Aspect Oriented Programming

2013-2014

Course 8

Course 8 Contents

- @AspectJ overview
- Spring AOP

@AspectJ Crosscutting Elements

Feature	Mapped element
Aspects	Class with @Aspect annotation
Pointcuts	Method with @Pointcut annotation
Advice	Method with @Before, @After, @AfterReturning, @AfterThrowing, or @Around annotation
Declaring parents	Field with @DeclareMixin and @DeclareParents annotation
Declaring errors and warnings	Field with @DeclareError and @DeclareWarning annotation
Introducing data and methods	Not supported
Exception handling	Not supported
Privileged aspects	Not supported

Mapping Aspects

@ AspectJ syntax	AspectJ syntax

`@Aspect`
`public abstract class Monitoring`
`{`
`...`
`}`
	`public abstract aspect Monitoring`
	`{`
	`...`
	`}`

`@Aspect`
`public class BankingMonitoring`
`  extends Monitoring {`
`...`
`}`
	`public aspect BankingMonitoring`
	`  extends Monitoring {`
	`...`
	`}`

Remark: Only classes can be marked with the @Aspect annotation (not other types such as interfaces and enums).

Mapping Aspects

- There are a few restrictions on the class declaration to match the aspect semantics:
  - Include a default public constructor if the class defines a constructor.
  - Not extend a concrete class carrying the @Aspect annotation. It matches the restriction that an aspect cannot extend another concrete aspect.
  - Not declare a generic parameter unless it is an abstract aspect. It matches the restriction that only abstract aspects are allowed to declare generic type parameters.
  - Be marked static if it is a nested class. It matches the restriction that an inner aspect must be static (not bound to an instance of the enclosing type).
- The @Aspect annotation has an optional property that you can use to specify aspect association.

Declaring aspect precedence

- The traditional syntax offers the **`declare precedence`** construct to control precedence between aspects.
- @AspectJ syntax offers the **`@DeclarePrecedence`** annotation :

```
@Aspect
@DeclarePrecedence("ajia.HomeSecurityAspect,
   ajia.SaveEnergyAspect")
public class HomeSystemCoordinationAspect {
}
```

- The @DeclarePrecedence annotation must be attached only to an aspect (a class with the @Aspect annotation).
- You set the annotation's value attribute to aspect type patterns separated by a comma, following the same rules as the traditional syntax.

Mapping pointcuts

- The @AspectJ syntax uses a method with a @Pointcut annotation to represent a pointcut.
- The value parameter of the annotation represents the pointcut expression.
- All other pointcut characteristics such as the name, access specification, abstractness, and parameters match with the corresponding characteristics of the method representing it.
- The pointcut expression used is the same as in the traditional syntax except for two differences:
  - the use of fully qualified type names
  - a special treatment of the if() pointcut.

Mapping Abstract Pointcuts

@ AspectJ syntax	AspectJ syntax
`@Pointcut` `public abstract void readOperation();`	`public abstract pointcut` `readOperation();`

`@Pointcut`
`public abstract void`
`accountOperation(Account account,`
`float amount);`
	`public abstract pointcut`
	`accountOperation(Account account,`
	`float amount);`

Remark: The method-access specification can be public, package (default), or protected, but not private.

Mapping concrete pointcuts

- A concrete pointcut uses a pointcut expression to specify a join point selection criterion and collect join point context.
- The @AspectJ syntax maps concrete pointcuts to a concrete method with the @Pointcut annotation, which specifies the pointcut expression.
- The method body for a concrete pointcut is empty, because the method is a placeholder without any significance for the code inside it (except if())

```
@Pointcut("<pointcut-definition>")
[access specifier] void <pointcut-name>([args]) {}
```

Eg. //@AspectJ

```
@Pointcut("execution(public void set*(*))")
public void setter() {}
```

//AspectJ

```
public pointcut setter() : execution(public void set*(*));
```

Requirements For Pointcut Expressions

//@AspectJ

```
@Pointcut("call(* java.sql.Connection.*(..))")
public void connectionOperation() {}
```

//AspectJ

```
public pointcut connectionOperation()
: call(* java.sql.Connection.*(..))
```

//or AspectJ

```
import java.sql.*;
...
public pointcut connectionOperation ()
: call(* Connection.*(..))
```

Mapping dynamic crosscutting constructs

- An advice is represented with a method.
- The method representing an advice carries a @Before, @After, @AfterReturning, @AfterThrowing, or @Around annotation to denote the kind of advice the method is implementing.
- The value attribute of the annotation denotes the associated pointcut (named or anonymous), whereas the method body denotes the advice to be executed.
- Because a method has a name, that name is also the advice's name.
- In the traditional syntax, advice does not have an inherent name (it may be annotated with @AdviceName to assign it a name).

Mapping dynamic crosscutting constructs

- The methods that stand in for advice:
  - Should not return a value (they must return `void`), except for the around advice. It matches the traditional syntax, where advice may not return a value unless it is an around advice.
  - May declare that it throws an exception similar to the traditional syntax.
  - Should not be declared static. It matches the traditional syntax, where advice behaves like an instance method, because it has access to the `this` variable that points to the aspect instance.
- Although all advice constructs follow common ideas, each construct has a few particularities.

The before advice

- The before advice is created using a method with the @Before annotation.

```
@Before("<pointcut>")
public void <advice-name>([arguments]) {
   ... advice body
}
```

- The value of the @Before annotation specifies the pointcut associated with the advice.
- The pointcut may be named (referring to a pointcut defined in the same or a different aspect) or may be a pointcut expression.
- The method used as advice must be public and must return void.
- The optional arguments to the method represent the join point context.

The after advice

- In AspectJ, the after advice has three variations: after (finally), after returning, and after throwing.
- The after (finally) advice, which executes regardless of how the join point execution completes, is similar to before advice from the code point of view. The only difference is the annotation used: @After.

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) &&
   target(connection)")
  public void connectionOperation(Connection connection) {}

  @After("connectionOperation(connection)")
  public void monitorUse(Connection connection) {
    System.out.println("Just used " + connection);
  }
}
```

The after advice

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) && target(connection)")
  public void connectionOperation(Connection connection) {}

  @AfterReturning("connectionOperation(connection)")
   public void monitorSuccessfulUse(Connection connection) {
     System.out.println("Just used " + connection + " successfully");
   }

   @AfterThrowing("connectionOperation(connection)")
    public void monitorFailedUse(Connection connection) {
       System.out.println("Just used " + connection+ " but met with a
   failure");
   }
}
```

Collecting The Return Value And Thrown Exception

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) && target(connection)")
  public void connectionOperation(Connection connection) {}

  @AfterReturning(value="connectionOperation(connection)",
   returning="ret")
  public void monitorSuccessfulUse(Connection connection, Object ret) {
    //...
  }

  @AfterThrowing(value="connectionOperation(connection)",throwing="ex")
  public void monitorFailedUse(Connection connection, Exception ex) {
    //...
 }
}
```

The around advice

- An around advice in @AspectJ is represented by a method with an @Around annotation.
- The method must be public and may return a value.
- The rules governing the return value are the same as the rules in the traditional syntax.
- The return value must be compatible with all matching join points. The method may also return Object, where the matching join points may return any type and the weaver takes care of necessary unboxing and casting.

Proceeding With The Original Join Point Execution

- In the traditional syntax, if you want to execute the original join point you use the special keyword **`proceed()`** in around advice.
- No such special keyword exists in @AspectJ styled around advice.
- If you need to proceed with the original join point, you must declare the method to take a parameter of the **`ProceedingJoinPoint`** type (which extends **`JoinPoint`**).
- The **`ProceedingJoinPoint`** interface provides two methods: **`proceed()`** and **`proceed(Object[])`**.
- The no-argument **`proceed()`** proceeds with the original join point with unaltered join point context (the execution object, method arguments, and so on).

Proceeding With The Original Join Point Execution

```
package ajia.monitoring;
import ...
@Aspect
public abstract class Monitoring {
  @Pointcut
  public abstract void monitored();

  @Around("monitored()")
  public Object measureTime(ProceedingJoinPoint pjp) throws
   Throwable{
    long startTime = System.nanoTime();
    Object ret = pjp.proceed();
    long endTime = System.nanoTime();
    System.out.println("Method "+
   pjp.getSignature().toShortString()+ " took " + (endTime-
   startTime));
    return ret;
  }
}
```

Mapping weave-time declarations

- You declare weave-time errors and warnings in @AspectJ by declaring a static final member of the String type annotated with a @DeclareError or @DeclareWarning annotation.
- The value of the string is the message emitted by the weaver upon detecting the occurrence of a matching join point; the annotation attribute specifies the pointcut for which errors and warning are to be emitted.

```
@DeclareError("callToUnsafeCode()")
static final String unsafeCodeUsageError= "This third-
   party...";

@DeclareWarning("callToBlockingOperations()")
static final String blockingCallFromAWTWarning
= "Please ensure you are not calling this from the AWT thread";
```

Mapping declare parents

- The @AspectJ syntax offers limited but useful support for static-crosscutting constructs aimed at type modification.
- The support for declaring parents comes through the @DeclareMixin annotation.
- This annotation mixes an interface into a matching set of types and delegates the implementation to a specified object.
- To mix in a set of types, you annotate a factory method with this annotation and specify a type-pattern to select types being mixed-in as its value attribute.
- The return type of the method (which must be an interface) is mixed in with the matched types. The AspectJ weaver uses the object returned by the method to delegate the implementation for the mixed-in interface.
- @AspectJ syntax also supports the @DeclareParents annotation.
- The @DeclareParents cannot implement the exact equivalent of the declare parents construct and may mislead developers into believing they are equivalent.
- @DeclareMixin is the preferred approach.

Declaring Parents For Marker Interfaces

- Declaring parents for marker interfaces (interfaces without any methods) requires that the factory method return a null (you may return any other object, but since there is nothing to delegate to, the object will remain unused).
- Eg. It declares all the types in the ajia.banking.domain package to implement java.io.Serializable:

```
@DeclareMixin("ajia.banking.domain.*")
public Serializable serializableMixin() {
   return null;
}
```

- The declaration is equivalent to

```
declare parents: ajia.banking.domain.* implements Serializable;
```

Declaring Parents For Non-marker Interfaces

- For a non marker interface, either the classes selected by the type pattern must already implement the methods for the interface or the factory method must return an object to delegate the implementations for the interface methods.

```
public interface Identifiable {
  public Long getId();
  public void setId(Long id);
}
```

- All classes in `ajia.banking.domain.*` must already implement getId and setId.

```
@Aspect
public class AccountTracking {
  @DeclareMixin("ajia.banking.domain.*")  //
   public Identifiable identifiableMixin() {
    return null;
  }
   ... advice
}
```

Requiring classes to already include an implementation is not useful in most situations.

Declaring Parents For Non-marker Interfaces

- The method annotated with @DeclareMixin annotation must return an object that is used as the delegate.

```
@Aspect
public class AccountTracking {
  @DeclareMixin("ajia.banking.domain.*")
   public Identifiable identifiableMixin() {
     return new IdentifiableDefaultImpl();
   }
}
```

- The AspectJ weaver uses the return object to delegate implementation of the interface to the types specified.
- The annotated factory method is public and may be declared static.
- If it is an instance method, it can use aspect instance members while creating the delegate object.

Spring Framework

- The Spring Framework is based on three core ideas: dependency injection, enterprise services abstraction, and aspect-oriented programming.
- Dependency injection (DI) is at the heart of Spring. It allows components to be wired in a declarative manner.
- The enterprise services abstraction encourages isolating stable application logic from volatile infrastructure code.
- AOP lets you separate the implementation of crosscutting concerns from business concerns.

Spring AOP fundamentals

- Various AOP extensions can implement the general AOP concepts in a variety of ways. Although their styles and capabilities differ, all the systems aim at the same problem of modularizing crosscutting concerns and share the core concepts.
- Spring provides its own way of implementing AOP: it uses proxies to avoid the need for explicit weaving (build-time or load-time).
- As AspectJ works with any Java application, you can still use AspectJ byte-code weaving.
- Spring also provides a few options to simplify AspectJ usage.

Spring AOP fundamentals

- The AOP programming model from the beginning used to be complex due to the exposure of low-level constructs directly to developers.
- Most aspects using that model were a part of Spring itself or written by advanced Spring developers.
- Starting with version 2.0, Spring simplifies the programming model through AspectJ integration that still works within the proxy-based framework.
- You can write aspects using:
  - A subset of the @AspectJ syntax.
  - Plain Java (without annotations), with AOP constructs expressed in XML (the schema-style AOP). This alternative is useful if you cannot use Java 5 or above (annotations).

Spring AOP fundamentals

- Spring also simplifies the use of AspectJ that relies on byte-code modification techniques. It supports
  - Configuring aspect instances as Spring beans.
  - Simplifying load-time weaving (LTW) without requiring modifications to the launch script (to add the weaving agent) for a few web application servers.

Example

```
public class ParticipantJdbcDAO extends SimpleJdbcDaoSupport
   implements ParticipantRepository {
   public void save(Participant p) { ...}
   public List<Participant> getByPoints() { ...}
   //...
}
@Aspect
public class LoggingContestTracingAspect {
   @Pointcut("execution(public * contest.repository.*.*(..))&&
   !within(LoggingContestTracingAspect)")
   public void tracing(){}
   private Logger logger=Logger.getLogger("contest");
   @Before("tracing()")
   public void trace(JoinPoint jp){
        logger.info("Entering:”+jp.getSignature());
    }
   @After("tracing()")
   public void afterTrace(JoinPoint jp){
        logger.info("Exiting: "+jp.getSignature());
    }
}
```

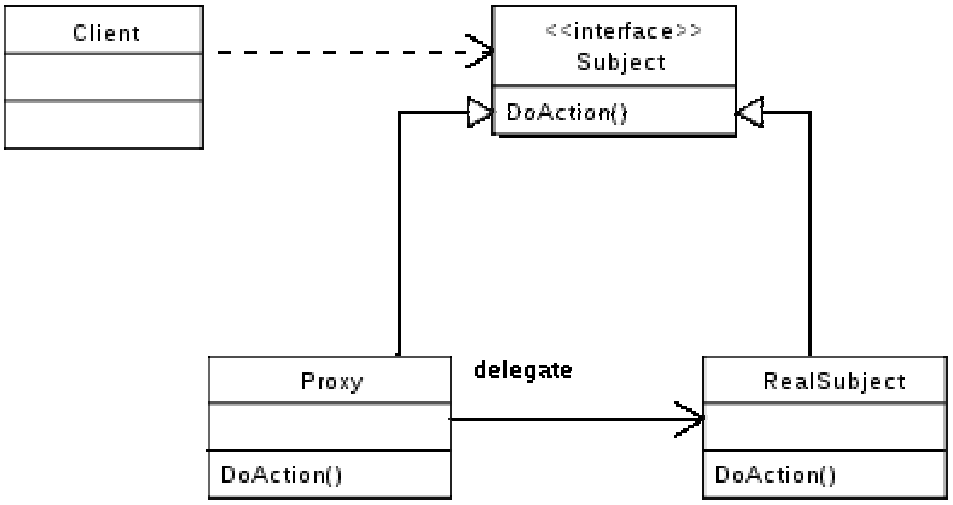
Example

```
//spring-contest.xml configuration file
<beans ....>
   <bean id="partRepo"
        class="contest.repository.dao.ParticipantJdbcDAO">
        <property name="dataSource"    ref="dataSource"/>
   </bean>
   <bean id="traceAspect"
        class="contest.aspects.LoggingContestTracingAspect"/>
   <aop:aspectj-autoproxy/>
</beans>
```

- In the configuration, you declare a bean corresponding to the `LoggingContestTracingAspect` type.
- To instruct Spring to use an advice declared in such a bean, you use the `<aop:aspectj-autoproxy/>`

Spring AOP

- Spring AOP is a proxy-based AOP system that modularizes crosscutting concerns.
- Based on the configuration instructions, Spring automatically creates a proxy for each bean that matches the criteria specified in pointcuts.
- The proxy intercepts calls to the original object.
- Spring AOP implementation uses dynamic proxies.



Introduction to dynamic proxies

- Starting with JDK 1.3, Java offers a mechanism to create proxies for interfaces.
- Given a set of interfaces, you can use the `Proxy` class to create a proxy. The creation method takes an invocation handler of the `InvocationHandler` type (which defines a single `invoke()` method).
- The dynamically created proxy dispatches each method to the invocation handler.
- You can start modularizing crosscutting functionality directly with proxies by writing an `InvocationHandler` whose `invoke()` method implements the crosscutting logic.
- JDK dynamic proxies work only with interfaces. You can use the Code Generation Library (CGLIB) and other byte-code engineering libraries to create proxies based on classes.

Example- dynamic proxies

- Tracing with dynamic proxies:

```
import ...
public class TracingHandler implements InvocationHandler {
   private Object target;
   public TracingHandler(Object target) {
        this.target = target;
   }
   public Object invoke(Object proxy, Method method, Object[]
            args) throws Throwable {  //similar to around advice
        System.out.println("Entering " + method);
        return method.invoke(target, args); //similar to proceed
   }
}
```

Example- dynamic proxies

- Tracing with dynamic proxies:

```
import ...
public class Main {
public static void main(String[] args) {
   ParticipantRepository participantRepository = new
   ParticipantRepositoryMock();
   ParticipantRepository participantRepositoryProxy =
   (ParticipantRepository)
Proxy.newProxyInstance(ParticipantRepository.class.getClassLoader(),
        new Class[]{ParticipantRepository.class},
        new TracingHandler(participantRepository));
   System.out.println(participantRepositoryProxy.findById("123"));
   System.out.println(participantRepositoryProxy.getByPoints());

   }
}
```

Example- dynamic proxies

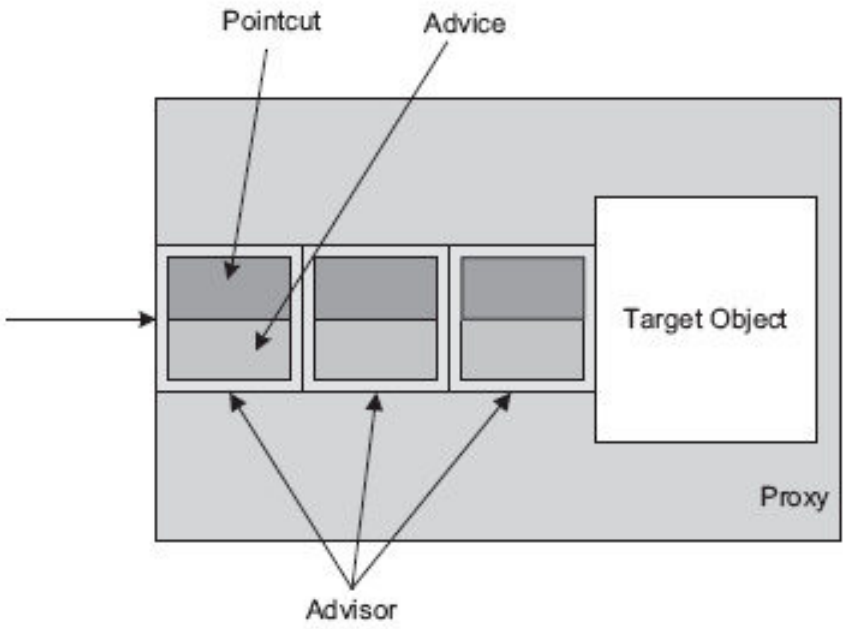
- Programming using proxies is too low level for typical needs of enterprise applications.
- Problems:
  - *Lack of a pointcut language* —There is no pointcut language with plain proxy usage. This forces the invocation handlers to perform two roles: selecting the join point, and implementing crosscutting logic. If you wanted to trace only a few specific methods, the logic to select those methods would have to reside in the `invoke()` method.
  - *Explicit creation of proxies* —You must control the creation of each object that needs crosscutting functionality, in order to create a proxy and wrap the original object in it.
  - *Weakly typed access to join point context* —Inside the `invoke()` method, join point context (the target object and method arguments) is available only as the `Object` type. This forces potentially erroneous typecasts if you need to invoke any methods on them.

Proxy-based AOP with Spring

- Spring AOP uses proxies as the underlying implementation while simplifying the associated programming model.
- Spring's programming model overcomes issues with the use of dynamic proxies:
  - *Pointcut language* —Spring supports multiple possibilities for selecting join points.
  - *Automatic proxy creation* —The Spring container already controls the creation of beans for the purpose of dependency injection (DI). Spring can extend the creation logic to wrap those beans in an automatically created proxy. The result is reduced programming burden for developers.
  - *Strongly typed access to join point context* —Through @AspectJ and schema-style aspect integration, due to use of context-collecting pointcuts, advice can access join point context in a strongly typed manner.

Spring AOP

- Spring uses proxies configured with pointcuts and advice as a mechanism to implement AOP while staying with the overall mechanism dictated by proxies.
- The proxy object is a wrapper around the target object and advisors. Each advisor contains a pointcut and an advice. The pointcut decides whether the invoked method is to be advised, and the advice contains the crosscutting functionality.



Spring AOP - Roles

- A proxy implements the same interfaces as those implemented by the target object.
- The proxy may extend the class of the target object.
- The external code may use the proxy as a replacement for the target object.
- Each proxy contains advisors. Advisor is an internal Spring concept that developers do not have to know about. Each advisor, in turn, contains two parts: pointcut and advice.
  - The pointcut selects the join points, to decide if the advice should execute.
  - The advice implements the logic needed for the crosscutting functionality.

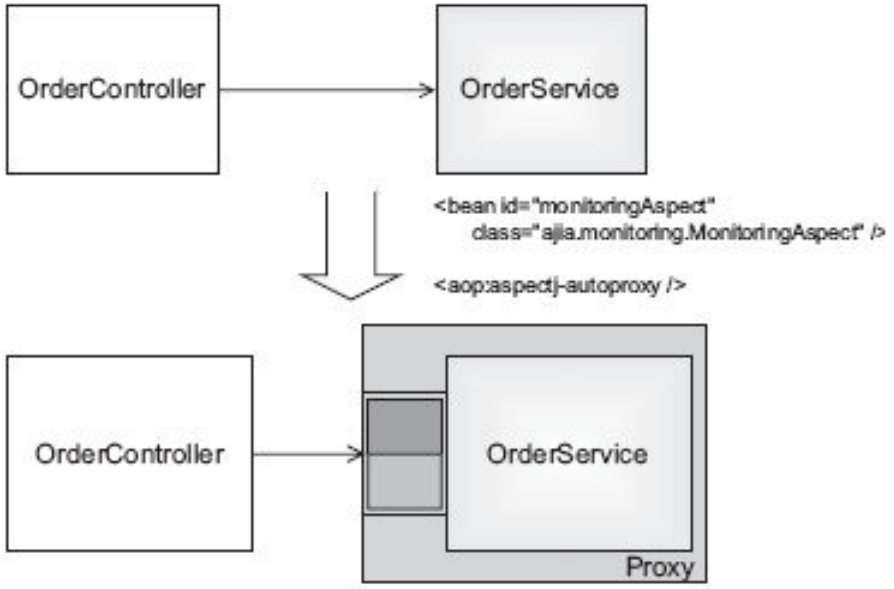
Spring AOP

- By default, Spring uses JDK dynamic proxies. It works only when the target class implements one or more interfaces.
- If you explicitly want to avoid JDK proxies, Spring offers CGLIB-based proxies, where it dynamically creates a subclass of the target class.
- Because Java prevents overriding final methods, the dynamically created proxy can not advise final methods in the target class. Spring produces a warning if it detects such a situation.
- If a final method is invoked, it can lead to inconsistency between the target and proxy object state. If you see such a warning, you should consider avoiding CGLIB proxies.
- When using CGLIB, your class must have a default constructor (public, no-args).

Proxy-based AOP in DI framework

- Spring AOP fits with the overall DI mechanism of the Spring framework. You declare configuration only for the primary injection.
- E.g., the injection of the OrderService bean into the OrderController bean.

Spring's proxy-based AOP creates proxies around any bean that needs to be advised and injects that bean instead of the original.



Proxy-based AOP in DI framework

- To implement a crosscutting concern, you declare a bean for an aspect and configure the application context to have Spring's autoproxy mechanism use that aspect:

```
<bean id="monitoringAspect"
   class="ajia.monitoring.MonitoringAspect"/>
<aop:aspectj-autoproxy/>
```

- The declaration `<aop:aspectj-autoproxy/>` is an instruction to Spring that it should do the following:
  - Examine each bean to check if it's an aspect (by checking for the @Aspect annotation associated with it). If so, check pointcuts and advices in it.
  - For each bean in the application context, check if the advice would apply to a method in that bean.

Proxy-based AOP in DI framework

- Cont.:
  - For each matching bean, automatically create a proxy (hence the name autoproxy) wrapping the original bean—the target object. The proxy implements the same interfaces or extends the same class as the target object.
  - Replace the original bean in the application context with its proxy. When Spring's DI performs injection, the proxied bean—and not the target bean—is injected.
- The DI mechanism makes proxy-based AOP transparent to the programmer.
- The result is a clean separation of core injection logic and crosscutting logic.

Limitations of Spring AOP

- Method execution-only join point.

  Spring AOP works only for method execution join points, because proxies can intercept only method executions.

  For many crosscutting concerns seen in enterprise applications, this may not be a problem. Even in implementations using AspectJ, method execution is by far the most commonly used join point.

- Bean-only crosscutting

  The proxy creation requires that the Spring container creates the objects that need to be advised. Because Spring controls the instantiation of beans, Spring AOP works naturally with beans.

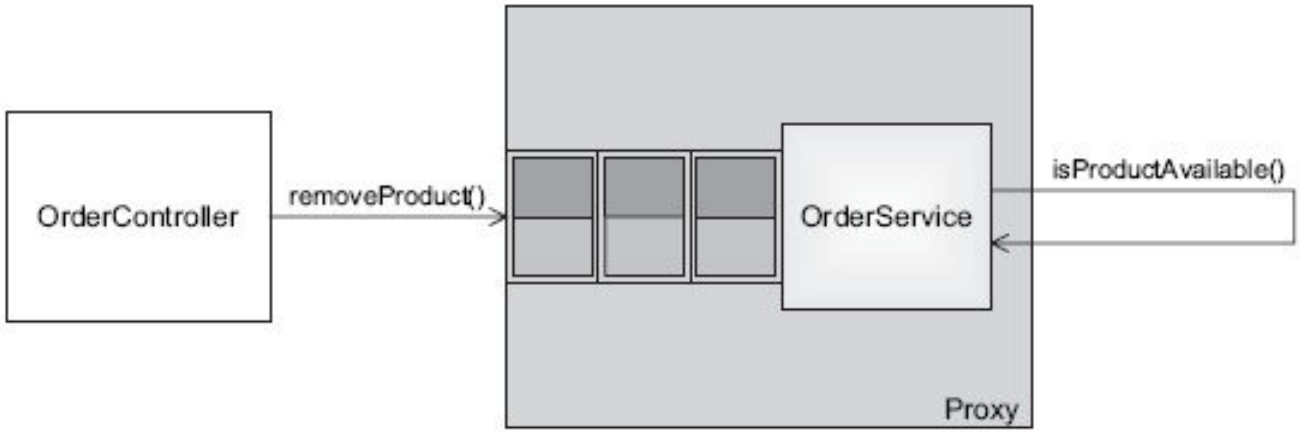
  Although you can apply Spring AOP programmatically to any object, as long as you can control its instantiation, this is not usually done in practice. The main reason is to limit dependency on the Spring Framework.

Limitations of Spring AOP

- External call–only interception

  With Spring's proxy-based AOP, the calls to the self (the this object) made from the target object are not advised. Unless the object on which a method is invoked is a proxy, the advisor does not apply.

  You can get around the self-call limitation by obtaining the proxy to the self object and invoking a method on it (it couples business logic with Spring Framework).



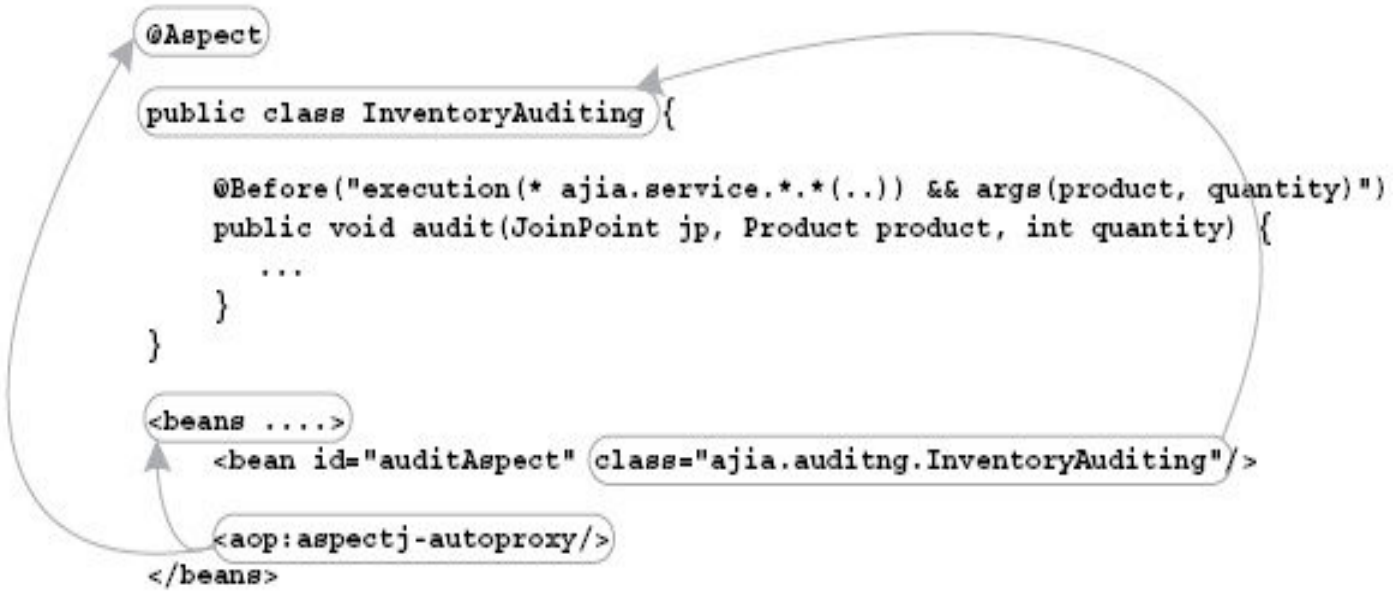
Spring @AspectJ support

- Spring offers integration with aspects expressed in the @AspectJ syntax.
- This facilitates easy migration to AspectJ, when needed, by switching to AspectJ weaving.
- Because Spring AOP uses the proxy-based implementation, Spring supports only a subset of the @AspectJ language.
- Spring's @AspectJ integration does not support pointcuts such as call() and handler().
- In addition, some of the static crosscutting constructs, such as compile-time errors and warnings, are not supported either.
- There is an additional pointcut to select join points in specific beans.

Spring @AspectJ support

- To integrate an aspect with Spring, you declare a bean corresponding to the @AspectJ aspect in the same way as any other bean.
- If needed, you can even inject dependencies into this bean and use them in the advice logic.
- Then, you instruct Spring through XML to use those beans to advise Spring beans.
- You can use any class written using the @AspectJ syntax.
- In the application context, you must instantiate a bean of that class.
- The `<aop:aspectj-autoproxy>` element indicates application of the aspects defined using the @AspectJ syntax.

The @AspectJ Integration



The @AspectJ Integration

- By default, Spring includes all beans with the @Aspect annotation for the proxy creation purpose.
- If you want to control which aspects should be automatically applied, you can specify the `<aop:include>` element with a list of bean names nested in `<aop:aspectj-autoproxy>`:

```
<aop:aspectj-autoproxy>
   <aop:include name="auditAspect"/>
</aop:aspectj-autoproxy>
```

- When there is such a declaration, Spring uses only the listed beans for autoproxying.

The @AspectJ Integration

- By default, Spring uses JDK dynamic proxies.
- But if a class being advised does not implement any interfaces, you can not use JDK dynamic proxies.
- In those situations, or if you wish to use CGLIB-based proxies, you can set the optional `proxy-target-class` attribute to true:

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

- This attribute lets you advise beans that do not implement any interfaces.
- It also lets you advise a class that implements interfaces, but you want to advise methods declared in the class that are not declared in any of the interfaces.

Dynamic crosscutting

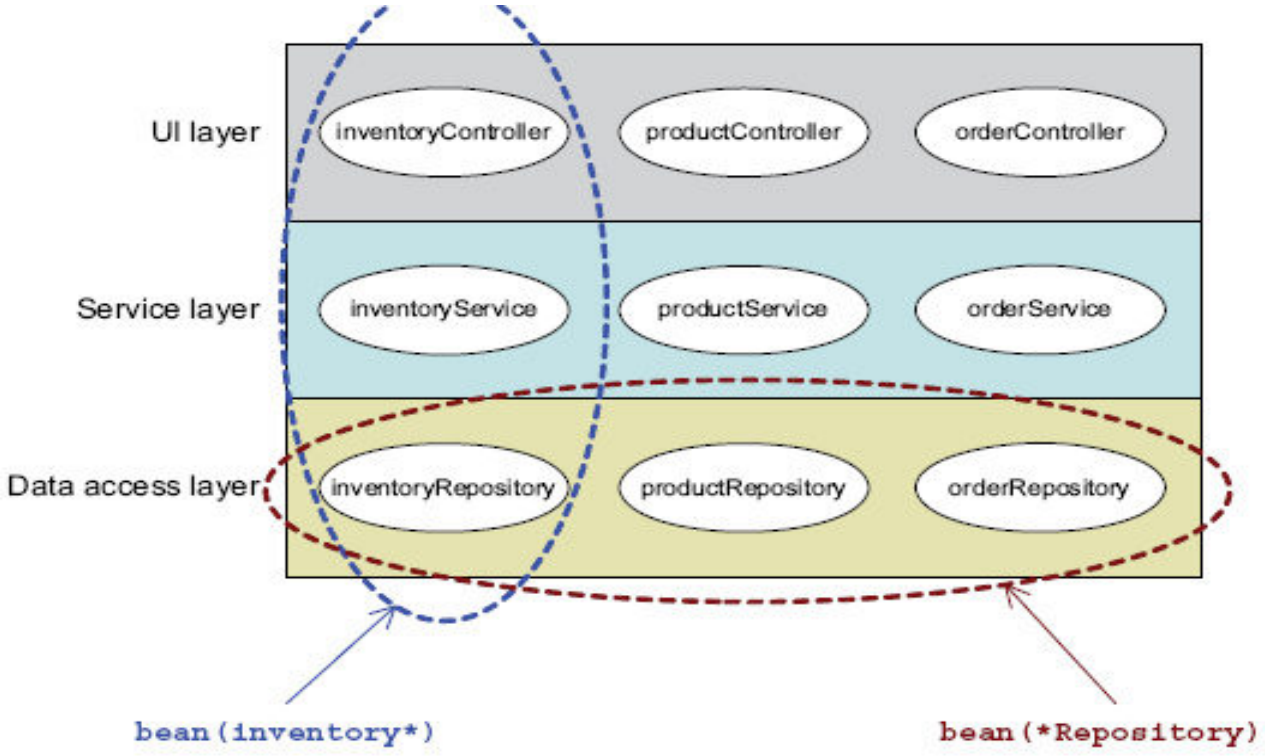
- Dynamic crosscutting comes in the form of advice that uses pointcuts.
- Spring uses the @AspectJ syntax and semantics to the extent possible given the constraint of a proxy-based implementation.
- Pointcuts you cannot use `call()`, `initialization()`, `preinitialization()`, `staticinitialization()`, `get()`, `set()`, `handler()`, `adviceexecution()`, `withincode()`, `cflow()`, `cflowbelow()`, `if()`, `@this()`, and `@withincode()` pointcuts.
- Available pointcuts:
  - `execution(Method-pattern)`, `this(TypeOrIdentifier)`, `target(TypeOrIdentifier)`, `args(TypeOrIdentifier1, TypeOrIdenifier2, ..)`, `within(Type-pattern)`, `@target(AnnotTypeOrIdentifier)`, `@args(AnnotTypeOrIdentifier1, AnnotTypeOrIdentifier2, ..)`, `@within(AnnotTypeOrIdentifier)`, `@annotation(AnnotTypeOrIdentifier)`
- Spring also offers an additional pointcut to select Spring beans.

Selecting Spring Beans

- Starting with version 2.5, Spring offers a new pointcut designator: `bean(<name-pattern>).`
- The name-pattern follows the AspectJ matching rules for a name pattern.
- The `*` is the only wildcard allowed.
- The pointcut represents a Spring-specific extension to the AspectJ expression language and it is useful only with Spring AOP.
- This pointcut designator offers two ways to select beans if you follow an appropriate naming convention:
  - Selecting a vertical slice of beans —If you follow a convention where bean names include a string indicating their role from the business perspective, a `bean()` pointcut can select beans based on business functionality.
  - Selecting a horizontal slice of beans —If you follow a convention where bean names include a string indicating their role from the architectural perspective, a `bean()` pointcut can select beans based on their architectural role.

Selecting Spring Beans

- Selecting horizontal and vertical slices of beans based on their names using the bean() pointcut designator:



Selecting Spring Beans

- You can use unary and binary operators to negate or combine `bean()` pointcuts as with other pointcuts.
- Examples:

`bean(inventoryRepository)` The bean named `inventoryRepository`

`bean(*)` Any bean

`bean(inventory*)` Any bean whose name starts with `inventory`

`bean(*Repository)` Any bean whose name ends with `Repository`

`bean(inventory/showInventory)` The bean named `inventory/showInventory` (eg. a controller handling that URL)

`bean(inventory/*)` Any bean whose name starts with `inventory`/ (eg. any controller handling inventory-related URLs)

`bean(inventory/*/edit)` Any bean whose name starts with `inventory/` and ends with `/edit` (eg. any controller handling the edit operation functionality related to inventory)

Advice

- Spring's @AspectJ integration supports all advice types: `before`, `after` (including `after returning` and `after throwing` variations), and `around`.
- They all follow the same syntax and semantics as the @AspectJ except for a few differences for the around advice.
- The `around` advice in Spring must also declare an argument of type `ProceedingJoinPoint` so that it can call the `proceed()` method on it.
- If the around advice needs to proceed with the same context, it can call the no-arg version of `proceed()`, again matching the core @AspectJ semantics.
- The difference comes when it wants to proceed with an altered context.
- In Spring, you must pass arguments to `proceed()` matching the join-point arguments.
- There is no way to modify the service (collected using the this() pointcut) with which the advice should proceed.
- In reality, in a typical Spring application, proceeding with altered context is uncommon.

Aspect Ordering

- Spring's integration does not support aspect precedence rules using @DeclarePrecedence annotation.
- It employs a scheme based on the `Ordered` interface.
- If an aspect needs to control ordering, it implements the `Ordered` interface and the configuration sets its order property, which specifies the relative order.

```
@Aspect
public class SecurityAspect implements Ordered {
   private int order;
   ... pointcut and advice etc.
   public void setOrder(int order) {
        this.order = order;
   }
}
```

Aspect Ordering

```
<bean id="securityAspect" class="ajia.security.SecurityAspect">
    <property name="order" value="1"/>
</bean>
<bean id="auditingAspect" class="ajia.auditing.AuditingAspect">
     <property name="order" value="2"/>
</bean>
```

- An aspect with a lower value for the **`order`** property has higher precedence over an aspect with a higher value.
- An alternative to the **`Ordered`** interface is **`@Order`**. The aspect is marked with the **`@Order`** annotation, whose value specifies the aspect's relative order.

```
@Aspect
@Order(1)
public class SecurityAspect {
    ...
}
```

Static crosscutting

- The @AspectJ integration supports the static crosscutting construct of declaring parent types by offering the **`@DeclareParents`** functionality.

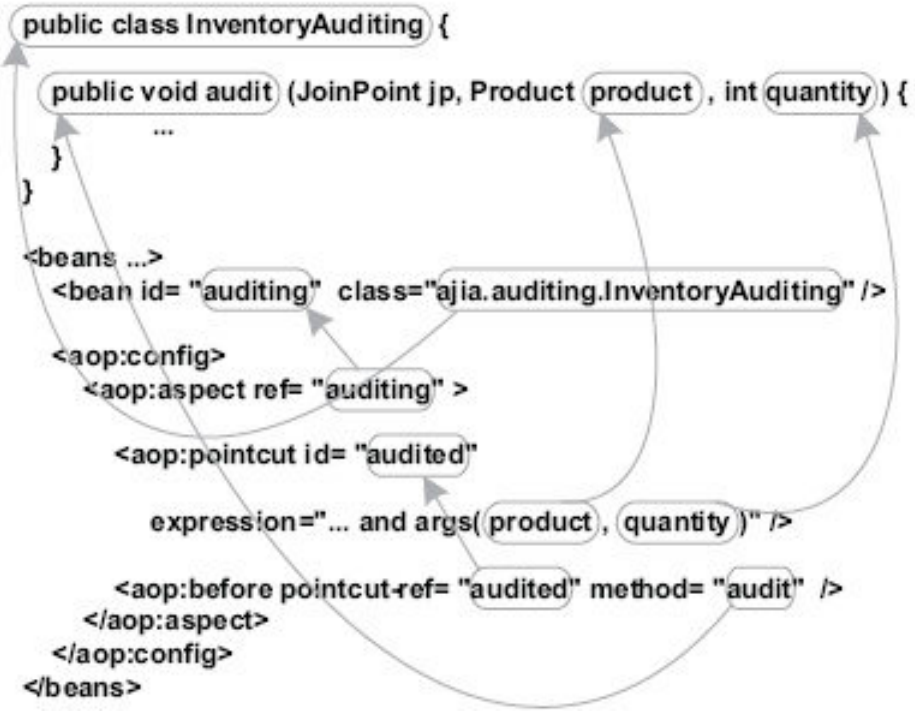
```
@Aspect
public class InventoryAuditing {
   @DeclareParents(value="ajia.service.*+",
        defaultImpl=AuditRecorderDefaultImpl.class)
   private AuditRecorder mixin;

   @Pointcut("execution(* *(ajia.domain.Product, int))&&
   this(auditRecorder) && args(product, quantity)")
   public void audited(AuditRecorder auditRecorder, Product
   product, int quantity) {}
  ...
}
```

Schema-style AOP support

- Spring provides an XML-based alternative, when Java 5 or above (annotation, @AspectJ syntax) cannot be used.
- Schema-style AOP support offers a way to turn a plain Java class into an aspect by specifying the aspect-related metadata using XML.
- The idea is similar to the @AspectJ syntax except here the crosscutting information is in XML form.
- By avoiding the use of Java annotations, the schema-styled AOP makes it possible to express AOP constructs in a form suitable to Java versions prior to Java 5, where there is no language support for annotations.
- Aspects stand in for class, methods stand in for advice, and pointcuts are described in XML. The use of context-collecting pointcuts such as args() lets you collect join point context and makes it available to advice in a strongly type manner.

Schema-style AOP support



Mapping aspects

- Plain classes along with metadata expressed in XML represent aspects in schema-style AOP. You create a bean of the class representing an aspect and declare an aspect based on that bean.
- An `<aop:aspect>` element inside an `<aop:config>` element declares the intention to use the bean referred by the `ref` attribute as an aspect.
- Declaring an `<aop:config>` element is an indication to Spring's application context loader that the aspects defined need to be applied to beans.
- This element may also declare an attribute `proxy-target-class` to instruct Spring to use class-based proxies created using CGLIB.
- The default value for this attribute is `false`, indicating the choice of JDK dynamic proxies.

Mapping pointcuts

- An `<aop:pointcut>` element inside `<aop:config>` or `<aop:aspect>` elements defines a pointcut.
- Any aspect may refer to the pointcuts defined at the `<aop:config>` level, making them global pointcuts.
- Only the enclosing aspect may refer to the pointcuts defined at the `<aop:aspect>` level.
- Each `<aop:pointcut>` element has two mandatory attributes: `id` and `expression`.
  - The `id` attribute assigns an identifier that an advice may later use to refer to the pointcut.
  - The expression attribute specifies a pointcut expression. No associated Java code is required to define pointcuts.
- A pointcut expression may use any of the supported pointcut designators.
- Although each pointcut is assigned an id, unlike the @AspectJ integration, it is not possible to use those ids to compose pointcuts the way you can using the traditional or @AspectJ syntax.

Mapping advice

- A method along with an XML element designating the kind of advice defines an advice in the schema-style AOP.
- The method's arguments serve as the join-point context, and the body serves as the advice implementation.
- Methods standing in for an advice do not have any special requirements: the methods must be public, non-static, and return void unless they represent an around advice.
- Following the @AspectJ syntax, a method may declare variables of `JoinPoint` or `JoinPoint.StaticPart` to obtain the join-point information. An around advice may also declare a variable of `ProceedingJoinPoint` to proceed with the advised join point.

Mapping advice

- A before advice is mapped using an **`<aop:before>`** element.
- An after advice is mapped using an **`<aop:after>`**, **`<aop:after-returning>`**, and **`<aop:after-throwing>`** element following the corresponding advice kind in AspectJ.
- An around advice is mapped using an **`<aop:around>`** element.
- Each of these elements may define the following attributes:
  - **`pointcut`**—This attribute specifies a pointcut expression.
  - **`pointcut-ref`**—This attribute specifies an id attribute to a **`<aop:pointcut>`** element.
    - The id must match a **`<aop:pointcut>`** element defined in the same **`<aop:aspect>`** definition as the advice or a global pointcut defined in an **`<aop:config>`** element.
    - The expression associated with the referred pointcut must match the same criteria of context-matching as described for the pointcut attribute.

Mapping advice

- Each of these elements may define the following attributes:
  - **`method`**—This attribute specifies the name of the method in the class of the referred bean (specified by the ref attribute of the enclosing **`<aop:aspect>`** element).
  - **`arg-names`**—This optional attribute specifies the name of the arguments for the method.

    It lets you map the name of the context collected by pointcut to the method parameters.

    It serves the same purpose as the **`argNames`** attribute in @AspectJ syntax. If you don't specify this parameter, you must compile the class standing for the referred aspect using either **`-g`** or **`-g:vars`**.

    As a last resort, Spring tries to deduce arguments from the parameter's type.
- Each advice kind may offer additional attributes and have additional particularities.

Mapping advice - Remarks

- The `<aop:before>` element does not specify any additional attributes.
- The `<aop:after-returning>` element may specify the `returning` attribute to collect and pass the return value to the advice. The value of this attribute must be one of the method arguments. The type of the argument must be compatible with the value returned by the advised join point.

```
<aop:after-returning pointcut="execution(String ship(Order))  &&
   args(order)“ method="recordShipment“ returning="trackingNumber">


public void recordShipment(Order order, String trackingNumber)
   {/*...*/}
```

- You can specify `Object` as the type to capture the return value of any type. Spring AOP boxes any primitives, if needed.
- The `<aop:after-throwing>` element is similar to the after returning advice, except you use the `throwing` attribute instead of `returning` attribute. The identifier specified by the throwing attribute gives access to the thrown exception.

Mapping advice - Remarks

- An around advice is mapped using an `<aop:around>` element. The method must declare a return value compatible with the selected join points.
- The return type `Object` is treated in the same way as in the @AspectJ syntax; it is compatible with any return type including primitives and void.
- The method typically declares a parameter of `ProceedingJoinPoint` type. This parameter lets you call `proceed()` when it wants to proceed with the advised join point.

Mapping static crosscutting

- With this feature, you can declare a new interface as the parent for beans matching a certain type pattern and delegate the implementation to a default implementation.
- This feature is much like the `declare parents` and `@DeclareParents` feature in AspectJ, with appropriate modifications to work with Spring AOP.
- An XML element maps to static crosscutting that offers a way to declare new interfaces as parent types. The declared interface and its default implementation are implemented in Java.

```
<aop:declare-parents types-matching="ajia.service.*"
implement-interface="ajia.auditing.AuditRecorder"
default-impl="ajia.auditing.AuditRecorderDefaultImpl"/>
```

Spring AOP vs AspectJ

- *The join point model —*
  - **Spring AOP** exposes only one kind of join point: execution of public non-static methods.
  - **AspectJ** exposes several in addition to method execution, such as object construction, class loading, method call, exception handler, and field access. Spring AOP also implicitly limits exposed join points to those for the beans. AspectJ, has a far reaching crosscutting capability.
- *Adoption complexity —*
  - **Spring AOP** has little adoption cost.

    You do not need to employ any special tools or modify the build or execution environment.

    The AOP concepts you need to learn are far fewer in Spring AOP.

    Spring AOP's limited power can be seen as a benefit; fewer things can go wrong while you are learning how to use AOP!

Spring AOP vs AspectJ

- *Configurability —*
  - **Spring AOP** is an object-based AOP and thus offers an easy way to configure crosscutting functionality for an individual bean.
  - **AspectJ** is type-based AOP and thus offers simple configuration at the type level.
  - Spring's autoproxy mechanism makes it possible to apply uniform crosscutting across multiple beans on the same type.
  - AspectJ needs to rely on some form of object identity (name, id, etc.) to distinguish between multiple instances of the same type.
- *Performance —*
  - Spring's proxy-based AOP, due to the use of reflection necessitated by proxies, has lower performance than byte-code weaving implemented by AspectJ.
  - Most applications of Spring AOP involve advising already expensive operations (database accesses or network operations).
  - In those cases, the overhead added by Spring's proxy is unimportant.
  - In general, Spring AOP's applications are self-selecting regarding performance characteristics—if Spring AOP is suitable for functionality, the performance characteristics are likely to be acceptable.