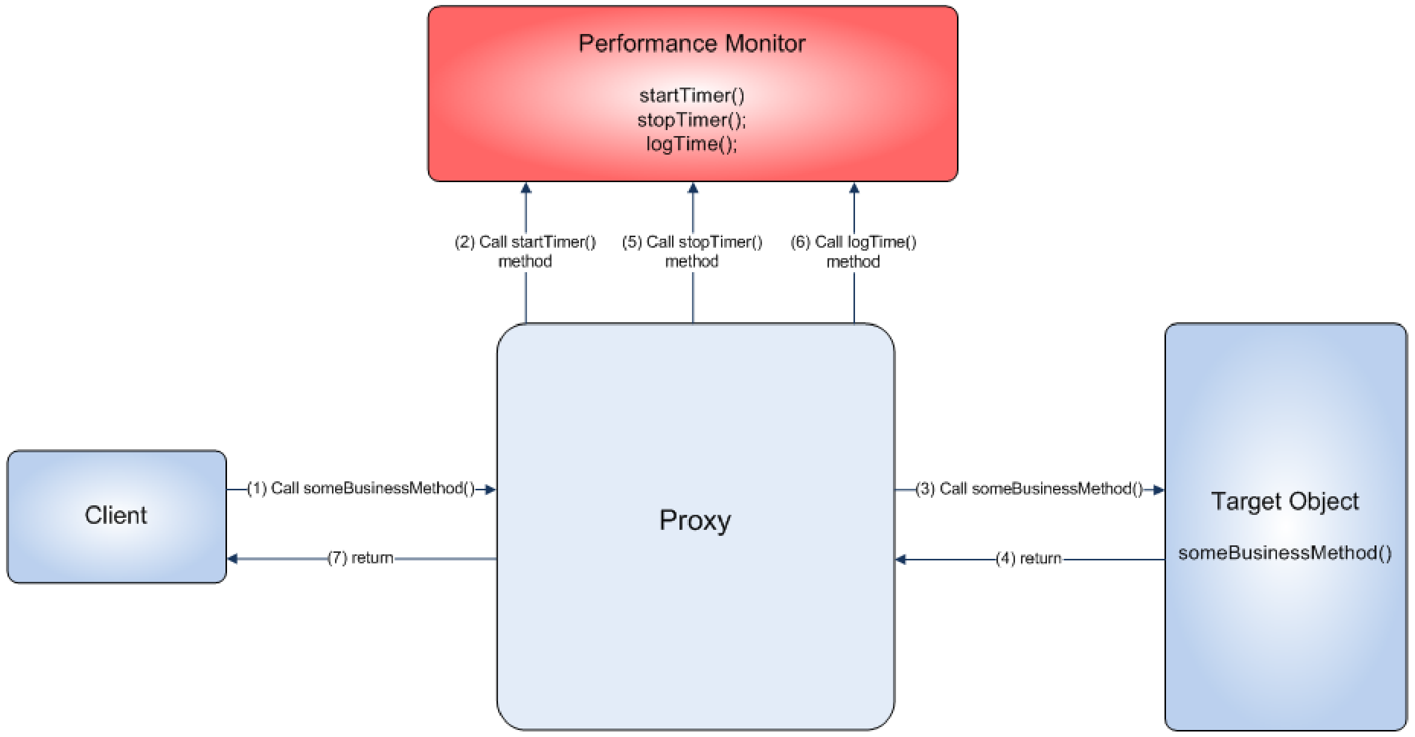
TYPES OF ADVICE (CONTD..)

- Around Advice
  - Advice that surrounds a method invocation
  - Can perform custom behavior before and after the method invocation
  - Can decide whether to execute the method (joinpoint) or to return its own value or throw an exception
  - Most powerful

SPRING AOP IMPLEMENTATION

NEELAM AGARWAL

SPRING AOP IMPLEMENTATION


Performance Monitor
startTimer()
stopTimer();
logTime();
(2) Call startTimer() method
(5) Call stopTimer() method
(6) Call logTime() method
Client
(1) Call someBusinessMethod()
(7) return
Proxy
(3) Call someBusinessMethod()
(4) return
Target Object
someBusinessMethod()

SPRING AOP CAPABILITIES

- Spring AOP is implemented in Pure Java.
- Spring AOP currently supports only method execution join points
- Spring aim is to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications
  - Aspects are configured using normal bean definition syntax
- Spring supports 2 ways to implement Aspects
  - Annotations- @AspectJ style
  - Schema-based approach

SPRING AOP CAPABILITIES

- Spring AOP is implemented in pure Java
- Spring AOP currently supports only method execution join points
- Spring supports 2 ways to implement Aspects
  - Annotation-@AspectJ style
  - Schema-based approach

AspectJ Support

@ASPECTJ SUPPORT

- @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations
- The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver
- To enable @AspectJ support, include the below in the Spring configuration

```
<aop:aspectj-autoproxy/>
            or
@EnableAspectJProxy
```

DECLARING AN ASPECT

- Any bean with *@Aspect annotation will be configured* with Spring AOP as an aspect

- In order to be auto-detected through auto-scanning, *Component annotation can be used along with @Aspect* annotation

```
@Component
@Aspect
public class NotVeryUsefulAspect {}
```

- Spring will automatically create proxies for any of your beans that are matched by your AspectJ aspects.

CREATING NAMED POINTCUTS

- A pointcut declaration has two parts
  - Signature - comprising a name and any parameters
  - Pointcut expression - that determines *exactly which method* executions we are interested in

- Ex: '*anyOldTransfer*' is the pointcut that will match the execution of any method named '*transfer*':

```
// the pointcut expression
@Pointcut("execution(* transfer(..))")
// the pointcut signature
private void anyOldTransfer() {}
```

Pointcut Designator

- This can be referred anywhere that you need a pointcut expression

SUPPORTED POINTCUTS DESIGNATORS

- Execution
  - For matching method execution join points
- Within
  - Limits matching to join points within certain types
- This
  - Limits matching to join points where the bean reference is an instance of given type
- Target
  - Limits matching to join points where the target object is an instance of the given type
- args
  - Limits matching to join points where the arguments are instances of the given types

SUPPORTED POINTCUTS DESIGNATORS

- @target
  - Where the class of the executing object has an annotation of the given type
- @args
  - where the runtime type of the actual arguments passed have annotations of the given type(s)
- @within
  - limits matching to join points within types that have the given annotation
- @annotation
  - where the subject of the join point has the given annotation

EXAMPLES

Pointcut Expression	Meaning
execution(public * *(..))	Execution of any public method
execution(* set*(..))	Execution of any method beginning with "set"
execution(* com.xyz.service. AccountService.*(..))	Execution of any method defined by the AccountService interface
execution(* com.xyz.service.*.*(..))	Execution of any method defined in the service package
execution(* com.xyz.service..*.*(..))	Execution of any method defined in the service package or a sub-package
Within(com.xyz.service.*)	Any join point with a service package
Within(com.xyz.service.*.*)	Any join point with a service package or subpackage
This(com.xyz.service.AccountService)	Any join point where the proxy implements the AccountService interface
Target(com.xyz.service.AccountService)	Any join point where the target object implements the AccountService interface
Args(java.io.Serializable)	Any join point which takes a single parameter, and where the argument passed at runtime is Serializable

EXAMPLES

Pointcut Expression	Meaning
@target(org.springframework.transaction.annotation.Transactional)	any join point where the target object has an @Transactional annotation:
@within(org.springframework.transaction.annotation.Transactional)	any join point where the declared type of the target object has an @Transactional annotation
@annotation(org.springframework.transaction.annotation.Transactional)	any join point where the executing method has an @Transactional annotation
@args(com.xyz.security.Classified)	any join point which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:
bean(tradeService)	any join point on a Spring bean named tradeService
bean(*Service)	any join point on Spring beans having names that match the wildcard expression *Service

DECLARING AN ADVICE

- Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut

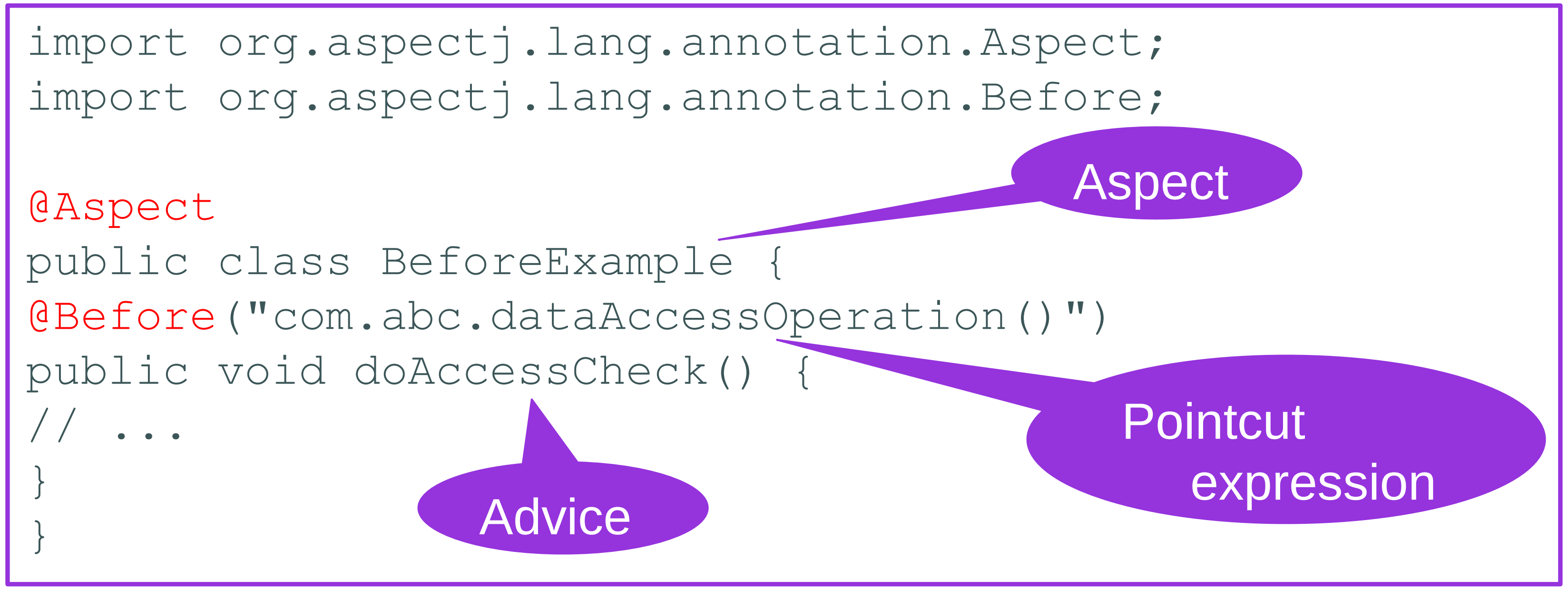
- The pointcut expression may be either a simple reference to a named pointcut, or a simple pointcut expression declared in place

DECLARING ADVICE

- Before Advice
  - Advice that executes before a method execution
- After Returning Advice
  - Advice that executes after a method execution completes normally
- After Throwing Advice
  - Advice to be executed if a method exits by throwing an exception
- After Advice
  - Advice to be executed regardless whether a method returns normally or throw an exception
- Around Advice
  - Advice that surrounds a method invocation
  - Can perform custom behavior before and after the method invocation
  - Can decide whether to execute the method (joinpoint) or to return its own value or throw an exception
  - Most powerful

BEFORE ADVICE

- Declared in an aspect using the @Before annotation



```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {
@Before("com.abc.dataAccessOperation()")
public void doAccessCheck() {
// ...
}
}
```

AFTER RETURNING ADVICE

- Use @AfterReturning annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {
@AfterReturning ("com.abc.dataAccessOperation()")
public void doLogging() {
// ...
} }
```

- To access the actual value returned inside the advice body

```
@AfterReturning(
pointcut="com.abc.dataAccessOperation()",
returning="retVal")
public void doAccessCheck(Object retVal) {
// ...
}
```

AFTER THROWING ADVICE

- Use @AfterThrowing annotation

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {
@AfterThrowing ("com.abc.dataAccessOperation()")
public void doRecoveryActions () {
// ...
} }
```

- To make the advice run only when exceptions of a given type are thrown

```
@AfterThrowing(
pointcut="com.abc.dataAccessOperation()",
throwing="ex")
public void doRecoveryActions(DataAccessException ex)   {
// ...
}
```

AROUND ADVICE

- Decides when, how and even if, the method actually gets to execute at all
- proceed causes the underlying method to execute

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;
@Aspect
public class AroundExample {
@Around("com.xyz.businessService()")
public Object doBasicProfiling(ProceedingJoinPoint pjp)
throws Throwable {
// start stopwatch
Object retVal = pjp.proceed();
// stop stopwatch
return retVal;
} }
```

SCHEMA-BASED AOP SUPPORT

NEELAM AGARWAL

SCHEMA-BASED AOP SUPPORT

- Uses XML-based format
- "aop" namespace tags are used to define aspects
- An <aop:config> element can contain pointcut, advisor, and aspect elements

DECLARING AN ASPECT

- An aspect is simply a regular Java object defined as a bean in the Spring application context
- Declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute

```
<aop:config>
<aop:aspect id="myAspect" ref="aBean">
...
</aop:aspect>
</aop:config>
<bean id="aBean" class="...">
...
</bean>
```

DECLARING A POINTCUT

- Top level Pointcut is declared inside <aop:config>
- Shared by several aspects and advisors

```
<aop:config>
<aop:pointcut id="businessService"
expression="com.xyz.businessService()"/>
</aop:config>
```

- Inline pointcut is specific to an aspect

```
<aop:config>
<aop:aspect id="myAspect" ref="aBean">
<aop:pointcut id="businessService" expression="execution(*
com.xyz.service.*.*(..))"/>
...
</aop:aspect>
</aop:config>
```

BEFORE ADVICE

- Declared in an aspect using the <aop:before> element

```
<aop:aspect id="beforeExample" ref="aBean">
<aop:before
pointcut-ref="dataAccessOperation"
method="doAccessCheck"/>
...
</aop:aspect>
```

Id of Top level Pointcut

- To define the pointcut inline, use pointcut attribute

```
<aop:aspect id="beforeExample" ref="aBean">
<aop:before
pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
method="doAccessCheck"/>
...
</aop:aspect>
```

Advice

AFTER RETURNING ADVICE

- Declared using <aop:after-returning > element

```
<aop:aspect id="afterReturningExample" ref="aBean">
<aop:after-returning
pointcut-ref="dataAccessOperation"
method="doAccessCheck"/>
...
</aop:aspect>
```

AFTER THROWING ADVICE

- Declared using <aop:after-throwing > element

```
<aop:aspect id="afterThrowingExample" ref="aBean">
<aop:after-throwing
pointcut-ref="dataAccessOperation"
method="doRecoveryActions"/>
...
```

AFTER (FINALLY) ADVICE

- Declared using <aop:after > element

```
<aop:aspect id="afterFinallyExample" ref="aBean">
<aop:after
pointcut-ref="dataAccessOperation"
method="doReleaseLock"/>
...
</aop:aspect>
```

AROUND ADVICE

- Use aop:around element

```
<aop:aspect id="aroundExample" ref="aBean">
<aop:around pointcut-ref="businessService"
method="doBasicProfiling"/>
...
</aop:aspect>
```

- The first parameter of the advice method is of type ProceedingJoinPoint and a call to proceed() on it causes the underlying method to execute

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws
Throwable {
// start stopwatch
Object retVal = pjp.proceed();
// stop stopwatch
return retVal;
}
```

UNDERSTANDING AOP PROXIES

- The core architecture of spring is built around proxies
- Goal of a proxy - Intercept method invocations
- When you create an advised instance of a class you must first create a proxy of an instance of that class.
- The proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call
- Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object

PROXYING MECHANISMS

CGLIB Proxy	**JDK Dynamic Proxy**
• This can generate proxies for classes on the fly • Used by default if a business object does not implement an interface. • final methods cannot be advised	• This can generate proxies only for interfaces not classes • This cannot be used when you are working with third party classes or legacy code

CGLIB OR JDK DYNAMIC PROXY?

- If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used

- To force the use of CGLIB proxies, set proxy-target-class attribute of <aop:config> to true

```
<aop:config proxy-target-class="true">
<!-- other beans defined here... -->
</aop:config>
```

- But prefer JDK dynamic proxies whenever you have a choice!!