9. Aspect Oriented Programming with Spring

9.1 Introduction

Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

> **Spring 2.0 AOP**
>
> Spring 2.0 introduces a simpler and more powerful way of writing custom aspects using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.
>
> The Spring 2.0 schema- and @AspectJ-based AOP support is discussed in this chapter. Spring 2.0 AOP remains fully backwards compatible with Spring 1.2 AOP, and the lower-level AOP support offered by the Spring 1.2 APIs is discussed in the following chapter.

AOP is used in the Spring Framework to...

- ... provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is *declarative transaction management*.
- ... allow users to implement custom aspects, complementing their use of OOP with AOP.

If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.

9.1.1 AOP concepts

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific... unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

- *Aspect*: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the `@Aspect` annotation (the `@AspectJ` style).
- *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
- *Advice*: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.
- *Pointcut*: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
- *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
- *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
- *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- *Weaving*: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

- *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns 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 of the means by which a join point exits (normal or exceptional return).
- *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the `proceed()` method on the `JoinPoint` used for around advice, and hence cannot fail to invoke it.

In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than `Object` arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

9.1.2 Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring 2.0 seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

One of the central tenets of the Spring Framework is that of *non-invasiveness*; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.
One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.
See Section 9.4, “Choosing which AOP declaration style to use” for a more complete discussion of the whys and wherefores of each style.

9.1.3 AOP Proxies

Spring AOP defaults to using standard J2SE *dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes, business classes normally will implement one or more business interfaces. It is possible to force the use of CGLIB, in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is *proxy-based*. See Section 9.6.1, "Understanding AOP proxies" for a thorough examination of exactly what this implementation detail actually means.

9.2 @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring 2.0 interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Using the AspectJ compiler and weaver enables use of the full AspectJ language, and is discussed in Section 9.8, "Using AspectJ with Spring applications".

9.2.1 Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and *autoproxying* beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support can be enabled with XML or Java style configuration. In either case you will also need to ensure that AspectJ's `aspectjweaver.jar` library is on the classpath of your application (version 1.6.8 or later). This library is available in the `'lib'` directory of an AspectJ distribution or via the Maven Central repository.

Enabling @AspectJ Support with Java configuration

To enable @AspectJ support with Java `@Configuration` add the `@EnableAspectJAutoProxy` annotation:

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

Enabling @AspectJ Support with XML configuration

To enable @AspectJ support with XML based configuration use the `aop:aspectj-autoproxy` element:

```
<aop:aspectj-autoproxy/>
```

This assumes that you are using schema support as described in Appendix E, *XML Schema-based configuration*. See Section E.2.7, "The `aop` schema" for how to import the tags in the aop namespace.

If you are using the DTD, it is still possible to enable @AspectJ support by adding the following definition to your application context:

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

9.2.2 Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the `@Aspect` annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:

A regular bean definition in the application context, pointing to a bean class that has the `@Aspect` annotation:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
   <!-- configure properties of aspect here as normal -->
</bean>
```

And the `NotVeryUsefulAspect` class definition, annotated with `org.aspectj.lang.annotation.Aspect` annotation;

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Aspects (classes annotated with `@Aspect`) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

You may register aspect classes as regular beans in your Spring XML configuration, or autodetect them through classpath scanning - just like any other Spring-managed bean. However, note that the *@Aspect* annotation is *not* sufficient for autodetection in the classpath: For that purpose, you need to add a separate *@Component* annotation (or alternatively a custom stereotype annotation that qualifies, as per the rules of Spring's component scanner).

In Spring AOP, it is *not* possible to have aspects themselves be the target of advice from other aspects. The *@Aspect* annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

9.2.3 Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *Spring AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines *exactly* which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the `@Pointcut` annotation (the method serving as the pointcut signature *must* have a `void` return type).

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named `'anyOldTransfer'` that will match the execution of any method named `'transfer'`:

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

The pointcut expression that forms the value of the `@Pointcut` annotation is a regular AspectJ 5 pointcut expression. For a full discussion of AspectJ's pointcut language, see the AspectJ Programming Guide (and for Java 5 based extensions, the AspectJ 5 Developers Notebook) or one of the books on AspectJ such as “Eclipse AspectJ” by Colyer et. al. or “AspectJ in Action” by Ramnivas Laddad.

Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

> **Other pointcut types**
>
> The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: `call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this` and `@withincode`. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an `IllegalArgumentException` being thrown.
>
> The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

- *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- `@target` - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- `@args` - limits matching to join points (the execution of methods when using Spring AOP) 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 (the execution of methods declared in types with the given annotation when using Spring AOP)
- *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both '`this`' and '`target`' refer to the same object - the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (bound to '`this`') and the target object behind the proxy (bound to '`target`').

Due to the proxy-based nature of Spring's AOP framework, protected methods are by definition *not* intercepted, neither for JDK proxies (where this isn't applicable) nor for CGLIB proxies (where this is technically possible but not recommendable for AOP purposes). As a consequence, any given pointcut will be matched against *public methods only*!
If your interception needs include protected/private methods or even constructors, consider the use of Spring-driven native AspectJ weaving instead of Spring's proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision.

Spring AOP also supports an additional PCD named '`bean`'. This PCD allows you to limit the matching of join points to a particular named Spring bean, or to a set of named Spring beans (when using wildcards). The '`bean`' PCD has the following form:

```
bean(idOrNameOfBean)
```

The '`idOrNameOfBean`' token can be the name of any Spring bean: limited wildcard support using the '`*`' character is provided, so if you establish some naming conventions for your Spring beans you can quite easily write a '`bean`' PCD expression to pick them out. As is the case with other pointcut designators, the '`bean`' PCD can be &&'ed, ||'ed, and ! (negated) too.

Please note that the '`bean`' PCD is *only* supported in Spring AOP - and *not* in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines.
The '`bean`' PCD operates at the *instance* level (building on the Spring bean name concept) rather than at the type level only (which is what weaving-based AOP is limited to). Instance-based pointcut designators are a special capability of Spring's proxy-based AOP framework and its close integration with the Spring bean factory, where it is natural and straightforward to identify specific beans by name.

Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: `anyPublicOperation` (which matches if a method execution join point represents the execution of any public method); `inTrading` (which matches if a method execution is in the trading module), and `tradingOperation` (which matches if a method execution represents any public method in the trading module).

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

    @Pointcut("within(com.xyz.someapp.trading..*)")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut *matching*.

Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.someapp.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.someapp.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
```

```
    * in a type in the com.xyz.someapp.dao package or any sub-package
    * under that.
    */
   @Pointcut("within(com.xyz.someapp.dao..*)")
   public void inDataAccessLayer() {}

   /**
    * A business service is the execution of any method defined on a service
    * interface. This definition assumes that interfaces are placed in the
    * "service" package, and that implementation types are in sub-packages.
    *
    * If you group service interfaces by functional area (for example,
    * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
    * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
    * could be used instead.
    *
    * Alternatively, you can write the expression using the 'bean'
    * PCD, like so "bean(*Service)". (This assumes that you have
    * named your Spring service beans in a consistent fashion.)
    */
   @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
   public void businessService() {}

   /**
    * A data access operation is the execution of any method defined on a
    * dao interface. This definition assumes that interfaces are placed in the
    * "dao" package, and that implementation types are in sub-packages.
    */
   @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
   public void dataAccessOperation() {}

}
```

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

```
<aop:config>
  <aop:advisor
      pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
      advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
```

```
  </tx:attributes>
</tx:advice>
```

The `<aop:config>` and `<aop:advisor>` elements are discussed in Section 9.3, "Schema-based AOP support". The transaction elements are discussed in Chapter 12, *Transaction Management*.

Examples

Spring AOP users are likely to use the `execution` pointcut designator the most often. The format of an execution expression is:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
          throws-pattern?)
```

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use `*` as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the `*` wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: `()` matches a method that takes no parameters, whereas `(..)` matches any number of parameters (zero or more). The pattern `(*)` matches a method taking one parameter of any type, `(*,String)` matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

- the execution of any public method:

  ```
  execution(public * *(..))
  ```

- the execution of any method with a name beginning with "set":

  ```
  execution(* set*(..))
  ```

- the execution of any method defined by the `AccountService` interface:

  ```
  execution(* com.xyz.service.AccountService.*(..))
  ```

- the execution of any method defined in the service package:

  ```
  execution(* com.xyz.service.*.*(..))
  ```

- the execution of any method defined in the service package or a sub-package:

  ```
  execution(* com.xyz.service..*.*(..))
  ```

- any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

- any join point (method execution only in Spring AOP) within the service package or a sub-package:

```
within(com.xyz.service..*)
```

- any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```

'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.

- any join point (method execution only in Spring AOP) where the target object implements the `AccountService` interface:

```
target(com.xyz.service.AccountService)
```

'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is `Serializable`:

```
args(java.io.Serializable)
```

'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.
Note that the pointcut given in this example is different to `execution(* *(java.io.Serializable))`: the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type `Serializable`.

- any join point (method execution only in Spring AOP) where the target object has an `@Transactional` annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```

'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the declared type of the target object has an `@Transactional` annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```

'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the executing method has an `@Transactional` annotation:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the `@Classified` annotation:

```
@args(com.xyz.security.Classified)
```

'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.

- any join point (method execution only in Spring AOP) on a Spring bean named '`tradeService`':

```
bean(tradeService)
```

- any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression '`*Service`':

```
bean(*Service)
```

Writing good pointcuts

During compilation, AspectJ processes pointcuts in order to try and optimize matching performance. Examining code and determining if each join point matches (statically or dynamically) a given pointcut is a costly process. (A dynamic match means the match cannot be fully determined from static analysis and a test will be placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ will rewrite it into an optimal form for the matching process. What does this mean? Basically pointcuts are rewritten in DNF (Disjunctive Normal Form) and the components of the pointcut are sorted such that those components that are cheaper to evaluate are checked first. This means you do not have to worry about understanding the performance of various pointcut designators and may supply them in any order in a pointcut declaration.

However, AspectJ can only work with what it is told, and for optimal performance of matching you should think about what they are trying to achieve and narrow the search space for matches as much as possible in the definition. The existing designators naturally fall into one of three groups: kinded, scoping and context:

- Kinded designators are those which select a particular kind of join point. For example: execution, get, set, call, handler
- Scoping designators are those which select a group of join points of interest (of probably many kinds). For example: within, withincode
- Contextual designators are those that match (and optionally bind) based on context. For example: this, target, @annotation

A well written pointcut should try and include at least the first two types (kinded and scoping), whilst the contextual designators may be included if wishing to match based on join point context, or bind that context for use in the advice. Supplying either just a kinded designator or just a contextual designator will work but could affect weaving performance (time and memory used) due to all the extra processing and analysis. Scoping designators are very fast to match and their usage means AspectJ can very quickly dismiss groups of join points that should not be further processed - that is why a good pointcut should always include one if possible.

9.2.4 Declaring 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 pointcut expression declared in place.

Before advice

Before advice is 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.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

If using an in-place pointcut expression we could rewrite the above example as:

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

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the `@AfterReturning` annotation:

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

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We're just showing a single advice declaration in these examples to focus on the issue under discussion at the time.

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of `@AfterReturning` that binds the return value for this:

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

@Aspect
public class AfterReturningExample {

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

}
```

The name used in the `returning` attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A `returning` clause also restricts matching to only those method executions that return a value of the specified type (`Object` in this case, which will match any return value).

Please note that it is *not* possible to return a totally different reference when using after-returning advice.

After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the `@AfterThrowing` annotation:

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

@Aspect
public class AfterThrowingExample {

  @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }

}
```

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the `throwing` attribute to both restrict matching (if desired, use `Throwable` as the exception type otherwise) and bind the thrown exception to an advice parameter.

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

@Aspect
public class AfterThrowingExample {

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

}
```

The name used in the `throwing` attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A `throwing` clause also restricts matching to only those method executions that throw an exception of the specified type (`DataAccessException` in this case).

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `@After` annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

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

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
    // ...
  }

}
```

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don't use around advice if simple before advice would do).

Around advice is declared using the `@Around` annotation. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be called passing in an `Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds.

The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to (Don't worry if this doesn't make sense right now!). The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you are compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

Advice parameters

Spring 2.0 offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with `Object[]` arrays all the time. We'll see how to make argument and other contextual values available to the advice body in a moment. First

let's take a look at how to write generic advice that can find out about the method the advice is currently advising.

Access to the current `JoinPoint`

Any advice method may declare as its first parameter, a parameter of type `org.aspectj.lang.JoinPoint` (please note that around advice is *required* to declare a first parameter of type `ProceedingJoinPoint`, which is a subclass of `JoinPoint`. The `JoinPoint` interface provides a number of useful methods such as `getArgs()` (returns the method arguments), `getThis()` (returns the proxy object), `getTarget()` (returns the target object), `getSignature()` (returns a description of the method that is being advised) and `toString()` (prints a useful description of the method being advised). Please do consult the Javadocs for full details.

Passing parameters to advice

We've already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of `args`. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
        "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

The `args(account,..)` part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of `Account`; secondly, it makes the actual `Account` object available to the advice via the `account` parameter.

Another way of writing this is to declare a pointcut that "provides" the `Account` object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
          "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
  // ...
}
```

The interested reader is once more referred to the AspectJ programming guide for more details.

The proxy object (`this`), target object (`target`), and annotations (`@within, @target, @annotation, @args`) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an `@Auditable` annotation, and extract the audit code.

First the definition of the `@Auditable` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
    AuditCode value();
}
```

And then the advice that matches the execution of `@Auditable` methods:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
        "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

Advice parameters and generics

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like this:

```
public interface Sample<T> {
  void sampleGenericMethod(T param);
  void sampleGenericCollectionMethod(Collection>T> param);
}
```

You can restrict interception of method types to certain parameter types by simply typing the advice parameter to the parameter type you want to intercept the method for:

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
  // Advice implementation
}
```

That this works is pretty obvious as we already discussed above. However, it's worth pointing out that this won't work for generic collections. So you cannot define a pointcut like this:

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
  // Advice implementation
}
```

To make this work we would have to inspect every element of the collection, which is not reasonable as we also cannot decide how to treat `null` values in general. To achieve something similar to this you have to type the parameter to `Collection<?>` and manually check the type of the elements.

Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are *not* available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

1. If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names *are* available at runtime. For example:

```
@Before(
   value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
   argNames="bean,auditable")
public void audit(Object bean, Auditable auditable) {
  AuditCode code = auditable.value();
  // ... use code and bean
}
```

   If the first parameter is of the `JoinPoint`, `ProceedingJoinPoint`, or `JoinPoint.StaticPart` type, you may leave out the name of the parameter from the value of the "argNames" attribute. For example, if you modify the preceding advice to receive the join point object, the "argNames" attribute need not include it:

```
@Before(
   value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
   argNames="bean,auditable")
public void audit(JoinPoint jp, Object bean, Auditable auditable) {
  AuditCode code = auditable.value();
  // ... use code, bean, and jp
}
```

   The special treatment given to the first parameter of the `JoinPoint`, `ProceedingJoinPoint`, and `JoinPoint.StaticPart` types is particularly convenient for advice that do not collect any other join point context. In such situations, you may simply omit the "argNames" attribute. For example, the following advice need not declare the "argNames" attribute:

```
@Before(
   "com.xyz.lib.Pointcuts.anyPublicMethod()")
public void audit(JoinPoint jp) {
  // ... use jp
}
```

2. Using the `'argNames'` attribute is a little clumsy, so if the `'argNames'` attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information (`'-g:vars'` at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.

If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) even without the debug information then there is no need to add the argNames attribute as the compiler will retain the needed information.

3. If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an `AmbiguousBindingException` will be thrown.
4. If all of the above strategies fail then an `IllegalArgumentException` will be thrown.

Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call *with arguments* that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

```
@Around("execution(List<Account> find*(..)) &&" +
        "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

In many cases you will be doing this binding anyway (as in the example above).

Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second).

When two pieces of advice defined in *different* aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the `org.springframework.core.Ordered` interface in the aspect class or annotating it with the `Order` annotation. Given two aspects, the aspect returning the lower value from `Ordered.getValue()` (or the annotation value) has the higher precedence.

When two pieces of advice defined in *the same* aspect both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the declaration order via reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per join point in each aspect class, or refactor the pieces of advice into separate aspect classes - which can be ordered at the aspect level.

9.2.5 Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `@DeclareParents` annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
          "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

The interface to be implemented is determined by the type of the annotated field. The `value` attribute of the `@DeclareParents` annotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

9.2.6 Aspect instantiation models

(This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.)

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ's `perthis` and `pertarget` instantiation models (`percflow, percflowbelow,` and `pertypewithin` are not currently supported).

A "perthis" aspect is declared by specifying a `perthis` clause in the `@Aspect` annotation. Let's look at an example, and then we'll explain how it works.

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;
```

```
  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
    // ...
  }

}
```

The effect of the `'perthis'` clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The `'pertarget'` instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

9.2.7 Example

Now that you have seen how all the constituent parts work, let's put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks:

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
```

```
        this.order = order;
    }

    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        }
        while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }

}
```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice. Notice that for the moment we're applying the retry logic to all `businessService()s`. We try to proceed, and if we fail with an `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
     <property name="maxRetries" value="3"/>
     <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it only retries idempotent operations, we might define an `Idempotent` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

9.3 Schema-based AOP support

If you are unable to use Java 5, or simply prefer an XML-based format, then Spring 2.0 also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new *syntax* and refer the reader to the discussion in the previous section (Section 9.2, "@AspectJ support") for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described in Appendix E, *XML Schema-based configuration*. See Section E.2.7, "The `aop` schema" for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an `<aop:config>` element (you can have more than one `<aop:config>` element in an application context configuration). An `<aop:config>` element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

The `<aop:config>` style of configuration makes heavy use of Spring's auto-proxying mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of `BeanNameAutoProxyCreator` or suchlike. The recommended usage pattern is to use either just the `<aop:config>` style, or just the `AutoProxyCreator` style.

9.3.1 Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is 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>
```

The bean backing the aspect ("`aBean`" in this case) can of course be configured and dependency injected just like any other Spring bean.

9.3.2 Declaring a pointcut

A named pointcut can be declared inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in Section 9.2, "@AspectJ support". If you are using the schema based declaration style with Java 5, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression, but this feature is not available on JDK 1.4 and below (it relies on the Java 5 specific AspectJ reflection APIs). On JDK 1.5 therefore, another way of defining the above pointcut would be:

```
<aop:config>

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

</aop:config>
```

Assuming you have a `SystemArchitecture` aspect as described in the section called "Sharing common pointcut definitions".

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
          expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    ...
```

```
  </aop:aspect>

</aop:config>
```

Much the same way in an @AspectJ aspect, pointcuts declared using the schema based definition style may collect join point context. For example, the following pointcut collects the 'this' object as the join point context and passes it to advice:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
          expression="execution(* com.xyz.myapp.service.*.*(..)) &amp;&amp; this(service)"/>
    <aop:before pointcut-ref="businessService" method="monitor"/>
    ...

  </aop:aspect>

</aop:config>
```

The advice must be declared to receive the collected join point context by including parameters of the matching names:

```
public void monitor(Object service) {
    ...
}
```

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively. For example, the previous pointcut may be better written as:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
          expression="execution(* com.xyz.myapp.service.*.*(..)) and this(service)"/>
    <aop:before pointcut-ref="businessService" method="monitor"/>
    ...

  </aop:aspect>

</aop:config>
```

Note that pointcuts defined in this way are referred to by their XML id and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

9.3.3 Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

Before advice

Before advice runs before a matched method execution. It is declared inside an `<aop:aspect>` using the <aop:before> element.

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Here `dataAccessOperation` is the id of a pointcut defined at the top (`<aop:config>`) level. To define the pointcut inline instead, replace the `pointcut-ref` attribute with a `pointcut` attribute:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method (`doAccessCheck`) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an `<aop:aspect>` in the same way as before advice. For example:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      returning="retVal"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

The doAccessCheck method must declare a parameter named `retVal`. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

```
public void doAccessCheck(Object retVal) {...
```

After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an `<aop:aspect>` using the after-throwing element:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      throwing="dataAccessEx"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

The doRecoveryActions method must declare a parameter named `dataAccessEx`. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `after` element:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
      pointcut-ref="dataAccessOperation"
      method="doReleaseLock"/>

    ...

</aop:aspect>
```

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don't use around advice if simple before advice would do.

Around advice is declared using the `aop:around` element. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be calling passing in an

`Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds. See the section called “Around advice” for notes on calling proceed with an `Object[]`.

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
      pointcut-ref="businessService"
      method="doBasicProfiling"/>

    ...

</aop:aspect>
```

The implementation of the `doBasicProfiling` advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

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

Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See the section called “Advice parameters” for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described) then this is done using the `arg-names` attribute of the advice element, which is treated in the same manner to the "argNames" attribute in an advice annotation as described in the section called “Determining argument names”. For example:

```
<aop:before
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
  method="audit"
  arg-names="auditable"/>
```

The `arg-names` attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

```
package x.y.service;

public interface FooService {
```

```
   Foo getFoo(String fooName, int age);
}

public class DefaultFooService implements FooService {

   public Foo getFoo(String name, int age) {
      return new Foo(name, age);
   }
}
```

Next up is the aspect. Notice the fact that the `profile(..)` method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the `profile(..)` is to be used as `around` advice:

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

   public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
      StopWatch clock = new StopWatch(
            "Profiling for '" + name + "' and '" + age + "'");
      try {
         clock.start(call.toShortString());
         return call.proceed();
      } finally {
         clock.stop();
         System.out.println(clock.prettyPrint());
      }
   }
}
```

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular join point:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

   <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
   <bean id="fooService" class="x.y.service.DefaultFooService"/>
```

```
   <!-- this is the actual advice itself -->
   <bean id="profiler" class="x.y.SimpleProfiler"/>

   <aop:config>
      <aop:aspect ref="profiler">

         <aop:pointcut id="theExecutionOfSomeFooServiceMethod"
                   expression="execution(* x.y.service.FooService.getFoo(String,int))
                   and args(name, age)"/>

         <aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"
                  method="profile"/>

      </aop:aspect>
   </aop:config>

</beans>
```

If we had the following driver script, we would get output something like this on standard output:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.FooService;

public final class Boot {

   public static void main(final String[] args) throws Exception {
      BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
      FooService foo = (FooService) ctx.getBean("fooService");
      foo.getFoo("Pengo", 12);
   }
}
```

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
-----------------------------------------
ms     %     Task name
-----------------------------------------
00000  ?  execution(getFoo)
```

Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in the section called “Advice ordering”. The precedence between aspects is determined by either adding the `Order` annotation to the bean backing the aspect or by having the bean implement the `Ordered` interface.

9.3.4 Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `aop:declare-parents` element inside an `aop:aspect` This element is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
      types-matching="com.xzy.myapp.service.*+"
      implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
      default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
    pointcut="com.xyz.myapp.SystemArchitecture.businessService()
              and this(usageTracked)"
    method="recordUsage"/>

</aop:aspect>
```

The class backing the `usageTracking` bean would contain the method:

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

The interface to be implemented is determined by `implement-interface` attribute. The value of the `types-matching` attribute is an AspectJ type pattern :- any bean of a matching type will implement the `UsageTracked` interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

9.3.5 Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

9.3.6 Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring 1.2 and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in Section 10.3.2, “Advice types in Spring”. Advisors can take advantage of AspectJ pointcut expressions though.

Spring 2.0 supports the advisor concept with the `<aop:advisor>` element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring 2.0. Here's how it looks:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
      pointcut-ref="businessService"
      advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

As well as the `pointcut-ref` attribute used in the above example, you can also use the `pointcut` attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the `order` attribute to define the `Ordered` value of the advisor.

9.3.7 Example

Let's see how the concurrent locking failure retry example from Section 9.2.7, “Example” looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks (it's just a regular Java class using the schema support):

```
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;
```

```
   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice method. We try to proceed, and if we fail with a `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.

This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

```
<aop:config>

  <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">
```

```
    <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:around
       pointcut-ref="idempotentOperation"
       method="doConcurrentOperation"/>

  </aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
     <property name="maxRetries" value="3"/>
     <property name="order" value="100"/>
</bean>
```

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an `Idempotent` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and using the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```
  <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and
                    @annotation(com.xyz.myapp.service.Idempotent)"/>
```

9.4 Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

9.4.1 Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects typically), then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you are not using Java 5+ then the choice has been made for you... use the code style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) plugin for Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

9.4.2 @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. Clearly if you are not running on Java 5+, then the XML style is the appropriate choice; for Java 5 projects there are various tradeoffs to consider.

The XML style will be most familiar to existing Spring users. It can be used with any JDK level (referring to named pointcuts from within pointcut expressions does still require Java 5+ though) and is backed by genuine POJOs. When using AOP as a tool to configure enterprise services then XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently). With the XML style arguably it is clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of *how* a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like:

```
@Pointcut(execution(* get*()))
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..))
public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

In the XML style I can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
    expression="execution(* get*())"/>
```

```
<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

The downside of the XML approach is that you cannot define the '`accountPropertyAccess`' pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ-based approach. On balance the Spring team prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

9.5 Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined `<aop:aspect>` aspects, `<aop:advisor>` declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

9.6 Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

- `final` methods cannot be advised, as they cannot be overridden.
- As of Spring 3.2, it is no longer necessary to add CGLIB to your project classpath, as CGLIB classes are repackaged under org.springframework and included directly in the spring-core JAR. This means that CGLIB-based proxy support 'just works' in the same way that JDK dynamic proxies always have.
- The constructor of your proxied object will be called twice. This is a natural consequence of the CGLIB proxy model whereby a subclass is generated for each proxied object. For each proxied instance, two objects are created: the actual proxied object and an instance of the subclass that implements the advice. This behavior is not exhibited when using JDK proxies. Usually, calling the constructor of the proxied type twice, is not an issue, as there are usually only assignments taking place and no real logic is implemented in the constructor.

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

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

To force CGLIB proxying when using the @AspectJ autoproxy support, set the `'proxy-target-class'` attribute of the `<aop:aspectj-autoproxy>` element to `true`:

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

Multiple `<aop:config/>` sections are collapsed into a single unified auto-proxy creator at runtime, which applies the *strongest* proxy settings that any of the `<aop:config/>` sections (typically from different XML bean definition files) specified. This also applies to the `<tx:annotation-driven/>` and `<aop:aspectj-autoproxy/>` elements.
To be clear: using '`proxy-target-class="true"`' on `<tx:annotation-driven/>`, `<aop:aspectj-autoproxy/>` or `<aop:config/>` elements will force the use of CGLIB proxies *for all three of them*.

9.6.1 Understanding AOP proxies

Spring AOP is *proxy-based*. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based aspects supplied with the Spring Framework.

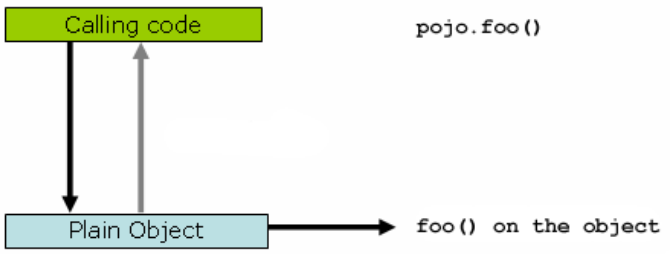
Consider first the scenario where you have a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference, as illustrated by the following code snippet.

```
public class SimplePojo implements Pojo {

   public void foo() {
      // this next method invocation is a direct
      call on the 'this' reference
      this.bar();
   }

   public void bar() {
      // some logic...
   }
}
```

If you invoke a method on an object reference, the method is invoked *directly* on that object reference, as can be seen below.



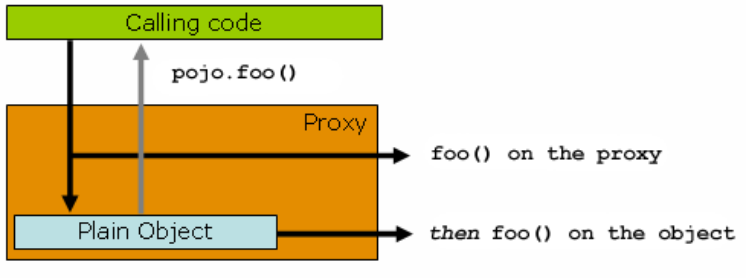
```
public class Main {

   public static void main(String[] args) {

      Pojo pojo = new SimplePojo();

      // this is a direct method call on the 'pojo' reference
      pojo.foo();
   }
}
```

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet.




```
public class Main {

   public static void main(String[] args) {

      ProxyFactory factory = new ProxyFactory(new SimplePojo());
      factory.addInterface(Pojo.class);
      factory.addAdvice(new RetryAdvice());

      Pojo pojo = (Pojo) factory.getProxy();

      // this is a method call on the proxy!
      pojo.foo();
   }
}
```

The key thing to understand here is that the client code inside the `main(..)` of the `Main` class *has a reference to the proxy*. This means that method calls on that object reference will be calls on the proxy, and as such the proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object, the `SimplePojo` reference in this case, any method calls that it may make on itself, such as `this.bar()` or `this.foo()`, are going to be invoked against the `this` reference, and *not* the proxy. This has important implications. It means that self-invocation is *not* going to result in the advice associated with a method invocation getting a chance to execute.

Okay, so what is to be done about this? The best approach (the term best is used loosely here) is to refactor your code such that the self-invocation does not happen. For sure, this does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and I am almost reticent to point it out precisely because it is so horrendous. You can (choke!) totally tie the logic within your class to Spring AOP by doing this:

```
public class SimplePojo implements Pojo {

   public void foo() {
      // this works, but... gah!
      ((Pojo) AopContext.currentProxy()).bar();
   }

   public void bar() {
      // some logic...
   }
}
```

This totally couples your code to Spring AOP, *and* it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created:

```
public class Main {

   public static void main(String[] args) {

      ProxyFactory factory = new ProxyFactory(new SimplePojo());
      factory.adddInterface(Pojo.class);
      factory.addAdvice(new RetryAdvice());
      factory.setExposeProxy(true);

      Pojo pojo = (Pojo) factory.getProxy();

      // this is a method call on the proxy!
      pojo.foo();
   }
}
```

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

9.7 Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either `<aop:config>` or `<aop:aspectj-autoproxy>`, it is also possible programmatically to create proxies that advise target objects. For the full details of Spring's AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class `org.springframework.aop.aspectj.annotation.AspectJProxyFactory` can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the Javadocs for full information.

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

9.8 Using AspectJ with Spring applications

Everything we've covered so far in this chapter is pure Spring AOP. In this section, we're going to look at how you can use the AspectJ compiler/weaver instead of, or in addition to, Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library, which is available standalone in your distribution as `spring-aspects.jar`; you'll need to add this to your classpath in order to use the aspects in it. Section 9.8.1, “Using AspectJ to dependency inject domain objects with Spring” and Section 9.8.2, “Other Spring aspects for AspectJ” discuss the content of this library and how you can use it. Section 9.8.3, “Configuring AspectJ aspects using Spring IoC” discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, Section 9.8.4, “Load-time weaving with AspectJ in the Spring Framework” provides an introduction to load-time weaving for Spring applications using AspectJ.

9.8.1 Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a *pre-existing* object given the name of a bean definition containing the configuration to be applied. The `spring-aspects.jar` contains an annotation-driven aspect that exploits this capability to allow dependency injection of *any object*. The support is intended to be used for objects created *outside of the control of any container*. Domain objects often fall into this category because they are often created programmatically using the `new` operator, or by an ORM tool as a result of a database query.

The `@Configurable` annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
```

```
public class Account {
   // ...
}
```

When used as a marker interface in this way, Spring will configure new instances of the annotated type (`Account` in this case) using a bean definition (typically prototype-scoped) with the same name as the fully-qualified type name (`com.xyz.myapp.domain.Account`). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the `id` attribute:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
  <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
   // ...
}
```

Spring will now look for a bean definition named "`account`" and use that as the definition to configure new `Account` instances.

You can also use autowiring to avoid having to specify a dedicated bean definition at all. To have Spring apply autowiring use the '`autowire`' property of the `@Configurable` annotation: specify either `@Configurable(autowire=Autowire.BY_TYPE)` or `@Configurable(autowire=Autowire.BY_NAME` for autowiring by type or by name respectively. As an alternative, as of Spring 2.5 it is preferable to specify explicit, annotation-driven dependency injection for your `@Configurable` beans by using `@Autowired` or `@Inject` at the field or method level (see Section 5.9, “Annotation-based container configuration” for further details).

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the `dependencyCheck` attribute (for example: `@Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true)`). If this attribute is set to true, then Spring will validate after configuration that all properties (*which are not primitives or collections*) have been set.

Using the annotation on its own does nothing of course. It is the `AnnotationBeanConfigurerAspect` in `spring-aspects.jar` that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object of a type annotated with `@Configurable`, configure the newly created object using Spring in accordance with the properties of the annotation". In this context, *initialization* refers to newly instantiated objects (e.g., objects instantiated with the '`new`' operator) as well as to `Serializable` objects that are undergoing deserialization (e.g., via readResolve()).

One of the key phrases in the above paragraph is '*in essence*'. For most cases, the exact semantics of '*after returning from the initialization of a new object*' will be fine... in this context, '*after initialization*' means that the dependencies will be injected *after* the object has been constructed - this means that the

> dependencies will not be available for use in the constructor bodies of the class. If you want the dependencies to be injected *before* the constructor bodies execute, and thus be available for use in the body of the constructors, then you need to define this on the `@Configurable` declaration like so:
>
> ```
> @Configurable(preConstruction=true)
> ```
>
> You can find out more information about the language semantics of the various pointcut types in AspectJ in this appendix of the AspectJ Programming Guide.

For this to work the annotated types must be woven with the AspectJ weaver - you can either use a build-time Ant or Maven task to do this (see for example the AspectJ Development Environment Guide) or load-time weaving (see Section 9.8.4, "Load-time weaving with AspectJ in the Spring Framework"). The `AnnotationBeanConfigurerAspect` itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you are using Java based configuration simply add `@EnableSpringConfigured` to any `@Configuration` class.

```
@Configuration
@EnableSpringConfigured
public class AppConfig {

}
```

If you prefer XML based configuration, the Spring `context` namespace defines a convenient `context:spring-configured` element:

```
<context:spring-configured/>
```

If you are using the DTD instead of schema, the equivalent definition is:

```
<bean
      class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"
      factory-method="aspectOf"/>
```

Instances of `@Configurable` objects created *before* the aspect has been configured will result in a message being issued to the debug log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

```
<bean id="myService"
  class="com.xzy.myapp.service.MyService"
  depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

  <!-- ... -->

</bean>
```

> Do not activate `@Configurable` processing through the bean configurer aspect unless you really mean to rely on its semantics at runtime. In particular,

> make sure that you do not use `@Configurable` on bean classes which are registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the aspect.

Unit testing `@Configurable` objects

One of the goals of the `@Configurable` support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If `@Configurable` types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If `@Configurable` types *have* been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an `@Configurable` object indicating that it has not been configured by Spring.

Working with multiple application contexts

The `AnnotationBeanConfigurerAspect` used to implement the `@Configurable` support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of `static` members, that is to say there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the `@EnableSpringConfigured` bean and where to place `spring-aspects.jar` on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the `AnnotationBeanConfigurerAspect` can only hold a reference to one of them. In this case we recommend defining the `@EnableSpringConfigured` bean in the shared (parent) application context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the @Configurable mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in `spring-aspects.jar` using its own classloader (for example, by placing `spring-aspects.jar` in `'WEB-INF/lib'`). If `spring-aspects.jar` is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

9.8.2 Other Spring aspects for AspectJ

In addition to the `@Configurable` aspect, `spring-aspects.jar` contains an AspectJ aspect that can be used to drive Spring's transaction management for types and methods annotated with the `@Transactional` annotation. This is primarily intended for users who want to use the Spring Framework's transaction support outside of the Spring container.

The aspect that interprets `@Transactional` annotations is the `AnnotationTransactionAspect`. When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

A `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any *public* operation in the class.

A `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods with `public`, `protected`, and default visibility may all be annotated. Annotating `protected` and default visibility methods directly is the only way to get transaction

demarcation for the execution of such methods.

For AspectJ programmers that want to use the Spring configuration and transaction management support but don't want to (or cannot) use annotations, `spring-aspects.jar` also contains `abstract` aspects you can extend to provide your own pointcut definitions. See the sources for the `AbstractBeanConfigurerAspect` and `AbstractTransactionAspect` aspects for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototype bean definitions that match the fully-qualified class names:

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

  public DomainObjectConfiguration() {
    setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
  }

  // the creation of a new bean (any object in the domain model)
  protected pointcut beanCreation(Object beanInstance) :
    initialization(new(..)) &&
    SystemArchitecture.inDomainModel() &&
    this(beanInstance);

}
```

9.8.3 Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (the '`per-xxx`' clause) used by the aspect.

The majority of AspectJ aspects are *singleton* aspects. Configuration of these aspects is very easy: simply create a bean definition referencing the aspect type as normal, and include the bean attribute `'factory-method="aspectOf"'`. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
      factory-method="aspectOf">
  <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

Non-singleton aspects are harder to configure: however it is possible to do so by creating prototype bean definitions and using the `@Configurable` support from `spring-aspects.jar` to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are all configured using Spring, then you will need to tell the Spring AOP @AspectJ autoproxying support which exact subset of the @AspectJ aspects defined in the configuration should be used for autoproxying. You can do this by using one or more `<include/>` elements inside

the `<aop:aspectj-autoproxy/>` declaration. Each `<include/>` element specifies a name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

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

Do not be misled by the name of the `<aop:aspectj-autoproxy/>` element: using it will result in the creation of *Spring AOP proxies*. The @AspectJ style of aspect declaration is just being used here, but the AspectJ runtime is *not* involved.

9.8.4 Load-time weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application's class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework: this section is not an introduction to LTW though. For full details on the specifics of LTW and configuring LTW with just AspectJ (with Spring not being involved at all), see the LTW section of the AspectJ Development Environment Guide.

The value-add that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is thus a JVM-wide setting, which may be fine in some situations, but often is a little too coarse. Spring-enabled LTW enables you to switch on LTW on a *per-*`ClassLoader` basis, which obviously is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, in certain environments, this support enables load-time weaving *without making any modifications to the application server's launch script* that will be needed to add `-javaagent:path/to/aspectjweaver.jar` or (as we describe later in this section) `-javaagent:path/to/org.springframework.instrument-{version}.jar` (previously named `spring-agent.jar`). Developers simply modify one or more files that form the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW using Spring, followed by detailed specifics about elements introduced in the following example. For a complete example, please see the Petclinic sample application.

A first example

Let us assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, what we are going to do is switch on a simple profiling aspect that will enable us to very quickly get some performance metrics, so that we can then apply a finer-grained profiling tool to that specific area immediately afterwards.

The example presented here uses XML style configuration, it is also possible to configure and use @AspectJ with Java Configuration. Specifically the `@EnableLoadTimeWeaving` annotation can be used as an alternative to `<context:load-time-weaver/>` (see below for details).

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the @AspectJ-style of aspect declaration.

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

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

We will also need to create an '`META-INF/aop.xml`' file, to inform the AspectJ weaver that we want to weave our `ProfilingAspect` into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called '`META-INF/aop.xml`' is standard AspectJ.

```
<!DOCTYPE aspectj PUBLIC
        "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>

        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>

    </weaver>
```

```
    <aspects>

        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>

    </aspects>

  </aspectj>
```

Now to the Spring-specific portion of the configuration. We need to configure a `LoadTimeWeaver` (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more '`META-INF/aop.xml`' files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
          class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>

</beans>
```

Now that all the required artifacts are in place - the aspect, the '`META-INF/aop.xml`' file, and the Spring configuration -, let us create a simple driver class with a `main(..)` method to demonstrate the LTW in action.

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml", Main.class);
```

```
        EntitlementCalculationService entitlementCalculationService
            = (EntitlementCalculationService) ctx.getBean("entitlementCalculationService");

        // the profiling aspect is 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

There is one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per-`ClassLoader` basis with Spring, and this is true. However, just for this example, we are going to use a Java agent (supplied with Spring) to switch on the LTW. This is the command line we will use to run the above `Main` class:

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

The '`-javaagent`' is a Java 5+ flag for specifying and enabling agents to instrument programs running on the JVM. The Spring Framework ships with such an agent, the `InstrumentationSavingAgent`, which is packaged in the `spring-instrument.jar` that was supplied as the value of the `-javaagent` argument in the above example.

The output from the execution of the `Main` program will look something like that below. (I have introduced a `Thread.sleep(..)` statement into the `calculateEntitlement()` implementation so that the profiler actually captures something other than 0 milliseconds - the `01234` milliseconds is *not* an overhead introduced by the AOP :))

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
------ ----- ----------------------------
ms     %     Task name
------ ----- ----------------------------
01234  100%  calculateEntitlement
```

Since this LTW is effected using full-blown AspectJ, we are not just limited to advising Spring beans; the following slight variation on the `Main` program will yield the same result.

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

        new ClassPathXmlApplicationContext("beans.xml", Main.class);
```

```
        EntitlementCalculationService entitlementCalculationService =
            new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

Notice how in the above program we are simply bootstrapping the Spring container, and then creating a new instance of the `StubEntitlementCalculationService` totally outside the context of Spring... the profiling advice still gets woven in.

The example admittedly is simplistic... however the basics of the LTW support in Spring have all been introduced in the above example, and the rest of this section will explain the 'why' behind each bit of configuration and usage in detail.

The `ProfilingAspect` used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development (of course), and then quite easily exclude from builds of the application being deployed into UAT or production.

Aspects

The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. The latter option is of course only an option if you are using Java 5+, but it does mean that your aspects are then both valid AspectJ *and* Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

'`META-INF/aop.xml`'

The AspectJ LTW infrastructure is configured using one or more '`META-INF/aop.xml`' files, that are on the Java classpath (either directly, or more typically in jar files).

The structure and contents of this file is detailed in the main AspectJ reference documentation, and the interested reader is referred to that resource. (I appreciate that this section is brief, but the '`aop.xml`' file is 100% AspectJ - there is no Spring-specific information or semantics that apply to it, and so there is no extra value that I can contribute either as a result), so rather than rehash the quite satisfactory section that the AspectJ developers wrote, I am just directing you there.)

Required libraries (JARS)

At a minimum you will need the following libraries to use the Spring Framework's support for AspectJ LTW:

1. `spring-aop.jar` (version 2.5 or later, plus all mandatory dependencies)
2. `aspectjweaver.jar` (version 1.6.8 or later)

If you are using the Spring-provided agent to enable instrumentation, you will also need:

1. `spring-instrument.jar`

Spring configuration

The key component in Spring's LTW support is the `LoadTimeWeaver` interface (in the `org.springframework.instrument.classloading` package), and the numerous implementations of it that ship with the Spring distribution. A `LoadTimeWeaver` is responsible for adding one or more `java.lang.instrument.ClassFileTransformers` to a `ClassLoader` at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.

If you are unfamiliar with the idea of runtime class file transformation, you are encouraged to read the Javadoc API documentation for the `java.lang.instrument` package before continuing. This is not a huge chore because there is - rather annoyingly - precious little documentation there... the key interfaces and classes will at least be laid out in front of you for reference as you read through this section.

Configuring a `LoadTimeWeaver` for a particular `ApplicationContext` can be as easy as adding one line. (Please note that you almost certainly will need to be using an `ApplicationContext` as your Spring container - typically a `BeanFactory` will not be enough because the LTW support makes use of `BeanFactoryPostProcessors`.)

To enable the Spring Framework's LTW support, you need to configure a `LoadTimeWeaver`, which typically is done using the `@EnableLoadTimeWeaving` annotation.

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {

}
```

Alternatively, if you prefer XML based configuration, use the `<context:load-time-weaver/>` element. Note that the element is defined in the '`context`' namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver/>

</beans>
```

The above configuration will define and register a number of LTW-specific infrastructure beans for you automatically, such as a `LoadTimeWeaver` and an `AspectJWeavingEnabler`. The default `LoadTimeWeaver` is the `DefaultContextLoadTimeWeaver` class, which attempts to decorate an automatically detected

`LoadTimeWeaver`: the exact type of `LoadTimeWeaver` that will be 'automatically detected' is dependent upon your runtime environment (summarized in the following table).

Table 9.1. `DefaultContextLoadTimeWeaver` `LoadTimeWeavers`

Runtime Environment	`LoadTimeWeaver` implementation
Running in BEA's Weblogic 10	`WebLogicLoadTimeWeaver`
Running in IBM WebSphere Application Server 7	`WebSphereLoadTimeWeaver`
Running in Oracle's OC4J	`OC4JLoadTimeWeaver`
Running in GlassFish	`GlassFishLoadTimeWeaver`
Running in JBoss AS	`JBossLoadTimeWeaver`
JVM started with Spring `InstrumentationSavingAgent` *(java -javaagent:path/to/spring-instrument.jar)*	`InstrumentationLoadTimeWeaver`
Fallback, expecting the underlying ClassLoader to follow common conventions (e.g. applicable to `TomcatInstrumentableClassLoader` and Resin)	`ReflectiveLoadTimeWeaver`

Note that these are just the `LoadTimeWeavers` that are autodetected when using the `DefaultContextLoadTimeWeaver`: it is of course possible to specify exactly which `LoadTimeWeaver` implementation that you wish to use.

To specify a specific `LoadTimeWeaver` with Java configuration implement the `LoadTimeWeavingConfigurer` interface and override the `getLoadTimeWeaver()` method:

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {
    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}
```

If you are using XML based configuration you can specify the fully-qualified classname as the value of the '`weaver-class`' attribute on the `<context:load-time-weaver/>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver
            weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>
```

The `LoadTimeWeaver` that is defined and registered by the configuration can be later retrieved from the Spring container using the well-known name '`loadTimeWeaver`'. Remember that the `LoadTimeWeaver` exists just as a mechanism for Spring's LTW infrastructure to add one or more `ClassFileTransformers`. The actual `ClassFileTransformer` that does the LTW is the `ClassPreProcessorAgentAdapter` (from the `org.aspectj.weaver.loadtime` package) class. See the class-level Javadoc for the `ClassPreProcessorAgentAdapter` class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this section.

There is one final attribute of the configuration left to discuss: the '`aspectjWeaving`' attribute (or '`aspectj-weaving`' if you are using XML). This is a simple attribute that controls whether LTW is enabled or not, it is as simple as that. It accepts one of three possible values, summarized below, with the default value if the attribute is not present being '`autodetect`'

Table 9.2. AspectJ weaving attribute values

Annotation Value	XML Value	Explanation
`ENABLED`	`on`	AspectJ weaving is on, and aspects will be woven at load-time as appropriate.
`DISABLED`	`off`	LTW is off... no aspect will be woven at load-time.
`AUTODETECT`	`autodetect`	If the Spring LTW infrastructure can find at least one '`META-INF/aop.xml`' file, then AspectJ weaving is on, else it is off. This is the default value.

Environment-specific configuration

This last section contains any additional settings and configuration that you will need when using Spring's LTW support in environments such as application servers and web containers.

Tomcat

Apache Tomcat's default class loader does not support class transformation which is why Spring provides an enhanced implementation that addresses this need. Named `TomcatInstrumentableClassLoader`, the loader works on Tomcat 5.0 and above and can be registered individually for *each* web application as follows:

- Tomcat 6.0.x or higher
  1. Copy `org.springframework.instrument.tomcat.jar` into *$CATALINA_HOME*/lib, where *$CATALINA_HOME* represents the root of the Tomcat installation)
  2. Instruct Tomcat to use the custom class loader (instead of the default) by editing the web application context file:

     ```
     <Context path="/myWebApp" docBase="/my/webApp/location">
         <Loader
             loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
     </Context>
     ```

     Apache Tomcat 6.0.x (similar to 5.0.x/5.5.x) series supports several context locations:
     - server configuration file - *$CATALINA_HOME/conf/server.xml*
     - default context configuration - *$CATALINA_HOME/conf/context.xml* - that affects all deployed web applications
     - per-web application configuration which can be deployed either on the server-side at *$CATALINA_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml* or embedded inside the web-app archive at *META-INF/context.xml*

     For efficiency, the embedded per-web-app configuration style is recommended because it will impact only applications that use the custom class loader and does not require any changes to the server configuration. See the Tomcat 6.0.x documentation for more details about available context locations.
- Tomcat 5.0.x/5.5.x
  1. Copy `org.springframework.instrument.tomcat.jar` into *$CATALINA_HOME*/server/lib, where *$CATALINA_HOME* represents the root of the Tomcat installation.
  2. Instruct Tomcat to use the custom class loader instead of the default one by editing the web application context file:

     ```
     <Context path="/myWebApp" docBase="/my/webApp/location">
         <Loader
             loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
     </Context>
     ```

     Tomcat 5.0.x and 5.5.x series supports several context locations:
     - server configuration file - *$CATALINA_HOME/conf/server.xml*
     - default context configuration - *$CATALINA_HOME/conf/context.xml* - that affects all deployed web applications
     - per-web application configuration which can be deployed either on the server-side at *$CATALINA_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml* or embedded inside the web-app archive at *META-INF/context.xml*

     For efficiency, the embedded web-app configuration style is recommended recommended because it will impact only applications that use the class loader. See the Tomcat 5.x documentation for more details about available context locations.

Tomcat versions prior to 5.5.20 contained a bug in the XML configuration parsing that prevented usage of the `Loader` tag inside *server.xml* configuration, regardless of whether a class loader is specified or whether it is the official or a custom one. See Tomcat's bugzilla for more details.
In Tomcat 5.5.x, versions 5.5.20 or later, you should set *useSystemClassLoaderAsParent* to `false` to fix this problem:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader
        loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
            useSystemClassLoaderAsParent="false"/>
</Context>
```

This setting is not needed on Tomcat 6 or higher.

Alternatively, consider the use of the Spring-provided generic VM agent, to be specified in Tomcat's launch script (see above). This will make instrumentation available to all deployed web applications, no matter what ClassLoader they happen to run on.

WebLogic, WebSphere, OC4J, Resin, GlassFish, JBoss

Recent versions of BEA WebLogic (version 10 and above), IBM WebSphere Application Server (version 7 and above), Oracle Containers for Java EE (OC4J 10.1.3.1 and above), Resin (3.1 and above) and JBoss (5.x or above) provide a ClassLoader that is capable of local instrumentation. Spring's native LTW leverages such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating load-time weaving as described earlier. Specifically, you do *not* need to modify the launch script to add `-javaagent:path/to/spring-instrument.jar`.

Note that GlassFish instrumentation-capable ClassLoader is available only in its EAR environment. For GlassFish web applications, follow the Tomcat setup instructions as outlined above.

Note that on JBoss 6.x, the app server scanning needs to be disabled to prevent it from loading the classes before the application actually starts. A quick workaround is to add to your artifact a file named `WEB-INF/jboss-scanning.xml` with the following content:

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

Generic Java applications

When class instrumentation is required in environments that do not support or are not supported by the existing `LoadTimeWeaver` implementations, a JDK agent can be the only solution. For such cases, Spring provides `InstrumentationLoadTimeWeaver`, which requires a Spring-specific (but very general) VM agent, `org.springframework.instrument-{version}.jar` (previously named `spring-agent.jar`).

To use it, you must start the virtual machine with the Spring agent, by supplying the following JVM options:

```
-javaagent:/path/to/org.springframework.instrument-{version}.jar
```

Note that this requires modification of the VM launch script which may prevent you from using this in application server environments (depending on your operation policies). Additionally, the JDK agent will instrument the *entire* VM which can prove expensive.

For performance reasons, it is recommended to use this configuration only if your target environment (such as Jetty) does not have (or does not support) a dedicated LTW.

9.9 Further Resources

More information on AspectJ can be found on the AspectJ website.

The book *Eclipse AspectJ* by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The book *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) comes highly recommended; the focus of the book is on AspectJ, but a lot of general AOP themes are explored (in some depth).

Prev | Up | Next
8. Spring Expression Language (SpEL) | Home | 10. Spring AOP APIs