JBoss AOP - Aspect-Oriented Framework for Java

JBoss AOP Reference Documentation

2.0.0

Table of Contents

Preface

Aspect-Oriented Programming (AOP) is a new paradigm that allows you to organize and layer your software applications in ways that are impossible with traditional object-oriented approaches. Aspects allow you to transparently glue functionality together so that you can have a more layered design. AOP allows you to intercept any event in a Java program and trigger functionality based on those events. Mixins allow you to introduce multiple inheritance to Java so that you can provide APIs for your aspects. Combined with JDK 5.0 annotations, it allows you to extend the Java language with new syntax.

JBoss AOP is a 100% Pure Java aspected oriented framework usable in any programming environment or tightly integrated with our application server.

This document is meant to be a boring reference guide. It focuses solely on syntax and APIs and worries less about providing real world examples. Please see our "User Guide: The Case for Aspects" document for a more interesting discussion on the use of aspects.

If you have questions, use the user forum linked on the JBoss AOP website. We also provide tracking links for tracking bug reports and feature requests. If you are interested in the development of JBoss AOP, post a message on the forum. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

Commercial development support, production support and training for JBoss AOP is available through JBoss Inc. (see http://www.jboss.org/). JBoss AOP is a project of the JBoss Professional Open Source product suite.

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
   long line that \
   does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

1

Terms

1.1. Overview

The section defines some basic terms that will be used throughout this guide.

Joinpoint
: A joinpoint is any point in your java program. The call of a method. The execution of a constructor the access of a field. All these are joinpoints. You could also think of a joinpoint as a particular Java event. Where an event is a method call, constructor call, field access etc...

Invocation
: An Invocation is a JBoss AOP class that encapsulates what a joinpiont is at runtime. It could contain information like which method is being called, the arguments of the method, etc...

Advice
: An advice is a method that is called when a particular joinpoint is executed, i.e., the behavior that is triggered when a method is called. It could also be thought of as the code that does the interception. Another analogy is that an advice is an "event handler".

Pointcut
: Pointcuts are AOP's expression language. Just as a regular expression matches strings, a pointcut expression matches a particular joinpoint.

Introductions
: An introduction modifies the type and structure of a Java class. It can be used to force an existing class to implement an interface or to add an annotation to anything.

Aspect
: An Aspect is a plain Java class that encapsulates any number of advices, pointcut definitions, mixins, or any other JBoss AOP construct.

Interceptor
: An interceptor is an Aspect with only one advice named "invoke". It is a specific interface that you can implement if you want your code to be checked by forcing your class to implement an interface. It also will be portable and can be reused in other JBoss environments like EJBs and JMX MBeans.

2

Chapter 2

2.1. Overview

JBoss AOP is a 100% pure Java framework. All your AOP constructs are defined as pure Java classes and bound to your application code via XML or by annotations. This Chapter walks through implementing aspects.

2.2. Aspect Class

The Aspect Class is a plain Java class that can define zero or more advices, pointcuts, and/or mixins.

```
public class Aspect
 {
   public Object trace(Invocation invocation) throws Throwable {
      try {
         System.out.println("Entering anything");
         return invocation.invokeNext(); // proceed to next advice or actual call
      } finally {
         System.out.println("Leaving anything");
      }
   }
}
```

The example above is of an advice `trace` that traces calls to any type of joinpoint. Notice that `Invocation` objects are the runtime encapsulation of joinpoints. The method `invocation.invokeNext()` is used to drive the advice chain. It either calls the next advice in the chain, or does the actual method or constructor invocation.

2.3. Advice Methods

For basic interception, any method that follows the form:

```
Object methodName(Invocation object) throws Throwable
```

can be an advice. The `Invocation.invokeNext()` method must be called by the advice code or no other advice will be called, and the actual method, field, or constructor invocation will not happen.

JBoss AOP provides five types of advice: before, around, after and after-throwing. The advice sginature above is the default one for an around advice. Advices types, signature rules and overloading will be covered in Chapter 4.

2.4. Interceptors

Interceptors are a special type of aspect that contains only one advice. This advice has it signature defined by an interface, `org.jboss.aop.advice.Interceptor`:

```
public interface Interceptor
{
   public String getName();

   public Object invoke(Invocation invocation) throws Throwable;
}
```

The method `invoke(Invocation)` is the unique advice contained in an interceptor. The method `getName()` is used for identification in the JBoss AOP framework. So, this method must return a name that is unique in the whole system (per instance?).

2.5. Resolving Annotations

JBoss AOP provides an abstraction for resolving JDK 5.0 annotations (and JDK 1.4 annotations if you use our Annotation Compiler). In future versions of JBoss AOP, there will be a way to override annotation values on a per thread basis, or via XML overrides, or even provide VM and cluster wide defaults for annotation values. Also if you want to write a truly generic advice that takes the base Invocation type, you can still get the annotation value of the method, constructor, or field you're invoking on by calling this method:

```
Object resolveAnnotation(Class annotation);
```

That's just resolving for resolving member annotations. If your aspect needs to resolve class level annotations then this method should be called:

```
Object resolveClassAnnotation(Class annotation)
```

2.6. Metadata

2.6.1. Resolving XML Metadata

Untyped metadata can be defined within XML files and bound to org.jboss.aop.metadata.SimpleMetaData structures. This XML data can be attached per method, field, class, and constructor. To resolve this type of metadata, the Invocation object provides a method to abstract out where the metadata comes from.

```
Object getMetaData(Object group, Object attr)
```

When this method is called, the invocation will look for metadata in this order:

1. First it looks in the Invocation's metadata (`SimpleMetaData getMetaData()`)
2. Next it looks in `org.jboss.aop.metadata.ThreadMetaData.instance()`. ThreadMetaData allows you to override metadata for the whole thread. The metadata is managed by a ThreadLocal. ThreadMetaData is used by every single invocation object at runtime.
3. Next it looks in either org.jboss.aop.Advisor.getMethodMetaData(), Advisor.getConstructorMetaData(), or Advisor.getFieldMetaData() depending on the invocation type.

4. Next it looks in either Advisor.getDefaultMetaData().

2.6.2. Attaching Metadata

You can attach untyped metadata to the invocation object, or even to the response. This allows advices to pass contextual data to one another in the incoming invocation or outgoing response for instance if you had advices running on a remote client that wanted to pass contextual data to server-side aspects. This method on invocation gets you access to a `org.jboss.aop.metadata.SimpleMetaData` instance so that you can attach or read data.

```
SimpleMetaData getMetaData()
```

`SimpleMetaData` has three types of metadata, AS_IS, MARSHALLED, and TRANSIENT. This allows you to specify whether or not metadata is marshalled across the wire. TRANSIENT says, attached metadata should not be sent across the wire. MARSHALLED is for classloader sensitive contextual data. AS_IS doesn't care about classloaders. Read the Javadocs for more information.

To piggyback and read metadata on the invocation response, two methods are provided. One to attach data one to read data.

```
Object getResponseAttachment(Object key);
void addResponseAttachment(Object key, Object value);
```

2.7. Mixin Definition

Mixins are a type of introduction in which you can do something like C++ multiple inheritance and force an existing Java class to implement a particular interface and the implementation of that particular interface is encapsulated into a particular class called a mixin.

Mixin classes have no restrictions other than they must implement the interfaces that you are introducing.

2.8. Dynamic CFlow

Dynamic CFlows allow you to define code that will be executed that must be resolved true to trigger positive on a cflow test on an advice binding. (See <cflow-stack> for more information). The test happens dynamically at runtime and when combined with a pointcut expression allows you to do runtime checks on whether a advice binding should run or not. To implement a dynamic CFlow you just have to implement the simple org.jboss.aop.pointcut.DynamicCFlow interface. You can then use it within cflow expressions. (See XML or Annotations)

```
public interface DynamicCFlow
{
   boolean shouldExecute(Invocation invocation);
}
```

3

Joinpoint and Pointcut Expressions

The pointcut language is a tool that allows joinpoint matching. A pointcut expression determines in which joinpoint executions of the base system an advice should be invoked.

In this Chapter, we will explore the syntax of pointcut expressions.

We will also see the API used to represent a matched joinpoint during advice execution, and how this relates to pointcut expression constructs.

3.1. Wildcards

There are two types of wildcards you can use within pointcut expressions

- `*` Is a regular wildcard that matches zero or more characters. It can be used within any type expression, field, or method name, but not in an annotation expression
- `..` Is used to specify any number of parameters in an constructor or method expression. `..` following a package-name is used to specify all classes from within a given package ut not within sub-packages. e.g `org.acme..` matches `org.acme.Foo` and `org.acme.Bar`, but it does not match `org.acme.sub.SubFoo`.

3.2. Type Patterns

Type patterns are defined by an annotation or by fully qualified class name. Annotation expressions are not allowed to have wildcards within them, but class expressions are.

- `org.acme.SomeClass` matches that class.
- `org.acme.*` will match `org.acme.SomeClass` as well as `org.acme.SomeClass.SomeInnerClass`
- `@javax.ejb.Entity` will match any class tagged as such.
- `String` or `Object` are illegal. You must specify the fully qualified name of every java class. Even those under the `java.lang` package.

To reference all subtypes of a certain class (or implementors of an interface), the `$instanceof{}` expression can be used. Wildcards and annotations may also be used within `$instanceof{}` expressions.

```
        $instanceof{org.acme.SomeInterface}
        $instanceof{@org.acme.SomeAnnotation}
```

```
        $instanceof{org.acme.interfaces.*}
```

are all allowed.

For very complex type expressions, the Typedef construct can be used. To reference a Typedef within a class expression `$typedef{id}` is used.

3.3. Method Patterns

```
public void org.acme.SomeClass->methodName(java.lang.String)
```

The attributes(`public, static, private`) of the method are optional. If the attribute is left out then any attribute is assumed. Attributes accept the `!` modifier for negation.

```
public !static void org.acme.SomeClass->*(..)
```

`$instanceof{}` can be used in place of the class name.

```
void $instanceof{org.acme.SomeInterface}->methodName(java.lang.String)
```

To pick out all `toString()` methods of all classes within the `org.acme` package, we can use `org.acme..` in place of the class name.

```
java.lang.String org.acme..->toString()
```

To only match methods from a given interface you can use the `$implements{}` or `$implementing{}` keywords in place of the method name. `$implements{}` only matches methods from the exact interface(s) given, while `$implementing{}` matches methods from the interface(s) given AND their super interfaces.

```
void $instanceof{org.acme.IfA}->$implements(org.acme.IfA)
```

```
void $instanceof{org.acme.IfB}->$implementing(org.acme.IfA, org.acme.IfB)
```

Annotations can be used in place of the class name. The below example matches any methodName() of a tagged @javax.ejb.Entity class.

```
void @javax.ejb.Entity->methodName(java.lang.String)
```

Annotations can be also be used in place of the method name. The below examples matches any method tagged as @javax.ejb.Tx.

```
* *->@javax.ejb.Tx(..)
```

In addition you can use typedefs, `$instanceof{}`, annotations and wildcards for method parameters and return types. The following matches all methods called `loadEntity` that return a class annotated with @javax.ejb.Entity, that takes a class (or a class whose superclass/interface is) annotated as @org.acme.Ann and any class that matches `java.*.String` (such as java.lang.String).

```
@javax.ejb.Entity *->loadEntity($instanceof{@org.acme.Ann}, java.*.String)
```

You can also include an optional throws clause in the pointcut expression:

```
public void org.acme.SomeClass->methodName(java.lang.String) \
      throws org.acme.SomeException, java.lang.Exception
```

If any exceptions are present in the pointcut expression they must be present in the throws clause of the methods to be matched.

3.4. Constructor Patterns

```
public org.acme.SomeClass->new(java.lang.String)
```

Constructor expressions are made up of the fully qualified classname and the `new` keyword The attributes(`public, static, private`) of the method are optional. If the attribute is left out then any attribute is assumed. Attributes accept the `!` modifier for negation.

```
!public org.acme.SomeClass->new(..)
```

`$instanceof{}` can be used in the class name.

```
$instanceof{org.acme.SomeInterface}->new(..)
```

To pick out all no-args constructors of all classes within the `org.acme` package, we can use `org.acme..` in place of the class name.

```
org.acme..->new()
```

Annotations can be used in place of the class name. The below example matches any constructor of a tagged @javax.ejb.Entity class.

```
@javax.ejb.Entity->new(..)
```

Annotations can be also be used in place of the `new` keyword. The below examples matches any constructor tagged as `@javax.ejb.MethodPermission`.

```
*->@javax.ejb.MethodPermission(..)
```

In addition, just as for methods you can use typedefs, `$instanceof{}`, annotations and wildcards for constructor parameters. The following matches all constructors that take a class annotated as @org.acme.Ann and any class that matches `java.*.String` (such as java.lang.String).

```
*->new(@org.acme.Ann, java.*.String)
```

You can also include an optional throws clause in the pointcut expression:

```
public void org.acme.SomeClass->new(java.lang.String) \
      throws org.acme.SomeException, java.lang.Exception
```

If any exceptions are present in the pointcut expression they must be present in the throws clause of the construct-

ors to be matched.

3.5. Field Patterns

```
public java.lang.String org.acme.SomeClass->fieldname
```

Constructor expressions are made up of the type, the fully qualified classname where the field resides and the field's name. The attributes(`public, static, private`) of the field are optional. If the attribute is left out then any attribute is assumed. Attributes accept the `!` modifier for negation.

```
!public java.lang.String org.acme.SomeClass->*
```

`$instanceof{}` can be used in the class name. The below expression matches any field of any type or subtype of `org.acme.SomeInterface`

```
* $instanceof{org.acme.SomeInterface}->*
```

Annotations can be used in place of the class name. The below example matches any field where the type class is tagged with `@javax.ejb.Entity`.

```
* @javax.ejb.Entity->*
```

Annotations can be also be used in place of the field name. The below examples matches any field tagged as `@org.jboss.Injected.`

```
* *->@org.jboss.Injected
```

In addition, you can use typedefs, `$instanceof{}`, annotations and wildcards for field types. The following matches all fields where the type class has been tagged with @javax.ejb.Entity.

```
@javax.ejb.Entity *->*
```

To pick out all fields annotated with `@org.foo.Transient` within the `org.acme` package, we can use `org.acme..` in place of the class name, and `@org.foo.Transient` in please of the field name

```
* org.acme..->@org.foo.Transient
```

3.6. Pointcuts

Pointcuts use class, field, constructor, and method expressions to specify the actual joinpoint that should be intercepted/watched.

`execution(method or constructor)`

```
execution(public void Foo->method()
execution(public Foo->new())
```

`execution` is used to specify that you want an interception to happen whenever a method or constructor is called. The the first example of matches anytime a method is called, the second matches a constructor. System classes cannot be used within `execution` expressions because it is impossible to instrument them.

`construction(constructor)`

```
construction(public Foo->new())
```

`construction` is used to specify that you want aspects to run within the constructor. The `execution` pointcut requires that any code that calls new() must be instrumented by the compiler. With `construction` the aspects are weaved right within the constructor after all the code in the constructor. The aspects are appended to the code of the constructor.

`get (field expression)`

```
get(public int Foo->fieldname)
```

`get` is used to specify that you want an interception to happen when a specific field is accessed for a read.

`set(field expression)`

```
get(public int Foo->fieldname)
```

`set` is used to specify that you want an interception to happen when a specific field is accessed for a write.

`field(field expression)`

```
field(public int Foo->fieldname)
```

`field` is used to specify that you want an interception to happen when a specific field is accessed for a read or a write.

`all(type expression)`

```
all(org.acme.SomeClass)
all(@org.jboss.security.Permission)
```

`all` is used to specify any constructor, method or field of a particular class will be intercepted. If an annotation is used, it matches the member's annotation, not the class's annotation.

`call(method or constructor)`

```
call(public void Foo->method()
call(public Foo->new())
```

`call` is used to specify any constructor or method that you want intercepted. It is different than `execution` in that the interception happens at the caller side of things and the caller information is available within the Invocation object. `call` can be used to intercept System classes because the bytecode weaving happens within the callers bytecode.

`within(type expression)`

```
within(org.acme.SomeClass)
within(@org.jboss.security.Permission)
```

`within` matches any joinpoint (method or constructor call) within any code within a particular type.

`withincode(method or constructor)`

```
withincode(public void Foo->method()
withincode(public Foo->new())
```

`withincode` matches any joinpoint (method or constructor call) within a particular method or constructor.

`has(method or constructor)`

```
has(void *->@org.jboss.security.Permission(..))
has(*->new(java.lang.String))
```

`has` is an additional requirement for matching. If a joinpoint is matched, its class must also have a constructor or method that matches the `has` expression.

`hasfield(field expression)`

```
hasfield(* *->@org.jboss.security.Permission)
hasfield(public java.lang.String *->*)
```

`has` is an additional requirement for matching. If a joinpoint is matched, its class must also have a field that matches the `hasfield` expression.

3.7. Pointcut Composition

Pointcuts can be composed into boolean expressions.

- `!` logical not.
- `AND` logical and.
- `OR` logical or.
- Parathesis can be used for grouping expressions.

Here's some examples.

```
call(void Foo->someMethod()) AND withincode(void Bar->caller())
execution(* *->@SomeAnnotation(..)) OR field(* *->@SomeAnnotation)
```

3.8. Pointcut References

Pointcuts can be named in XML or annotation bindings (See in later chapters). They can be referenced directly within a pointcut expression.

```
some.named.pointcut OR call(void Foo->someMethod())
```

3.9. Typedef Expressions

Sometimes, when writing pointcuts, you want to specify a really complex type they may or may not have boolean logic associated with it. You can group these complex type definitions into a JBoss AOP `Typedef` either in XML or as an annotation (See later in this document). Typedef expressions can also be used within `introduction` expressions. Typedef expressions can be made up of `has, hasfield, and class` expressions. `class` takes a fully qualified class name, or an `$instanceof{}` expression.

```
class(org.pkg.*) OR has(* *->@Tx(..)) AND !class($instanceof{org.foo.Bar})
```

3.10. Joinpoints

After getting acquainted with all pointcut constructs, let's see how this reflects on the API available to advices during their execution.

3.10.1. Joinpoint Beans

JBoss AOP provides JoinPoint Beans, so that an advice can access all information regarding a joinpoint during its execution. This information consists of context values, explained in the next subsection, and of reflection objects (`java.lang.reflection`). The reflection objects describe the joinpoint being intercepted like a `java.lang.Method` for a method execution joinpoint).

There are two groups of beans. The first one is the `Invocation` beans group. All classes of this group are subclasses of `org.jboss.aop.joinpoint.Invocation`. The `Invocation` class was presented in Chapter 2 as a runtime encapsulation of a joinpoint. An `Invocation` object also contains an interceptor chain, where all advices and interceptors that intercept the joinpoint are stored. Invocation beans provide the `invokeNext()` method, responsible for proceeding execution to the next advice in the interceptor chain (if there is an advice that has not started execution yet) or to the joinpoint itself (if all advices contained in the interceptor chain have already started running). We will see more on this in the next chapter.

The other group of beans contains only information regarding the joinpoint itself, and are called the `JoinPointBean` group. All beans of this group are defined in interfaces, with `org.jboss.joinpoint.JoinPointBean` being their common superinterface.

The `Invocation` objects are available only to around advices. All other types of advices can use the `JoinPointBean` types to access joinpoint specific data.

In both groups there is a specific type for each joinpoint type. The type of bean corresponding to each joinpoint type can be seen in Table 3.1. All beans are in the package `org.jboss.aop.joinpoint`.

3.10.2. Context Values

According to the type of the joinpoint, there are specific context values available.

The context values are:

- return value: joinpoints like a constructor execution or a non-void method call, have a return value.
- arguments: the arguments of a constructor or method execution joinpoint are the arguments received by the constructor or method. Similarly, the arguments of a call are the arguments received by the method or constructor being called.
- target: the target object of a joinpoint varies according to the joinpoint type. For method executions and calls, it refers to the object whose method is being executed (available only on non-static methods). For field reads and writes, it refers to the object that contains that field.
- caller: the caller object is available only on call joinpoints, and it refers to the object whose method or constructor is performing the call (notice the caller object is not available if the call is inside a static method).

Table 3.1 shows what context values may be available depending on the joinpoint type.

Table 3.1. Joinpoint Types Table

Joinpoint	Pointcut Construct	Bean		ContextValues			
		Invocation	JoinpointBean	Target	Caller	Arguments	Return Value
field read	`read`, `field`, `all`	`FieldReadInvocation`	`FieldAccess`	Yes	No	No	Yes
field write	`write`, `field`, `all`	`FieldWriteInvocation`	`FieldAccess`	Yes	No	Yes	No
method execution	`execution`, `all`	`MethodInvocation`	`MethodExecution`	Yes	No	Yes	Yes
constructor execution	`execution`	`ConstructorInvocation`	`ConstructorExecution`	No	No	Yes	Yes
construction	`construction`	`ConstructionInvocation`	`ConstructorExecution`	Yes	No	Yes	No
method call	`call`, `within`, `withincode`	`CallerInvocation`, `MethodCalledByConstructorInvocation`, `MethodCalledByMethodInvocation`	`MethodCall`, `MethodCallByConstructor`, `MethodCallByMethod`	Yes	Yes	Yes	Yes
constructor call	`call`, `within`, `withincode`	`CallerInvocation`, `ConstructorCalledByCon-`	`ConstructorCall`, `ConstructorCallByCon-`	Yes	Yes	Yes	Yes

Joinpoint	Pointcut Construct	Bean		ContextValues			
		Invocation	JoinpointBean	Target	Caller	Arguments	Return Value
		`structorInvocation`, `ConstructorCalledByMethodInvocation`	`structor`, `ConstructorCallByMethod`				

4

Advices

Advices are aspect methods that are invoked during specific joinpoint executions.

JBoss AOP provides four types of advice.

The default one is the around advice, and it can be used on all execution modes. This advice wraps the joinpoint, in a way that it replaces the joinpoint execution in the base system, and is responsible for proceeding execution to the joinpoint.

Besides around advices, you can write advices that, instead of wrapping the joinpoint, are executed before or after it. In this category, JBoss AOP provides before, after, after-throwing and finally advices. These advices are available only when using the generated advisor mode (this is the default mode in JBoss AOP, to learn how to select another weaving mode, refer to Chapter X).

The next sections will explain in detail the binding and signature rules for JBoss AOP advices.

4.1. Around Advices

An around advice can follow this template:

```
public Object [advice name]([Invocation] invocation) throws Throwable
{
   try{
      // do something before joinpoint execution
      ...
      // execute the joinpoint and get its return value
      Object returnValue = invocation.invokeNext();
      // do something after joinpoint has executed successfully ...
      // return a value
      return returnValue;
   }
   catch(Exception e)
   {
      //handle any exceptions arising from calling the joinpoint
      throw e;
   }
   finally
   {
      //Take some action once the joinpoint has completed successfully or not
   }
}
```

In the template above, *Invocation* refers to one of the Invocation beans, and can be the class `org.jboss.aop.joinpoint.Invocation` or one of its subtypes.

Since an around advice wraps a joinpoint, it must proceed execution to the joinpoint itself during its execution.

This can be done by calling the method `invokeNext()` on `invocation`. This method will proceed execution to the next around advice of that joinpoint. At the end of this chain this `invokeNext()` will proceed to the joinpoint itself. The value returned by the around advice will replace the joinpoint return value in the base system.

For example, in the case where there are two around advices bound to a joinpoint, the first around advice will trigger the second around advice by calling `invokeNext()`. The second advice will trigger the joinpoint execution by calling the same method. As a result of the `invokeNext()` execution, the second advice will receive the joinpoint return value. The value returned by this second advice will be received as a result by the first around advice. Finally, the value returned by this advice will replace the joinpoint return value in the base system execution. Normally though, around advices will simply return whatever value the joinpoint returned! This is shown in the preceding template example.

If an around advice wants to completely replace the joinpoint execution, it can skip the call to `invokeNext()`. This will also skip execution of any subsequent around advices in the chain. As a third alternative, the around advice can call the method `invokeTarget()` instead of `invokeNext()`. This method will invoke the target joinpoint directly, skipping any subsequent advices.

The presence of the `Invocation` parameter is optional. If an around advice does not have this parameter, it can replace the call to `invokeNext()` with a call to `org.jboss.aop.joinpoint.CurrentInvocation.proceed()`.

The signature described before is the default around advice signature rule. In addition to it, the around advice signature can also be of this form (only in generated advisor mode):

```
public [return type] [advice name]([annotated parameter],[annotated parameter],...[annotated parameter])
```

This signature is joinpoint dependent. The return type of the advice must be a type assignable to the the return type of the joinpoint to be intercepted (i.e. be the same type; a subclass, if the return type is class; or a subinterface or an implementing class, if the return type is an interface). In case the joinpoint being intercepted does not have a return type, this advice return type must be `void`.

An around advice can have zero or more annotated parameters. The annotated parameters will be covered in detail in Section 4.3.

Finally, JBoss AOP also features a special type of around advice: `Interceptor`. An interceptor class implements `org.jboss.aop.Interceptor`, and is described in Section 2.4.

4.2. Before/After/After-Throwing/Finally Advices

These advices are more lightweight in the JBoss AOP framework, since they do not wrap a joinpoint, avoiding the creation of the `Invocation` objects per joinpoint execution.

Instead of `Invocation` objects, JBoss AOP provides JoinPointBean beans for these advices. As described in Section 3.10.1, these beans contain all information regarding a joinpoint, like an `Invocation` would do. However, since `JoinPointBean` objects are not used on around advice types, they do not provide proceeding methods, like `invokeNext()`. They also do not allow you to attach metadata for a particular invocation.

The rules for before, after, after-throwing and finally advices are quite similar. All of them can have zero or more annotated advice parameters in their signature, which will be described in the next subsection.

4.2.1. Before Advice Signature

A before advice is executed before the joinpoint. The signature for a before advice must be of this form:

```
public void [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])
```

4.2.2. After Advice Signature

Since an after advice is executed after a joinpoint, it can return a value to replace the joinpoint return value in the base system. So, they can follow one of these signatures:

```
public void [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])

public [return type] [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])
```

In the first signature, the after advice does not overwrite the joinpoint return value. On the other hand, when using the second signature, the after advice return value will replace the joinpoint return value. As with around advices, this return type must be assignable to the joinpoint return type.

4.2.3. After-Throwing Advice Signature

The fourth type of advice provided by JBoss AOP is the after-throwing type. This advice is invoked only after the execution of a joinpoint that has thrown a `java.lang.Throwable` or one of its subtypes.

The signature of such an advice is the same as the one for before advices:

```
public void [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])
```

Different from the other advice types, an after-throwing advice has a mandatory annotated parameter. This parameter is the exception thrown by the joinpoint execution, as we will see in the next subsection.

4.2.4. Finally Advice Signature

Lastly, JBoss AOP provides the finally advice type. It is invoked from inside a finally block, after the joinpoint execution.

This advice is the only one that is called after a joinpoint execution in a deterministic way. Calls to after and after-throwing advices take place depending on the joinpoint execution outcome. After advices are not called when the joinpoint execution terminates abruptly with an exception. After-throwing ones, on the other hand, are not called when the joinpoint execution returns normally, since no exception is thrown this time. So, if an advice needs to be run no matter what is the outcome of the joinpoint, it should be a finally advice.

Pretty much as after advices, finally advices can follow one of the signatures below:

```
public void [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])

public [return type] [advice name]([annotated parameter], [annotated parameter],...[annotated parameter])
```

The last signature shows that finally advices can also overwrite the joinpoint execution return value by returning a

value themselves. But notice that this return value will not be received by the base system if an exception has been thrown. However, it is easy to know whether this condition is met, by making use of annotated parameters.

4.3. Annotated Advice Parameters

This section lists the annotated parameters that can be used on JBoss AOP advices (available only in generated advisor execution mode). Table 4.1 lists all annotations and their semantics.

Except for the `@JoinPoint` annotation, used to refer to joinpoint beans, all other annotations are used on parameters that contain joinpoint context values.

Notice that the types of annotated parameters are dependent on the joinpoint being intercepted by the advice.

JBoss AOP will accept any type that is assignable from the type referred by that parameter, as shown in the *Type Assignable From* column of the table below. For example, for a joinpoint whose target is of type `POJO`, the annotated parameter that receives the target must be of `POJO` type, one of `POJO`'s superclasses, or one of the interfaces implemented by `POJO`.

Regarding the type of joinpoint bean parameters, the rules are the same for the default signature of around advices (without annotations). For example, an around advice that intercepts a method execution, can receive either a `MethodInvocation`, or an `Invocation` (the complete list of joinpoint beans and their relationship with joinpoint types was shown in Table 3.1). As already explained, around advices use `Invocation` instances, while the other advices use `JoinPointBean` objects.

Notice also that only one annotated parameter can be mandatory: `@Thrown`. This will be further explained in Section 4.3.1.

Except for `@Arg`, all annotations are single-enforced, i.e., there must be at most only one advice parameter with that annotation per advice.

Table 4.1. Annotated Parameters Table

Annotation	Semantics	Type assignable from	Mandatory	Advice type				
				Before	**Around**	**After**	**After-Throwing**	**Finally**
`@JoinPoint`	JoinPoint bean	Joinpoint invocation type	No	No	Yes	No	No	No
		JoinpointBean interface type	No	Yes	No	Yes	Yes	Yes
`@Target`	Joinpoint target	Joinpoint target type	No	Yes	Yes	Yes	Yes	Yes
`@Caller`	Joinpoint caller	JoinPoint caller type (only for call joinpoints)	No	Yes	Yes	Yes	Yes	Yes

Annotation	Semantics	Type assignable from	Mandatory	Advice type				
				Before	Around	After	After-Throwing	Finally
`@Thrown`	Joinpoint thrown exception	`Throwable`	Yes: - for after-throwing advices - for finally advices only if `@Return` is present No: otherwise	No	No	No	Yes	Yes
`@Return`	Joinpoint return value	JoinPoint return type	No	No	No	Yes	No	Yes
`@Arg`	One of the joinpoint arguments	JoinPoint argument type	No	Yes	Yes	Yes	Yes	Yes
`@Args`	All joinpoint arguments	`Object[]`	No	Yes	Yes	Yes	Yes	Yes

Due to the fact that most of these parameters represent context values, their availability depends on the joinpoint type. If an advice receives as a parameter a context value that is not available during a joinpoint execution, the parameter value will be null. The exception to this rule is @Return. If an advice has this parameter, it will not intercept joinpoints that don't have a return value.

The only exception to this rule is `@Args` on field read joinpoints. Such an advice will be called with an empty arguments array, in that case.

4.3.1. @Thrown annotated parameter

As shown in Table 4.1, the presence of a `@Thrown` annotated parameter can be mandatory depending on the advice type and its parameters.

This annotation is available only for after-throwing and finally advices. For after-throwing advices this parameter is always mandatory:

```
public class Aspect
{
   public void throwing1(@Thrown Throwable thrownException)
   {
      ...
   }
```

```
   public void throwing2()
   {
      ...
   }
}


<aop>
   <aspect class="Aspect"/>
   <bind pointcut="...">
      <throwing aspect="Aspect" name="throwing1"/>
      <throwing aspect="Aspect" name="throwing2"/>
   </bind>
</aop>
```

The advice `throwing1` follows this rule; advice `throwing2`, on the other hand, is invalid, because it does not contain the mandatory `@Thrown` annotated parameter.

For finally advices, the `@Thrown` annotation is compulsory only if a `@Return` annotated parameter is present. This way, a finally advice can identify whether the return value is valid or not. If the `@Thrown` parameter is `null`, it means that the joinpoint returned normally and that the value contained in the `@Return` annotated-parameter is valid. Otherwise, the value contained in `@Return` annotated parameter must be ignored (it will be `null` if the return type is not primitive, `0` if it is a primitive number or `false` if it is boolean). If the finally advice does not receive the joinpoint return value, the use of the `@Thrown` annotated parameter is optional and, as expected, its value will be `null` if the joinpoint being intercepted did not throw an exception. Take a look at the next example:

```
public class Aspect
{
   public void finally1(@Thrown Throwable thrownException)
   {
      ...
   }

   public void finally2()
   {
      ...
   }

   public void finally3(@Return int returnedValue, @Thrown Throwable thrownException)
   {
      if (thrownException == null)

      {

         //We returned normally, the @Return parameter is valid

         int i = returnedValue;

      }

      else

      {

         //An exception happened while invoking the target joinpoint

         //The return value is invalid

      }

    }
```

```
   public void finally4(@Return int returnedValue)
   {
      ...
   }

}


<aop>
   <aspect class="Aspect"/>
   <bind pointcut="execution(public int *->*(..))">
      <finally aspect="Aspect" name="finally1"/>
      <finally aspect="Aspect" name="finally2"/>

      <finally aspect="Aspect" name="finally3"/>

      <finally aspect="Aspect" name="finally4"/>

   </bind>
</aop>
```

This example binds four finally advices to the execution of all public methods that return an int value.

The presence of `@Thrown` is not mandatory in advices `finally1()` and `finally2()`, because they do not have a `@Return` annotated parameter. Hence, both advices are valid. Besides, `finally1()` will receive a non-null exception only when the joinpoint being intercepted throws an exception.

For advice method `finally3()` the presence of a `@Thrown` annotated parameter is mandatory because this advice also has a `@Return` annotated parameter. If an exception happens when invoking the target joinpoint, this advice will receive a non-null `@Thrown` parameter, meaning that the `@Return` annotated parameter is invalid. If the joinpoint completes normally, the `@Thrown` annotated parameter will be `null` and the `@Return` annotated parameter will contain the return value of the target joinpoint.

The `finally4()` advice is invalid, it contains a `@Return` parameter, but has no `@Thrown` annotated parameter. Finally advices require a `@Thrown` parameter if a `@Return` annotated parameter is present.

4.3.2. JoinPoint Arguments

As we saw, an advice can receive the joinpoint arguments as annotated parameters. This can be achieved with the use of two different annotations: `@Arg` and `@Args`.

There is a great difference between these two approaches, though. With `@Arg`, each parameter is equivalent to a single joinpoint parameter. With `@Args`, one single parameter, of type `Object[]`, receives an array containing all joinpoint arguments. This last possibility is more generic than the first one, since it can be used independently of the joinpoint argument types. Plus, it allows changes to the argument values. Any changes performed on the values of this array will be perpetuated to the joinpoint execution. However, the use of `@Args` parameters on a join point interception means the arguments array needs creation. The same happens with the use of `getArguments()` and `setArguments()` methods on `Invocation` classes. So the use of `@Arg` annotated parameters is more lightweight, and should be used whenever there is no need to changing the joinpoint arguments.

When using `@Arg` annotated parameters, the types of these parameters depend on the joinpoint being intercepted. Not all the target joinpoint arguments need to be included as parameters to the advice method. An advice can receive only the argument values that are relevant to its execution.

Given all the possibilities in the usage of `@Arg`, JBoss AOP will match the advice parameters with the joinpoint ones, to infer to which joinpoint argument each advice parameter refers to. This matching process consists of the following steps:

- Each advice parameter will be matched to the first unmatched joinpoint argument that has the same type. This is done in the order that the advice parameters appear in the advice method.
- If any advice parameter is left unmatched, we proceed to an additional step. Each advice parameter will be matched to the first unmatched joinpoint argument that is assignable to it. This is done in the order that the advice parameters appear in the advice method declaration.

To illustrate this mechanism, consider the following scenario:

```
public class POJO
{
    void method(Collection arg0,  List arg1, int arg2, String arg3){}
}


<aop>
   <aspect class="MyAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <before aspect="MyAspect" name="advice"/>
   </bind>
</aop>
```

The example above shows a xml-declared binding. We will use examples with those to illustrate signature concepts from now on. Detailed syntax of xml bindings is shown in Chapter 5.

Class `POJO` is a plain java old object that contains only one method. When calling this method, we want to trigger `MyAspet.advice()` before this method is called. `POJO.method()` receives four distinct arguments, all of them can be available to an advice by using `@Arg` annotated parameters. If `MyAspect.advice()` has the following signature:

```
public class MyAspect
{
   public void advice(@Arg Collection param0, @Arg List param1, @Arg int param2, @Arg String param3)
   {
      ...
   }
}
```

`MyAspect.advice()` parameters will be trivially matched to `POJO.method()` arguments as follows:

```
param0 <- arg0
param1 <- arg1
param2 <- arg2
param3 <- arg3
```

The matching outcome will be the same if `MyAspect.advice()` signature changes slightly in the following manner, since `Collection` is assignable from `List` for `param2`:

```
public class MyAspect
{
   public void advice (@Arg Collection param0, @Arg Collection param1, @Arg int param2, @Arg String param
```

```
   {
      ...
   }
}
```

If `MyAspect.advice()` receives only one parameter, of type `java.lang.Object`:

```
public class MyAspect
{
   public void advice(@Arg Object param0)
   {
      ...
   }
}
```

The parameter matching outcome will be:

```
param0 <- arg0
```

Since there is no joinpoint argument of type `Object`, we proceed to the additional matching step in this case. Because `arg0` is the first unmatched argument that is assignable to `Object`, we assign this argument to `param0`.

Notice that JBoss AOP will match all parameters correctly if we invert the order of parameters:

```
public class MyAspect
{
   public void advice(@Arg int param2, @Arg Collection param0, @Arg String param3, @Arg List param1)
   {
      ...
   }
}
```

If one writes an advice whose unique parameter is a `Collection`, and we want to refer to the second joinpoint argument:

```
public class MyAspect
{
   public void advice (@Arg Collection param1)
   {
      ...
   }
}
```

It will not work as desired. JBoss AOP will assign `arg0` to `param1`:

```
param1 <- arg0
```

In cases like this, it is possible to enforce the correct matching of joinpoint arguments and advice parameters. The annotation `@Arg` has an attribute, index, whose purpose is to define the index of the argument to which that parameter refers.

So, in this case, the `MyAspect.advice()` parameter list below:

```
public class MyAspect
{
   public void advice (@Arg(index=1) Collection param1)
   {
      ...
   }
}
```

Will have the desired matching, which is:

```
param1 <- arg1
```

In the example just shown in this section, `MyAspect.advice()` was a before advice, but the same rules are used for all advices using `@Arg` annotated parameters.

4.4. Overloaded Advices

Method names can be overloaded for interception in different joinpoint scenarios. For instance, let's say you wanted to have a different trace advice for each invocation type. You can specify the same method name `trace` and just overload it with the concrete invocation type.

```
public class AroundAspect
{
   public Object trace(MethodInvocation invocation) throws Throwabl
   {
      try
      {
         System.out.println("Entering method: " + invocation.getMethod()");
         return invocation.invokeNext(); // proceed to next advice or actual call
      }
      finally
      {
         System.out.println("Leaving method: " + invocation.getMethod()");
      }
   }

   public Object trace(ConstructorInvocation invocation) throws Throwable
   {
      try
      {
         System.out.println("Entering constructor: " + invocation.getConstructor()");
         return invocation.invokeNext(); // proceed to next advice or actual call
      }
      finally
      {
         System.out.println("Leaving constructor: " + invocation.getConstructor()");
      }
   }
}
```

As you can see, the selection of the advice method is very dynamic. JBoss AOP will select the most appropriate advice method for each joinpoint interception. For the following setup:

```
class POJO
{
   public POJO(){}
```

```
   public someMethod(){}
}

<aop>
   <aspect class="AroundAspect"/>
   <bind pointcut="all(POJO)">
      <advice aspect="AroundAspect" name="trace"/>
   </bind>
</aop>
```

When calling POJO's constructor:

```
pojo.someMethod();
```

JBoss AOP will call the `trace()` method taking a `ConstructorInvocation`, and when calling:

```
pojo.someMethod();
```

JBoss AOP will call the `trace()` method taking a `MethodInvocation`.

This examples shows that JBoss AOP will select the most appropriate advice method for each joinpoint interception. The capability of selecting overloaded advices is available for all types of advices. And its impact in the system performance is minimal since this selection is done once.

In this section, we will describe every rule JBoss AOP uses to select an advice method when this one is overloaded.

4.4.1. Annotated-parameter Signature

Let's start with the selection of advices when all of them use the annotated-parameter signature. As we will see later, very similar rules are used for selecting advices with the default signature.

The process of selection of advices that follow the annotated-parameter signature depends on the priority of each kind of parameter:

`@JoinPoint` > `@Target` > `@Caller` > `@Throwable` = `@Return` > `@Arg` > `@Args`

This priority is used in two different criteria:

- presence of the annotated parameter
- assignability degree of the annotation parameter

4.4.1.1. Presence priority

This rule is quite simple, it means that an advice that receives only a joinpoint bean (`@JoinPoint`) as its parameter will have a higher priority than another advice that receives all other annotated parameters available (notice we are following the annotation priority order just described).

In other words, the first `OneAspect.after()` advice method will be chosen when calling `POJO.someMethod()` in this example:

```
public class POJO
{
   String someMethod(String s){}
}

<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->someMethod(..))">
      <after aspect="OneAspect" name="after"/>
   </bind>
</aop>


public class OneAspect
{
   public void after(@JoinPoint MethodJoinPoint mjp){} //1
   public String after(@Target POJO pojo, @Return String ret, @Arg String arg0){} //2
}
```

Again in the following example, the first `OneAspect.after()` advice method will be chosen when calling `POJO.someMethod()`. The first `after()` advice method's highest priority parameter is `@Target`, the second advice parameter's highest priority parameter is `@Return`, and `@Target` has a higher priority than `@Return`:

```
public class POJO
{
   String someMethod(String s){}
}

<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->someMethod(..))">
      <after aspect="OneAspect" name="after"/>
   </bind>
</aop>


public class OneAspect
{
   public void after(@Target POJO pojo){} //1
   public String after(@Return String ret, @Arg String arg0){} //2
}
```

In cases where the highest priority annotated parameter of two advice methods is the same, we move on to the next highest priority annotated parameter of both advices. In the following scenario, both `OneAspect.after()` methods have the `@JoinPoint` parameter as the highest priority parameter. The first one has a `@Target` as its second-highest priority parameter while the second one has `@Return` as its second-highest priority parameter. Since `@Target` has a higher priority than `@Return`, the first `OneAspect.after()` is chosen for `POJO.someMethod()`.

```
public class POJO
{
   String someMethod(String s){}
}

<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->someMethod(..))">
      <after aspect="OneAspect" name="after"/>
   </bind>
</aop>


public class OneAspect
```

```
{
   public void after(@JoinPoint MethodJoinPoint mjp, @Target POJO pojo){} //1
   public String after(@JoinPoint MethodJoinPoint mjp, @Return String ret){} //2
}
```

In the next example, the first `OneAspect.before()` advice is chosen over the second one when calling `POJO.someMethod()`. The reason is that, all else being equal, the first one matches more parameters:.

```
public class POJO
{
   String someMethod(String s, int i){}
}

<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->someMethod(..))">
      <before aspect="OneAspect" name="before"/>
   </bind>
</aop>


public class OneAspect
{
   public void before(@Arg String s, @Arg int i){} //1
   public String before(@Arg String s){} //2
}
```

If the priority of annotated parameters using the presence criterion is the same on more than one advice, the next criterion, the assignability degree, is used.

4.4.1.2. Assignability Degree

The assignability degree rule will select the advice with the lowest assignability degree on the highest priority parameter. The assignability degree is simply the distance in the class hierarchy between the parameter type, and the type it must be assignable from.

As an example, let us look at the following class hierarchy:

```
public interface POJOInterface{}

public class POJOSuperClass extends java.lang.Object{}

public class POJO extends POJOSuperClass implements POJOInterface
{
   void method(){}
}
```

And this advice binding:

```
<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <before aspect="OneAspect" name="before"/>
   </bind>
</aop>

public class OneAspect
{
   public void before(@Target POJO target){} //1
```

```
   public void before(@Target POJOInterface target){} //2
   public void before(@Target POJOSuperClass target){} //3
   public void before(@Target Object target){} //4
}
```

With `POJO` as the target of a joinpoint, the parameter list fo the first `OneAspect.before()` advice method has an assignability degree 0 on `@Target`.

The parameter lists for the second and third `OneAspect.before()` advice methods both have an assignability degree of 1 for `@Target`, since it takes one step through the hierarchy to reach the desired type, `POJO`.

Finally, the parameter list for the fourth `OneAspect.before()` advice method has an assignability degree of 2 on `@Target`.

Hence, JBoss AOP will select the first advice in the example above, since it has the lowest asignability degree on `@Target`.

The assignability degree rule is, similarly to the presence rule, applied on the highest priority annotated parameter, which is `@JoinPoint`. In case there is a match using this criteria (i.e., either both advices lack a `@JoinPoint` annotated parameter, or they both have the same type on the `@JoinPoint` parameter), we move to the next highest priority annotated parameter, which is `@Target`. The same rule is applied until we can find an advice with the highest priority.

Notice that the assignability degree of an advice on `@Arg` is the sum of the assignability degree on all `@Arg` parameters. In the following scenario:

```
public class POJO
{
   public void method(POJO argument0, String argument1, int argument2)
}


<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <before aspect="OneAspect" name="before"/>
   </bind>
</aop>


public class OneAspect
{
   public void before(@Arg POJO p, @Arg String s, @Arg int i){} //1
   public void before(@Arg POJOSuperClass p, @Arg String s, @Arg int i){} //2
   public void before(@Arg POJO p, @Arg Object s, @Arg int i){} //3
   public void before(@Arg Object p, @Arg Object s, @Arg int i){} //4
}
```

The first advice has assignability degree of 0 (for `POJO`) + 0 (for `String`) + 0 (for `int`). Notice how primitive types don't have superclasses, and, hence, have always a 0 value of assinability degree.

The second advice has a larger assignability degree, since `POJOSuperClass` is the superclass of `POJO`, `@Arg POJOSuperClass p` has assignability degree of 1. Hence, this advice assignability degree on `@Arg` is: 1 + 0 + 0 = 1.

The third one also has an assignability degree of 1, since `Object` is the superclass of `String`.

Finally, the last advice has assignability degree of 3 on `@Arg`. The first parameter, `@Arg Object p`, refers to `POJO` and has assignability degree of 2. The second one, assignability degree of 1, since it refers to `String`. And, since `@Arg int` refers to the `int` argument of `POJO.method()`, we have 2 + 1 + 0 = 3.

In the above example, JBoss AOP would select the first advice to intercept `POJO.method()` execution.

4.4.1.3. Return Types

For annotated parameters typed around advices, there is a third rule, which is the return type. This rule also applies to after and finally advices. If the joinpoint has a non-void return type, the assignability degree of the advice return type is analyzed, pretty much in the same way we do with annotated parameters. So, for overloaded around advices, these three criteria are applied:

- presence of annotated parameter
- assignability degree of annotated parameter
- assignability degree of return type

If two advices have the same ranking on the first two criteria, we check their return types and pick the advice with the lowest assignability degree:

```
public class POJO
{
   public Collection method(int arg0, boolean arg1, short arg2) {…}
}


<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <advice aspect="OneAspect" name="around"/>
   </bind>
</aop>


public class OneAspect
{
   public Collection around(@JoinPoint Invocation inv, @Arg int param0) throws Throwable
   {...} //1

   public List around(@JoinPoint Invocation inv, @Arg boolean param1) throws Throwable
   {...} //2

}
```

In OneAspect above, we have two around advices. Both of them are equal when compared using the presence criteria. When comparing them using the assignability of annotated parameter, both of them have the same degrees on `@JoinPoint` and on `@Arg` parameters. In this case, we will compare their return type assignability degree.

Notice that, when it comes to return types, it is the return type that must be assignable to the joinpoint type, and not the contrary. This is due to the fact that JBoss AOP will assign the advice return value to the joinpoint return result in the base system. Hence, in the example above, the caller of `POJO.method()` expects a `Collection` return value. So, it is ok to receive either a `Collection` from the first advice, as the more specific type `List` from the second advice. But JBoss AOP will complain if your advice returns an `Object` (`Object` return type is allowed only in the de-

fault signature; here we are discussing the annotated-parameter signature), because we can't give an `Object` to the base system when it is expecting a `Collection`.

So, in the above example, the first advice has an assignability degree of 0 on the return type, becase it takes 0 steps in the hierarchy to go from Collection to `Collection`. In the second advice, this value is 1, because it takes 1 step to go from `List` to `Collection`. JBoss AOP would select the first advice.

On overloaded after and finally advices, we also have a return type rule. But, since the return type is optional (these advices can return a value, but is not enforced to it), we have a total of four rules for this advice:

- presence of annotated parameter
- assignability degree of annotated parameter
- presence of non-void return type
- assignability degree of return value type

The third rule, presence of non-void return type, states that JBoss AOP will give preference to an after advice that returns a value:

```
<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <after aspect="OneAspect" name="around"/>
   </bind>
</aop>


public class OneAspect
{
   public Collection after(@Arg int param0) {...} //1
   public List after(@Arg boolean param1) { ... } //2
   public void after(@Arg short param2) { ... }   //3
}
```

Considering the same `POJO` class defined previously (with `public void method(int, boolean, short)`), all three overloade versions of `OneAspect.after()` advice wil be considered equivalent in the first two criteria. Hence, we move to the third rule, that states that JBoss AOP prefers an after advice that returns a value over another one that is `void`. So, in the example above, the third advice is ruled out, and JBoss AOP still has two advices to select. Moving to the next rule, he assignability degree of the return type, we have the same result as the `OneAspect.around()` advice: the first one has a 0 degree, and the second one, a 1 degree value. As a conclusion of these degrees, JBoss AOP will select the first advice, with the lowest return assignability degree.

4.4.1.4. A Match

Notice that, iIf JBoss AOP cannot find an advice with highest priority, it just selects one of the methods arbitrarily. This would be the case of the following advice method scenario:

```
public class POJO
{
   public void method(int arg0, long arg1) {…}
}


<aop>
```

```
   <aspect class="OneAspect"/>
   <bind pointcut="execution(* POJO->method(..))">
      <before aspect="OneAspect" name="before"/>
   </bind>
</aop>


public class OneAspect
{
   public void advice(@Arg int arg0) {}
   public void advice(@Arg long arg1) {}
}
```

4.4.1.5. Lowest Priority

There are exceptions for the rules we've seen. Advices with one or more of the following characteristics will be considered lowest priority, regardless of any other criteria:

- an advice that receives `@Target` parameter to intercept a joinpoint with no target available
- an advice that receives `@Caller` parameter to intercept a joinpoint with no caller available
- an advice that receives `@Arg` parameter to intercept a field read joinpoint

4.4.2. Default Signature

For the default around advice signature (i.e., without annotated parameters), there is only one parameter to analyze, the invocation. So, the priority rules are very simple:

- presence of the invocation parameter
- assignability degree of the invocation parameter.

Lets revisit the example given in the beginning of this section, in augmented version:

```
class POJO
{
   public int field;
   public POJO(){}
   public someMethod(){}
}


public class OneAspect
{
   public Object trace(MethodInvocation invocation) throws Throwable {...} //1
   public Object trace(ConstructorInvocation invocation) throws Throwable {...} //2
   public Object trace(Invocation invocation) throws Throwable {...} //3
   public Object trace() throws Throwable {...} //4
}


<aop>
   <aspect class="OneAspect"/>
   <bind pointcut="all(POJO)">
      <advice aspect="OneAspect" name="trace"/>
   </bind>
```

```
</aop>
```

The fourth advice above will never be called, considering the presence rule. It is the only one that lacks the `Invocation` parameter, and would be called only if all others were considered invalid in a scenario, which won't happen in this example. By ruling out this advice with the presence rule, all other advices are equivalent: the invocation parameter is present in all of them. So, we need to move on to the assignability degree rule to select one of them. However, the assignability degree needs to be calculated accordingly to the joinpoint being intercepted. JBoss AOP needs to evaluate each joinpoint type to be intercepted to do the correct selection for each case.

Consider the interception of the constructor of `POJO`. In that case, the first advice is considered invalid, becase a `MethodInvocation` is not assignable from the invocation type that JBoss AOP will provide, `ConstrucorInvocation`. We are now left with the second and third advices. The second one has assignability degree of 0 on the invocation type. The third one, assignability degree of 1 (it takes one step in the hierarchy to go fom `ConstructorInvocation` to `Invocation`). So, in this case, JBoss AOP will select the second advice, because it is the valid advice with the lower assignability degree on the invocation.

Similary, to intercept the execution of `POJO.someMethod()`, JBoss AOP will consider the second advice invalid, because it is supposed to receive an invocation whose type is assignable from `MethodInvocation`. Since the first advice has an assignability degree of 0 on the invocation, and the third one, assignability degree of 1, JBoss AOP will select the first one.

Given that `Invocation` will always be the super class of the expected invocation type, JBoss AOP will select this advice, whose assignability degree will always be 1, only when the other two advices are invalid. That would be the case of a field read, where the invocation type is `FieldReadInvocation`.

4.4.3. Mixing Different Signatures

Finally, when we mix default signature methods with annotated parameter ones, an advice in one of the forms:

```
public Object [advice name]([Invocation] invocation) throws Throwable

public Object [advice name]([Invocation] invocation) throws Throwable

public Object [advice name]() throws Throable
```

Has the highest priority over all annotated-parameter advices. If there is more than one with the default signature, the criteria described in the previous section will be used to select one of them..

Notice that mixing different signatures is possible only with around advices, since only these ones can follow the default signature.

5

XML Bindings

5.1. Intro

In the last sections you saw how to code aspects and how pointcut expressions are formed. This chapter puts it all together. There are two forms of bindings for advices, mixins, and introductions. One is XML which will be the focus of this chapter. The Annotated Bindings chapter discusses how you can replace XML with JDK 5.0 annotations.

5.2. Resolving XML

JBoss AOP resolves pointcut and advice bindings at runtime. So, bindings are a deployment time thing. How does JBoss AOP find the XML files it needs at runtime? There are a couple of ways.

5.2.1. Standalone XML Resolving

When you are running JBoss AOP outside of the application server there are a few ways that the JBoss AOP framework can resolve XML files.

- `jboss.aop.path` This is a system property that is a ';' (Windows) or ':' (Unix) delimited list of XML files and/or directories. If the item in the list is a directory, JBoss AOP will load any xml file in those directories with the filename suffix `-aop.xml`
- `META-INF/jboss-aop.xml` Any JAR file in your CLASSPATH that has a `jboss-aop.xml` file in the `META-INF/` will be loaded. JBoss AOP does a `ClassLoader.getResources("META-INF/jboss-aop.xml")` to obtain all these files.

5.2.2. Application Server XML Resolving

When you are running JBoss AOP outside of the application server there are a few ways that the JBoss AOP framework can resolve XML files. One is to place an XML file with the suffix `*-aop.xml` in the deploy directory. The other way is to JAR up your classes and provide a `META-INF/jboss-aop.xml` file in this JAR. This JAR file must be suffixed with `.aop` and placed within the deploy/ directory or embedded as a nested archive.

5.3. XML DTD

```
<?xml version='1.0' encoding='UTF-8' ?>

<!ELEMENT aop (interceptor|introduction|metadata-loader|metadata|
               stack|aspect|pointcut|pluggable-pointcut|bind|
               prepare|cflow-stack|dynamic-cflow|annotation-introduction|typedef)+>

<!ELEMENT interceptor ANY>
<!ATTLIST interceptor name CDATA #IMPLIED>
<!ATTLIST interceptor class CDATA #IMPLIED>
<!ATTLIST interceptor factory CDATA #IMPLIED>
<!ATTLIST interceptor scope (PER_VM|PER_CLASS|PER_INSTANCE|PER_JOINPOINT) "PER_VM">

<!ELEMENT aspect ANY>
<!ATTLIST aspect name CDATA #IMPLIED>
<!ATTLIST aspect class CDATA #IMPLIED>
<!ATTLIST aspect factory CDATA #IMPLIED>
<!ATTLIST aspect scope (PER_VM|PER_CLASS|PER_INSTANCE|PER_JOINPOINT) "PER_VM">

<!ELEMENT introduction (mixin*,interfaces)>
<!ATTLIST introduction class CDATA #IMPLIED>
<!ATTLIST introduction expr CDATA #IMPLIED>
<!ELEMENT mixin (interfaces, class, construction?)>
<!ATTLIST mixin transient (true|false) "true">
<!ELEMENT interfaces (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT construction (#PCDATA)>

<!ELEMENT metadata-loader EMPTY>
<!ATTLIST metadata-loader tag CDATA #REQUIRED>
<!ATTLIST metadata-loader class CDATA #REQUIRED>

<!ELEMENT metadata ANY>
<!ATTLIST metadata tag CDATA #REQUIRED>
<!ATTLIST metadata class CDATA #REQUIRED>

<!ELEMENT stack (interceptor|interceptor-ref|stack-ref|advice)+>
<!ATTLIST stack name CDATA #REQUIRED>

<!ELEMENT interceptor-ref EMPTY>
<!ATTLIST interceptor-ref name CDATA #REQUIRED>

<!ELEMENT stack-ref EMPTY>
<!ATTLIST stack-ref name CDATA #REQUIRED>

<!ELEMENT advice EMPTY>
<!ATTLIST advice name CDATA #REQUIRED>
<!ATTLIST advice aspect CDATA #REQUIRED>

<!ELEMENT pointcut EMPTY>
<!ATTLIST pointcut name CDATA #REQUIRED>
<!ATTLIST pointcut expr CDATA #REQUIRED>

<!ELEMENT prepare EMPTY>
<!ATTLIST prepare expr CDATA #REQUIRED>

<!ELEMENT pluggable-pointcut ANY>
<!ATTLIST pluggable-pointcut name CDATA #REQUIRED>
<!ATTLIST pluggable-pointcut class CDATA #REQUIRED>

<!ELEMENT bind (interceptor|interceptor-ref|stack-ref|advice|before|around|after|throwing|finally)+>
<!ATTLIST bind name CDATA #IMPLIED>
<!ATTLIST bind pointcut CDATA #REQUIRED>
<!ATTLIST bind cflow CDATA #IMPLIED>

<!ELEMENT cflow-stack (called|not-called)+>
```

```
<!ATTLIST cflow-stack name CDATA #REQUIRED>

<!ELEMENT called EMPTY>
<!ATTLIST called expr CDATA #REQUIRED>
<!ELEMENT not-called EMPTY>
<!ATTLIST not-called expr CDATA #REQUIRED>


<!ELEMENT dynamic-cflow EMPTY>
<!ATTLIST dynamic-cflow name CDATA #REQUIRED>
<!ATTLIST dynamic-cflow class CDATA #REQUIRED>

<!ELEMENT annotation-introduction (#PCDATA)>
<!ATTLIST annotation-introduction expr CDATA #REQUIRED>
<!ATTLIST annotation-introduction invisible (true|false) #REQUIRED>

<!ELEMENT typedef EMPTY>
<!ATTLIST typedef name CDATA #REQUIRED>
<!ATTLIST typedef expr CDATA #REQUIRED>
```

5.4. aspect

The `<aspect>` tag specifies to the AOP container to declare an aspect class. It is also used for configuring aspects as they are created and defining the scope of the aspects instance.

5.4.1. Basic Definition

```
<aspect class="org.jboss.MyAspect"/>
```

In a basic declaration you specify the fully qualified class name of the aspect. If you want to reference the aspect at runtime through the AspectManager, the name of the aspect is the same name as the class name. The default Scope of this aspect is `PER_VM`. Another important note is that aspect instances are created on demand and NOT at deployment time.

5.4.2. Scope

```
<aspect class="org.jboss.MyAspect" scope="PER_VM"/>
```

The `scope` attribute defines when an instance of the aspect should be created. An aspect can be created per vm, per class, per instance, or per joinpoint.

Table 5.1. Aspect instance scope

Name	Description
PER_VM	One and only instance of the aspect class is allocated for the entire VM.
PER_CLASS	One and only instance of the aspect class is allocated for a particular class. This instance will be created if an advice of that aspect is bound to that particular class.

Name	Description
PER_INSTANCE	An instance of an aspect will be created per advised object instance. For instance, if a method has an advice attached to it, whenever an instance of that advised class is allocated, there will also be one created for the aspect.
PER_JOINPOINT	An instance of an aspect will be created per joinpoint advised. If the joinpoint is a static member (constructor, static field/method), then there will be one instance of the aspect created per class, per joinpoint. If the joinpoint is a regular non-static member, than an instance of the aspect will be created per object instance, per joinpoint.
PER_CLASS_JOINPOINT	An instance of an aspect will be created per advised joinpoint. The aspect instance is shared between all instances of the class (for that joinpoint).

5.4.3. Configuration

```
<aspect class="org.jboss.SomeAspect">
     <attribute name="SomeIntValue">55</attribute>
     <advisor-attribute name="MyAdvisor"/>
     <instance-advisor-attribute name="MyInstanceAdvisor"/>
     <joinpoint-attribute name="MyJoinpoint"/>
</aspect>
```

Aspects can be configured by default using a Java Beans style convention. The `<attribute>` tag will delegate to a setter method and convert the string value to the type of the setter method.

Table 5.2. Supported Java Bean types

primitive types (int, float, String, etc...)
java.lang.Class
java.lang.Class[]
java.lang.String[]
java.math.BigDecimal
org.w3c.dom.Document
java.io.File
java.net.InetAddress
java.net.URL
javax.management.ObjectName (if running in JBoss)

Besides types, you can also inject AOP runtime constructs into the aspect. These types of attributes are referenced within XML under special tags. See the table below.

Table 5.3. Injecting AOP runtime constructs

<advisor-attribute>	org.jboss.aop.Advisor
<instance-advisor-attribute>	org.jboss.aop.InstanceAdvisor
<joinpoint-attribute>	org.jboss.aop.joinpoint.Joinpoint

5.4.3.1. Names

If there is no `name` attribute defined, the name of the aspect is the same as the `class` or `factory` attribute value.

5.4.3.2. Example configuration

```
<aspect class="org.jboss.SomeAspect">
        <attribute name="SomeIntValue">55</attribute>
        <advisor-attribute name="MyAdvisor"/>
        <instance-advisor-attribute name="MyInstanceAdvisor"/>
        <joinpoint-attribute name="MyJoinpoint"/>
</aspect>
```

The above example would would need a class implemented as follows:

```
public class SomeAspect {
   public SomeAspect() {}

   public void setSomeIntValue(int val) {...}
   public void setMyAdvisor(org.jboss.aop.Advisor advisor) {...}
   public void setMyInstanceAdvisor(org.jboss.aop.InstanceAdvisor advisor) {...}
   public void setMyJoinpoint(org.jboss.aop.joinpoint.Joinpoint joinpoin) {...}
}
```

5.4.4. Aspect Factories

```
<aspect name="MyAspect" factory="org.jboss.AspectConfigFactory" scope="PER_CLASS">
     <some-arbitrary-xml>value</some-arbitrary-xml>
</aspect>
```

If you do not like the default Java Bean configuration for aspects, or want to delegate aspect creation to some other container, you can plug in your own factory class by specifying the `factory` attribute rather than the `class` attribute. Any arbitrary XML can be specified in the aspect XML declaration and it will be passed to the factory class. Factories must implement the `org.jboss.aop.advice.AspectFactory` interface.

5.5. interceptor

```
<interceptor class="org.jboss.MyInterceptor" scope="PER_VM"/>
<interceptor class="org.jboss.SomeInterceptor">
     <attribute name="SomeIntValue">55</attribute>
     <advisor-attribute name="MyAdvisor"/>
     <instance-advisor-attribute name="MyInstanceAdvisor"/>
```

```
     <joinpoint-attribute name="MyJoinpoint"/>
</interceptor>
<interceptor name="MyAspect" factory="org.jboss.InterceptorConfigFactory" scope="PER_CLASS">
  <some-arbitrary-xml>value</some-arbitrary-xml>
</interceptor>
```

Interceptors are defined in XML the same exact way as aspects are. No difference except the tag. If there is no `name` attribute defined, the name of the interceptor is the same as the `class` or `factory` attribute value.

5.6. bind

```
<bind pointcut="execution(void Foo->bar())">
      <interceptor-ref name="org.jboss.MyInterceptor/>
      <before name="beforeAdvice" aspect="org.jboss.MyAspect"/>
      <around name="aroundAdvice" aspect="org.jboss.MyAspect"/>
      <after name="afterAdvice" aspect="org.jboss.MyAspect"/>
      <throwing name="throwingAdvice" aspect="org.jboss.MyAspect"/>
      <finally name="finallyAdvice" aspect="org.jboss.MyAspect"/>
      <advice name="trace" aspect="org.jboss.MyAspect"/>
</bind>
```

In the above example, the `MyInterceptor` interceptor and several advice methods of the `MyAspect` class will be executed when the `Foo.bar` method is invoked.

bind
: `bind` tag is used to bind an advice of an aspect, or an interceptor to a specific joinpoint. The `pointcut` attribute is required and at least an advice or interceptor-ref definition.

interceptor-ref
: The `interceptor-ref` tag must reference an already existing `interceptor` XML definition. The name attribute should be the name of the interceptor you are referencing.

before, around, after, throwing and finally
: All these tags take a `name` attribute that should map to an advice of the specified type within the aspect class. The `aspect` attribute should be the name of the aspect definition.

advice
: The same as the previous, except for the fact that doesn't specify the type of the advice. This tag selects the default advice type, around, and is hence equivalent to the tag `around`.

5.7. stack

Stacks allow you to define a predefined set of advices/interceptors that you want to reference from within a `bind` element.

```
<stack name="stuff">
      <interceptor class="SimpleInterceptor1" scope="PER_VM"/>
      <advice name="trace" aspect="org.jboss.TracingAspect"/>
      <interceptor class="SimpleInterceptor3">
           <attribute name="size">55</attribute>
      </interceptor>
</stack>
```

After defining the stack you can then reference it from within a `bind` element.

```
<bind pointcut="execution(* POJO->*(..))">
       <stack-ref name="stuff"/>
</bind>
```

5.8. pointcut

The `pointcut` tag allows you to define a pointcut expression, name it and reference it within any binding you want. It is also useful to publish pointcuts into your applications to that others have a clear set of named integration points.

```
<pointcut name="publicMethods" expr="execution(public * *->*(..))"/>
<pointcut name="staticMethods" expr="execution(static * *->*(..))"/>
```

The above define two different pointcuts. One that matches all public methods, the other that matches the execution of all static methods. These two pointcuts can then be referenced within a `bind` element.

```
<bind pointcut="publicMethods AND staticMethods">
      <interceptor-ref name="tracing"/>
</bind>
```

5.9. introduction

5.9.1. Interface introductions

The `introduction` tag allows you to force an existing Java class to implement a particular defined interface.

```
<introduction class="org.acme.MyClass">
    <interfaces>java.io.Serializable</interfaces>
</introduction>
```

The above declaration says that the org.acme.MyClass class will be forced to implement java.io.Serializable. The `class` attribute can take wildcards but not boolean expressions. If you need more complex type expressions, you can use the `expr` attribute instead.

```
<introduction expr="has(* *->@test(..)) OR class(org.acme.*)">
    <interfaces>java.io.Serializable</interfaces>
</introduction>
```

The `expr` can be any type expression allowed in a `typedef` expression

5.9.2. Mixins

When introducing an interface you can also define a mixin class which will provide the implementation of that interface.

```
<introduction class="org.acme.MyClass">
      <mixin>
         <interfaces>
            java.io.Externalizable
         </interfaces>
         <class>org.acme.ExternalizableMixin</class>
         <construction>new org.acme.ExternalizableMixin(this)</construction>
      </mixin>
   </introduction>
```

interfaces
: defines the list of interfaces you are introduction

class
: The type of the mixin class.

construction
: The construction statement allows you to specify any Java code to create the mixin class. This code will be embedded directly in the class you are introducing to so `this` works in the construction statement.

5.10. annotation-introduction

Annotation introductions allow you to embed an annotation within a the class file of the class. You can introduce an annotation to a class, method, field, or constructor.

```
<annotation-introduction expr="constructor(POJO->new())">
      @org.jboss.complex (ch='a', string="hello world", flt=5.5, dbl=6.6, shrt=5, lng=6, integer=7, bool=
</annotation-introduction>
```

The `expr` attribute takes method(), constructor(), class(), or field(). Within those you must define a valid expression for that construct. The following rules must be followed for the annotation declaration:

- Any annotation, Class or Enum referenced, MUST be fully qualified.

5.11. cflow-stack

Control flow is a runtime construct. It allows you to specify pointcut parameters revolving around the call stack of a Java program. You can do stuff like, if method A calls method B calls Method C calls Method D from Constructor A, trigger this advice. In defining a control flow, you must first paint a picture of what the Java call stack should look like. This is the responsibility of the cflow-stack.

```
<cflow-stack name="recursive2">
      <called expr="void POJO->recursive(int)"/>
      <called expr="void POJO->recursive(int)"/>
      <not-called expr="void POJO->recursive(int)"/>
</cflow-stack>
```

A `cflow-stack` has a name and a bunch of `called` and `not-called` elements that define individual constructor or method calls with a Java call stack. The `expr` attribute must be a method or constructor expression. `called` states

that the expr must be in the call stack. `not-called` states that there should not be any more of the expression within the stack. In the above example, the `cflow-stack` will be triggered if there are two and only two calls to the `recursive` method within the stack. Once the `cflow-stack` has been defined, it can then be referenced within a `bind` element through the `cflow` attribute. Boolean expressions are allowed here as well.

```
<bind pointcut="execution(void POJO->recursive(int))" cflow="recursive2 AND !cflow2">
      <interceptor class="SimpleInterceptor"/>
</bind>
```

5.12. typedef

```
<typedef name="jmx" expr="class(@org.jboss.jmx.@MBean) OR
                          has(* *->org.jboss.jmx.@ManagedOperation) OR
                          has(* *->org.jboss.jmx.@ManagedAttribute)"/>
```

`typedefs` allow you to define complex type expressions and then use then pointcut expressions. In the above example, we're defining a class that is tagged as @Mbean, or has a method tagged as @ManagedOperaion or @ManagedAttribute. The above typedef could then be used in a pointcut, introduction, or bind element

```
<pointcut name="stuff" expr="execution(* $typedef{jmx}->*(..))"/>
<introduction expr="class($typedef{jmx})">
```

5.13. dynamic-cflow

`dynamic-cflow` allows you to define code that will be executed that must be resolved true to trigger positive on a cflow test on an advice binding. (See Dynamic CFlow for more information). The test happens dynamically at runtime and when combined with a pointcut expression allows you to do runtime checks on whether a advice binding should run or not. Create a dynamic cflow class, then you must declare it with XML so that it can be used in bind expressions.

```
<dynamic-cflow name="simple" class="org.jboss.SimpleDynamicCFlow"/>
```

You can then use it within a `bind`

```
<bind expr="execution(void Foo->bar())" cflow="simple">
```

5.14. prepare

The `prepare` tag allows you to define a pointcut expression. Any joinpoint that matches the expression will be aspectized and bytecode instrumented. This allows you to hotdeploy and bind aspects at runtime as well as to work with the per instance API that every aspectized class has. To prepare something, just define a pointcut expression that matches the joinpoint you want to instrument.

```
<prepare expr="execution(void Foo-bar())"/>
```

5.15. metadata

You can attach untyped metadata that is stored in `org.jboss.aop.metadata.SimpleMetaData structures` within the `org.jboss.aop.Advisor` class that manages each aspectized class. The XML mapping has a section for each type of metadata. Class, method, constructor, field, and defaults for the whole shabang. Here's an example:

```
<metadata tag="testdata" class="org.jboss.test.POJO">
      <default>
         <some-data>default value</some-data>
      </default>
      <class>
         <data>class level</data>
      </class>
      <constructor expr="POJOConstructorTest()">
         <some-data>empty</some-data>
      </constructor>
      <method expr="void another(int, int)">
         <other-data>half</other-data>
      </method>
      <field name="somefield">
         <other-data>full</other-data>
      </field>
</metadata>
```

Any element can be defined under the class, default, method, field, and constructor tags. The name of these elements are used as attribute names in SimpleMetaData structures. The `tag` attribute is the name used to reference the metadata within the Advisor, or Invocation lookup mechanisms.

5.16. metadata-loader

```
<metadata-loader tag="security" class="org.jboss.aspects.security.SecurityClassMetaDataLoader"/>
```

If you need more complex XML mappings for untyped metadata, you can write your own metadata binding. The tag attribute is used to trigger the loader. The loader class must implement the `org.jboss.aop.metadata.ClassMetaDataLoader` interface.

```
public interface ClassMetaDataLoader
{
   public ClassMetaDataBinding importMetaData(Element element, String name,
                                              String tag, String classExpr) throws Exception;

   public void bind(ClassAdvisor advisor, ClassMetaDataBinding data,
                    CtMethod[] methods, CtField[] fields, CtConstructor[] constructors) throws Exception;

   public void bind(ClassAdvisor advisor, ClassMetaDataBinding data,
                    Method[] methods, Field[] fields, Constructor[] constructors) throws Exception;
}
```

Any arbitrary XML can be in the `metadata` element. The ClassMetaDataBinding.importMetaData method is responsible for parsing the element and building ClassMetaDataBinding structurs which are used in the precompiler and runtime bind steps. Look at the SecurityClassMetaDataLoader code shown above for a real concrete example.

5.17. precedence

Precedence allows you to impose an overall relative sorting order of your interceptors and advices.

```
        <precedence>
           <interceptor-ref name="org.acme.Interceptor"/>
           <advice aspect="org.acme.Aspect" name="advice1"/>
           <advice aspect="org.acme.Aspect" name="advice2"/>
        </precedence>
```

This says that when a joinpoint has both `org.acme.Interceptor` and `org.acme.Aspect.advice()` bound to it, `org.acme.Interceptor` must always be invoked before `org.acme.Aspect.advice1()` which must in turn be invoked before `org.acme.Aspect.advice2()`. The ordering of interceptors/advices that do not appear in a precedence is defined by their ordering for the individual bindings or intercerceptor stacks.

5.18. declare

You can declare checks to be enforced at instrumentation time. They take a pointcut and a message. If the pointcut is matched, the message is printed out.

5.18.1. declare-warning

```
   <declare-warning expr="class($instanceof{VehicleDAO}) \
            AND !has(public void *->save())">
         All VehicleDAO subclasses must override the save() method.
   </declare-warning>
```

The above declaration says that if any subclass of VehicleDAO does not implement a noargs save() method, a warning with the supplied message should be logged. Your application will continue to be instrumented/run (since we are using `declare-warning` in this case).

5.18.2. declare-error

```
      <declare-error expr="call(* org.acme.businesslayer.*->*(..)) \
            AND within(org.acme.datalayer.*)">
      Data layer classes should not call up to the business layer
   </declare-error>
```

The above declaration says that if any classes in the datalayer call classes in the business layer of your application, an error should be thrown. Instumentation/execution of your application will stop.

6

Annotation Bindings

JDK 5.0 has introduced a new concept called annotations. Annotations can be used as an alternative to XML for configuring classes for AOP. For backward compatibility with JDK 1.4.2, refer to Chapter 8

6.1. @Aspect

To mark a class as an aspect you annotate it with the `@Aspect` annotation. Remember that a class to be used as an aspect does not need to inherit or implement anything special, but it must have an empty constuctor and contain one or more methods (advices) of the format:

```
public Object <any-method-name>(org.jboss.aop.joinpoint.Invocation)
```

The declaration of `org.jboss.aop.Aspect` is:

```
package org.jboss.aop;

import org.jboss.aop.advice.Scope;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


@Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME)
        public @interface Aspect
{
   Scope scope() default Scope.PER_VM;
}
```

and Scope is:

```
package org.jboss.aop.advice;

public enum Scope
{
   PER_VM, PER_CLASS, PER_INSTANCE, PER_JOINPOINT
}
```

See Section 5.4.2 for a description of the various scopes.

We use the @Aspect annotation as follows:

```
package com.mypackage;
```

```
import org.jboss.aop.Aspect;
import org.jboss.aop.advice.Scope;
import org.jboss.aop.joinpoint.Invocation;

@Aspect (scope = Scope.PER_VM)
public class MyAspect
{
   public Object myAdvice(Invocation invocation)
}
```

The name of the class (in this case `com.mypackage.MyAspect`) gets used as the internal name of the aspect. The equivalent using XML configuration would be:

```
          <aop>
          <aspect class="com.mypackage.MyAspect" scope="PER_VM"/>
          </aop>
```

6.2. @InterceptorDef

To mark a class as an interceptor or an aspect factory you annotate it with the `@InterceptorDef` annotation. The class must either implement the `org.jboss.aop.advice.Interceptor` interface or the `org.jboss.aop.advice.AspectFactory` interface.

The declaration of `org.jboss.aop.InterceptorDef` is:

```
package org.jboss.aop;

@Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME)
        public @interface Aspect
{
   Scope scope() default Scope.PER_VM;
}
```

The same `Scope` enum is used as for `Aspect`. The following examples use the @Bind annotation, which will be described in more detail below.

6.2.1. Interceptor Example

We use the `@InterceptorDef` annotation to mark an Interceptor as follows:

```
package com.mypackage;

import org.jboss.aop.Bind;
import org.jboss.aop.InterceptorDef;
import org.jboss.aop.advice.Interceptor;

@InterceptorDef (scope = Scope.PER_VM)
@Bind (pointcut="execution("* com.blah.Test->test(..)")
```

```
public class MyInterceptor implements Interceptor
{
   public Object invoke(Invocation invocation)throws Throwable
   {
      return invocation.invokeNext();
   }
}
```

The name of the class (in this case `com.mypackage.MyInterceptor`) gets used as the class name of the interceptor. The equivalent using XML configuration would be:

```
            <aop>
            <interceptor class="com.mypackage.MyInterceptor" scope="PER_VM"/>
            </aop>
```

6.2.2. AspectFactory Example

The `@InterceptorDef` annotation is used to mark an AspectFactory as follows:

```
package com.mypackage;

import org.jboss.aop.advice.AspectFactory;

@InterceptorDef (scope=org.jboss.aop.advice.Scope.PER_VM)
@Bind (pointcut="execution("* com.blah.Test->test2(..)")
public class MyInterceptorFactory implements AspectFactory
{
   //Implemented methods left out for brevity
}
```

6.3. @PointcutDef

To define a named pointcut you annotate a field within an `@Aspect` or `@InterceptorDef` annotated class with `@PointcutDef`. `@PointcutDef` only applies to fields and is not recognised outside `@Aspect` or `@InterceptorDef` annotated classes.

The declaration of `org.jboss.aop.PointcutDef` is:

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
public @interface PointcutDef
{
   String value();
}
```

`@PointcutDef` takes only one value, a valid pointcut expression. The name of the pointcut used internally and when yo want to reference it is:

```
<name of @Aspect/@InterceptorDef annotated class>.<name of @PointcutDef annotated field>
```

An example of an aspect class containing a named pointcut which it references from a bindng's pointcut expression:

```
   package com.mypackage;

   import org.jboss.aop.PointcutDef;
   import org.jboss.aop.pointcut.Pointcut;

   @Aspect (scope = Scope.PER_VM)
   public class MyAspect
   {
      @PointcutDef ("(execution(* org.blah.Foo->someMethod()) OR \
              execution(* org.blah.Foo->otherMethod()))")
      public static Pointcut fooMethods;

      public Object myAdvice(Invocation invocation)
      {
         return invocation.invokeNext();
      }
   }
```

It is worth noting that named pointcuts can be referenced in pointcut expressions outside the class they are declared in (if the annotated fields are declared public of course!).

Using XML configuration this would be:

```
           <aop>
           <aspect class="com.mypackage.MyAspect" scope="PER_VM"/>
           <pointcut
           name="com.mypackage.MyAspect.fooMethods"
           expr="(execution(* org.blah.Foo->someMethod()) OR \
                 execution(* org.blah.Foo->otherMethod()))"
      />
           </aop>
```

6.4. @Bind

To create a binding to an advice method from an aspect class, you annotate the advice method with `@Bind`. To create a binding to an Interceptor or AspectFactory, you annotate the class itself with `@Bind` since Interceptors only contain one advice (the `invoke()` method). The @Bind annotation will only be recognised in the situations just mentioned.

The declaration of `org.jboss.aop.Bind` is:

```
package org.jboss.aop;

@Target({ElementType.METHOD, ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME)
public @interface Bind
{
  String pointcut();
  String cflow() default "";
```

```
}
```

The `@Bind` annotation takes two parameters:

- `pointcut`, which is a pointcut expression resolving to the joinpoints you want to bind an aspect/interceptor to
- `cflow`, which is optional. If defined it must resolve to the name of a defined cflow.)

In the case of a binding to an advice in an aspect class, the internal name of the binding becomes:

```
<name of the aspect class>.<the name of the advice method>
```

In the case of a binding to an `Interceptor` or `AspectFactory` implementation, the internal name of the binding becomes:

```
<name of the Interceptor/AspectFactory implementation class>
```

An example of a binding using an advice method in an aspect class:

```
package com.mypackage;

import org.jboss.aop.Bind;

@Aspect (scope = Scope.PER_VM)
public class MyAspect
{
   @PointcutDef ("(execution(* org.blah.Foo->someMethod()) \
         OR execution(* org.blah.Foo->otherMethod()))")
   public static Pointcut fooMethods;

   @Bind (pointcut="com.mypackage.MyAspect.fooMethods")
   public Object myAdvice(Invocation invocation)
   {
      return invocation.invokeNext();
   }

   @Bind (pointcut="execution("* org.blah.Bar->someMethod())")
   public Object myAdvice(Invocation invocation)
   {
      return invocation.invokeNext();
   }

}
```

The equivalent using XML configuration would be:

```
      <aop>
      <aspect class="com.mypackage.MyAspect" scope="PER_VM"/>
      <pointcut
      name="com.mypackage.MyAspect.fooMethods"
      expr="(execution("* org.blah.Foo->someMethod()) OR \
            execution("* org.blah.Foo->otherMethod()))"
   />
      <bind pointcut="com.mypackage.MyAspect.fooMethods">
      <advice name="myAdvice" aspect="com.mypackage.MyAspect">
      </bind>
      <bind pointcut="execution("* org.blah.Bar->someMethod())">
```

```
        <advice name="otherAdvice" aspect="com.mypackage.MyAspect">
        </bind>
        </aop>
```

Revisiting the examples above in the @InterceptorDef section, now that we know what @Bind means, the equivalent using XML configuration would be:

```
           <aop>
           <interceptor class="com.mypackage.MyInterceptor" scope="PER_VM"/>
           <interceptor factory="com.mypackage.MyInterceptorFactory" scope="PER_VM"/>

           <bind pointcut="execution("* com.blah.Test->test2(..)">
           <interceptor-ref name="com.mypackage.MyInterceptor"/>
           </bind>
           <bind pointcut="execution("* com.blah.Test->test2(..)">
           <interceptor-ref name="com.mypackage.MyInterceptorFactory"/>
           </bind>
           </aop>
```

6.5. @Introduction

Interface introductions can be done using the `@Introduction` annotation. Only fields within a class annotated with `@Aspect` or `@InterceptorDef` can be annotated with `@Introduction`.

The declaration of `org.jboss.aop.Introduction`:

```
  package org.jboss.aop;

  @Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
  public @interface Introduction
  {
     Class target() default java.lang.Class.class;
     String typeExpression() default "";
     Class[] interfaces();
  }
```

The parameters of `@Introduction` are:

- `target`, the name of the class we want to introduce an interface to.
- `typeExpression`, a type expression that should resolve to one or more classes we want to introduce an interface to.
- `interfaces`, an array of the interfaces we want to introduce

`target` or `typeExpression` has to be specified, but not both.

This is how to use this annotation:

```
  package com.mypackage;

  import org.jboss.aop.Introduction;
```

```
@Aspect (scope = Scope.PER_VM)
public class IntroAspect
{
   @Introduction (target=com.blah.SomeClass.class, \
         interfaces={java.io.Serializable.class})
   public static Object pojoNoInterfacesIntro;
}
```

This means make `com.blah.SomeClass.class` implement the `java.io.Serializable` interface. The equivalent configured via XML would be:

```
         <introduction class="com.blah.SomeClass.class">
         <interfaces>
      java.io.Serializable
         </interfaces>
         </introduction>
```

6.6. @Mixin

Sometimes when we want to introduce/force a new class to implement an interface, that interface introduces new methods to a class. The class needs to implement these methods to be valid. In these cases a mixin class is used. The mixin class must implement the methods specified by the interface(s) and the main class can then implement these methods and delegate to the mixin class.

Mixins are created using the @Mixin annotation. Only methods within a class annotated with `@Aspect` or `@InterceptorDef` can be annotated with `@Mixin`. The annotated method has

- be public
- be static
- have an empty parameter list, or receive the target of introduction as parameter
- contain the logic to create the mixin class
- return an instance of the mixin class

The declaration of `org.jboss.aop.Mixin`:

```
package org.jboss.aop;

@Target({ElementType.METHOD}) @Retention(RetentionPolicy.RUNTIME)
public @interface Mixin
{
   Class target() default java.lang.Class.class;
   String typeExpression() default "";
   Class[] interfaces();
   boolean isTransient() default true;
}
```

The parameters of `@Mixin` are:

- `target`, the name of the class we want to introduce an interface to.
- `typeExpression`, a type expression that should resolve to one or more classes we want to introduce an interface to.
- `interfaces`, an array of the interfaces we want to introduce, implemented by the mixin class.
- `isTransient`. Internally AOP makes the main class keep a reference to the mixin class, and this sets if that reference should be transient or not. The default is true.

`target` or `typeExpression` has to be specified, but not both.

An example aspect using `@Mixin` follows:

```
package com.mypackage;

import org.jboss.aop.Mixin;
import com.mypackage.POJO;

@Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
public class IntroductionAspect
{
   @Mixin (target=com.mypackage.POJO.class, interfaces={java.io.Externalizable.class})
   public static ExternalizableMixin createExternalizableMixin(POJO pojo) {
       return new ExternalizableMixin(pojo);
   }
}
```

Since this is slightly more complex than the previous examples we have seen, the `POJO` and `ExternalizableMixin` classes are included here.

```
package com.mypackage;

public class POJO
{
   String stuff;
}
```

```
package com.mypackage;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class ExternalizableMixin implements Externalizable
{
   POJO pojo;

   public ExternalizableMixin(POJO pojo)
   {
      this.pojo = pojo;
   }
```

```
   public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
   {
      pojo.stuff = in.readUTF();
   }

   public void writeExternal(ObjectOutput out) throws IOException
   {
      out.writeUTF(pojo.stuff);
   }
}
```

This has the same effect as the following XML configuration:

```
         <introduction classs="com.mypackage.POJO">
         <mixin transient="true">
         <interfaces>
           java.io.Externalizable
         </interfaces>
         <class>com.mypackage.ExternalizableMixin</class>
         <construction>IntroductionAspect.createExternalizableMixin(this)</construction>
         </mixin>
         </introduction>
```

6.7. @Prepare

To prepare a joinpoint or a set of joinpoints for DynamicAOP annotate a field with `@Prepare` in a class anotated with `@Aspect` or `@InterceptorDef`.

The declaration of `org.jboss.aop.Prepare` is:

```
package org.jboss.aop;

@Target({ElementType.FIELD, ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME)
      public @interface Prepare {
    String value() default "";
}
```

The single field `value` contains a pointcut expression matching one or more joinpoints.

To use `@Prepare` follow this example:

```
package com.mypackage;

import org.jboss.aop.Prepare;

@InterceptorDef (scope = Scope.PER_VM)
@Bind (pointcut="execution("* com.blah.Test->test(..)")
public class MyInterceptor2 implements Interceptor
{
   @Prepare ("all(com.blah.DynamicPOJO)")
   public static Pointcut dynamicPOJO;

   public Object invoke(Invocation invocation)throws Throwable
   {
```

```
         return invocation.invokeNext();
      }
   }
```

Using XML configuration instead we would write:

```
           <prepare expr="all(com.blah.DynamicPOJO)"/>
```

This simple example used an `@InterceptorDef` class for a bit of variety in the examples, and to reiterate that `@Pointcut`, `@Introduction`, `@Mixin`, `@Prepare`, `@Typedef`, `@CFlow`, `@DynamicCFlow` and `@AnnotationIntroductionDef` can all be used both in `@InterceptorDef` annotated classes AND `@Aspect` annotated classes. Same for `@Bind`, but that is a special case as mentioned above.

6.7.1. @Prepare POJO

You can also annotate a POJO with @Prepare directly in cases where you are using Dynamic AOP, and the exact bindings are not known at instrumentation time. In this case you annotate the class itself. Here's how it is done:

```
   package com.mypackage;

   import org.jboss.aop.Prepare;

   @Prepare ("all(this)")
   public class MyDynamicPOJO implements Interceptor
   {
      ...
   }
```

`all(this)` means the same as `all(com.blah.MyDynamicPOJO)`, but the use of `all(this)` is recommended.

The examples just given equate to this XML

```
           <prepare expr="all(com.blah.MyDynamicPOJO)"/>
```

To summarise, when using @Prepare within an @Interceptor or @Aspect annotated class, you annotate a field within that class. When using @Prepare with a POJO you annotate the class itself.

6.8. @TypeDef

To use a typedef, you annotate a field with `@TypeDef` in a class anotated with `@Aspect` or `@InterceptorDef`.

The declaration of `org.jboss.aop.TypeDef`:

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
```

```
public @interface TypeDef {
    String value();
}
```

The single `value` field takes a type expression that resolves to one or more classes. The name of the typedef used for reference and internally is:

```
<name of @Aspect/@InterceptorDef annotated class>.<name of @TypeDef annotated field>
```

Here's how to use it:

```
   package com.mypackage;

   import org.jboss.aop.TypeDef;
   import org.jboss.aop.pointcut.Typedef;
   @Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
   public class TypedefAspect
   {
      @TypeDef ("class(com.blah.POJO)")
      public static Typedef myTypedef;

       @Bind (pointcut="execution(* \
             $typedef{com.mypackage.TypedefAspect.myTypedef}->methodWithTypedef())")
       public Object typedefAdvice(Invocation invocation) throws Throwable
      {
         return invocation.invokeNext();
      }
   }
```

The equivalent using XML configuration would be:

```
          <aop>
          <aspect class="com.mypackage.TypedefAspect" scope="PER>VM"/>
          <typedef name="com.mypackage.TypedefAspect.myTypedef" expr="class(com.blah.POJO)"/>
          <bind
       pointcut="execution(* \
          $typedef{com.mypackage.TypedefAspect.myTypedef}->methodWithTypedef())"
          >
          <advice name="typedefAdvice" aspect="com.mypackage.TypedefAspect"/>
          </bind>
          </aop>
```

6.9. @CFlowDef

To create a CFlow stack, you annotate a field with `@CFlowDef` in a class anotated with `@Aspect` or `@InterceptorDef`. The declaration of `org.jboss.aop.CFlowStackDef` is:

```
   package org.jboss.aop;

   @Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
         public @interface CFlowStackDef
   {
      CFlowDef[] cflows();
```

```
   }
```

In turn the declaration of `org.jboss.aop.CFlowDef` is:

```
   package org.jboss.aop;

   public @interface CFlowDef {
       boolean called();
       String expr();
   }
```

The parameters of `@CFlowDef` are:

- `called`, whether the corresponding expr should appear in the stack trace or not.
- `expr`, a string matching stack a trace element

The name of the CFlowStackDef used for reference and internally is:

```
<name of @Aspect/@InterceptorDef annotated class>.<name of @CFlowStackDef annotated field>
```

CFlowStackDef is used like the following example:

```
   package com.mypackage;

   import org.jboss.aop.CFlowStackDef;
   import org.jboss.aop.pointcut.CFlowStack;

   @Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
   public class CFlowAspect
   {

      @CFlowStackDef (cflows={@CFlowDef(expr= "void com.blah.POJO->cflowMethod1()", \
            called=false),  @CFlowDef(expr = "void com.blah.POJO->cflowMethod2()", \
            called=true)})
      public static CFlowStack cfNot1And2Stack;

      @Bind (pointcut="execution(void com.blah.POJO*->privMethod())", \
            cflow="com.mypackage.CFlowAspect.cfNot1And2Stack")
      public Object cflowAdvice(Invocation invocation) throws Throwable
      {
         return invocation.invokeNext();
      }
   }
```

The above means the same as this XML:

```
            <aop>
            <cflow-stack name="com.mypackage.CFlowAspect.cfNot1And2Stack">
            <called expr="void com.blah.POJO->cflowMethod1()"/>
            <not-called expr="void com.blah.POJO->cflowMethod2()"/>
            </cflow-stack>
            </aop>
```

6.10. @DynamicCFlowDef

To create a dynamic CFlow you annotate a class implementing `org.jboss.aop.pointcut.DynamicCFlow` with `@DynamicCFlowDef`. The declaration of `@org.jboss.aop.DynamicCFlowDef` is:

```
package org.jboss.aop;

@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface DynamicCFlowDef
{
}
```

Here is a @DynamicCFlow annotated class:

```
package com.mypackage;

import org.jboss.aop.DynamicCFlowDef;
import org.jboss.aop.pointcut.DynamicCFlow;

@DynamicCFlowDef
public class MyDynamicCFlow implements DynamicCFlow
{
   public static boolean execute = false;

   public boolean shouldExecute(Invocation invocation)
   {
      return execute;
   }
}
```

The name of the `@DynamicCFlowDef` annotated class gets used as the name of the cflow for references.

To use the dynamic cflow we just defined:

```
package com.mypackage;

@Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
public class CFlowAspect
{
   @Bind (pointcut="execution(void com.blah.POJO->someMethod())", \
         cflow="com.mypackage.MyDynamicCFlow")
   public Object cflowAdvice(Invocation invocation) throws Throwable
   {
      return invocation.invokeNext();
   }
}
```

6.11. @AnnotationIntroductionDef

You can introduce annotations by annotating a field with the `@AnnotationIntroductionDef` in a class anotated with `@Aspect` or `@InterceptorDef`. The declaration of `org.jboss.aop.AnnotationIntroductionDef` is:

```
package org.jboss.aop;

@Target (ElementType.FIELD) @Retention(RetentionPolicy.RUNTIME)
      public @interface AnnotationIntroductionDef
{
   String expr();
   boolean invisible();
   String annotation();
}
```

The parameters of `@AnnotationIntroductionDef` are:

- `expr`, pointcut matching the classes/constructors/methods/fields we want to annotate.
- `invisible`, if true: the annotation's retention is RetentionPolicy.CLASS; false: RetentionPolicy.RUNTIME
- `annotation`, the annotation we want to introduce.

The listings below make use of an annotation called `@com.mypackage.MyAnnotation`:

```
package com.mypackage;
public interface MyAnnotation
{
   String string();
   int integer();
   boolean bool();
}
```

What its parameters mean is not very important for our purpose.

The use of `@AnnotationIntroductionDef`:

```
package com.mypackage;

import org.jboss.aop.AnnotationIntroductionDef:
import org.jboss.aop.introduction.AnnotationIntroduction;

@.InterceptorDef (scope=org.jboss.aop.advice.Scope.PER_VM)
@org.jboss.aop.Bind (pointcut="all(com.blah.SomePOJO)")
public class IntroducedAnnotationInterceptor implements Interceptor
{
   @org.jboss.aop.AnnotationIntroductionDef \
         (expr="method(* com.blah.SomePOJO->annotationIntroductionMethod())", \
          invisible=false, \
          annotation="@com.mypackage.MyAnnotation \
             (string='hello', integer=5, bool=true)")
   public static AnnotationIntroduction annotationIntroduction;

   public String getName()
   {
      return "IntroducedAnnotationInterceptor";
   }

   public Object invoke(Invocation invocation) throws Throwable
   {
      return invocation.invokeNext();
   }
}
```

```
```

Note that the reference to `@com.mypackage.MyAnnotation` must use the fully qualified class name, and that the value for its string parameter uses single quotes.

The previous listings are the same as this XML configuration:

```
      <annotation-introduction
   expr="method(* com.blah.SomePOJO->annotationIntroductionMethod())
   invisible="false"
      >
   @com.mypackage.MyAnnotation (string="hello", integer=5, bool=true)
      </annotation-introduction>
```

6.12. @Precedence

You can declare precedence by annotating a class with `@Precedence`, and then annotate fields where the types are the various Interfaces/Aspects you want to sort. You annotate fields where the type is an interceptor with `@PrecedenceInterceptor`. When the type is an aspect class, you annotate the field with `@PrecedenceAdvice`. The definitions of org.jboss.aop.Precedence, org.jboss.aop.PrecedenceInterceptor and org.jboss.aop.PrecedenceAdvice are

```
package org.jboss.aop;

@Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME)
public @interface Precedence
{
}
```

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
public @interface PrecedenceInterceptor
{
}
```

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
public @interface PrecedenceAdvice
{
   String value();
}
```

The `value()` attribute of `PrecedenceAdvice` is the name of the advice method to use.

The example shown below declares a relative sort order where `org.acme.Interceptor` must always be invoked before `org.acme.Aspect.advice1()` which must be invoked before `org.acme.Aspect.advice2()`:

```
import org.jboss.aop.Precedence;
import org.jboss.aop.PrecedenceAdvice;

@Precedence
public class MyPrecedence
{
   @PrecedenceInterceptor
   org.acme.Interceptor intercept;

   @PrecedenceAdvice ("advice1")
   org.acme.Aspect precAdvice1;

   @PrecedenceAdvice ("advice2")
   org.acme.Aspect precAdvice2;
}
```

The ordering of interceptors/advices defined via annotations that have no precedence defined, is arbitrary.

6.13. @DeclareError and @DeclareWarning

You can declare checks to be enforced at instrumentation time. They take a pointcut and a message. If the pointcut is matched, the message is printed out. To use this with annotations, annotate fields with `DeclareWarning` or `DeclareError` within a class annotated with `@Aspect` or `@InterceptorDef`. The definitions of `org.jboss.aop.DeclareError` and `org.jboss.aop.DeclareWarning` are:

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
public @interface DeclareWarning
{
   String expr();
   String msg();
}
```

```
package org.jboss.aop;

@Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)
public @interface DeclareError
{
   String expr();
   String msg();
}
```

For both: the expr() attribute is a pointcut expression that should not occur, and the msg() attribute is the message to print out if a match is found for the pointcut. If you use `DeclareWarning` instrumentation/your application will simply continue having printed the message you supplied. In the case of `DeclareError`, the message is logged and an error is thrown, causing instrumentation/your application to stop. Here is an example:

```
import org.jboss.aop.Aspect;
import org.jboss.aop.pointcut.Pointcut;
import org.jboss.aop.DeclareError;
```

```
import org.jboss.aop.DeclareWarning;

@Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
public class DeclareAspect
{
   @DeclareWarning (expr="class($instanceof{VehicleDAO}) AND \
      !has(public void *->save())", \
      msg="All VehicleDAO subclasses must override the save() method.")
   Pointcut warning;

   @DeclareError (expr="call(* org.acme.businesslayer.*->*(..)) \
      AND within(org.acme.datalayer.*)", \
      msg="Data layer classes should not call up to the business layer")
   Pointcut error;
}
```

7

Dynamic AOP

7.1. Hot Deploment

With JBoss AOP you can change advice and interceptor bindings at runtime. You can unregister existing bindings, and hot deploy new bindings if the given joinpoints have been instrumented. Hot-deploying within the JBoss application server is as easy as putting (or removing) a `*-aop.xml` file or `.aop` jar file within the deploy/ directory. There is also a runtime API for adding advice bindings at runtime. Getting an instance of `org.jboss.aop.AspectManager.instance()`, you can add your binding.

```
      org.jboss.aop.advice.AdviceBinding binding = new AdviceBinding("execution(POJO->new(..))", null);
      binding.addInterceptor(SimpleInterceptor.class);
      AspectManager.instance().addBinding(binding);
```

First, you allocated an `AdviceBinding` passing in a pointcut expression. Then you add the interceptor via its class and then add the binding through the AspectManager. When the binding is added the AspectManager will iterate through ever loaded class to see if the pointcut expression matches any of the joinpoints within those classes.

7.2. Per Instance AOP

Any class that is instrumented by JBoss AOP, is forced to implement the `org.jboss.aop.Advised interface.`

```
public interface InstanceAdvised
{
   public InstanceAdvisor _getInstanceAdvisor();
   public void _setInstanceAdvisor(InstanceAdvisor newAdvisor);
}

public interface Advised extends InstanceAdvised
{
   public Advisor _getAdvisor();
}
```

The InstanceAdvisor is the interesting interface here. InstanceAdvisor allows you to insert Interceptors at the beginning or the end of the class's advice chain.

```
public interface InstanceAdvisor
{
   public void insertInterceptor(Interceptor interceptor);
   public void removeInterceptor(String name);
   public void appendInterceptor(Interceptor interceptor);

   public void insertInterceptorStack(String stackName);
   public void removeInterceptorStack(String name);
   public void appendInterceptorStack(String stackName);
```

```
   public SimpleMetaData getMetaData();

}
```

So, there are three advice chains that get executed consecutively in the same java call stack. Those interceptors that are added with the `insertInterceptor()` method for the given object instance are executed first. Next, those advices/interceptors that were bound using regular `binds` . Finally, those interceptors added with the `appendInterceptor()` method to the object instance are executed. You can also reference `stacks` and insert/append full stacks into the pre/post chains.

Besides interceptors, you can also append untyped metadata to the object instance via the getMetaData() method.

7.3. Preparation

Dynamic AOP cannot be used unless the particular joinpoint has been instrumented. You can force intrumentation with the `prepare` functionality

7.4. DynamicAOP with HotSwap

When running JBoss AOP with HotSwap, the dynamic AOP operations may result in the weaving of bytecodes. In this case, the flow control of joinpoints matched only by `prepare` expressions is not affected before any advices or interceptors are applied to them via dynamic aop. Only then, the joinpoint bytecodes will be weaved to start invoking the added advices and interceptors and, as a result, their flow control will be affected.

On the other hand, if HotSwap is disabled, the joinpoints matched by `prepare` expressions are completely instrumented and the flow control is affected before classes get loaded, even if no interceptors are applied to them with dynamic aop.

To learn how to enable HotSwap, refer to the "Running Aspectized Application" chapter.

8

JDK 1.4.2 Backwards Compatibility

Despite the fact that it has been a while since the release of JDK 5, there are projects that still require backwards compatibility with JDK 1.4.2. This is relatively common, and the reasons for it are diverse. To deal with this type of requirement, JBoss AOP provides two different solutions. By using one of them, you can use all features JBoss AOP provides, including annotation-related features, and still keep your code backwards compatible with JDK1.4.2.

8.1. JBoss Retro

The first solution to achieve JDK 1.4.2 backwards compatibility is to use the JBoss Retro tool. JBoss Retro converts JDK 5 compiled bytecodes into bytecodes that can be run using a JDK 1.4.2 virtual machine.

This tool acts as a bytecode processor, and using it is very simple. You just need to write your code using JDK 5 constructs and features, and then compile it, using JDK 5. Next, process the generated bytecodes using JBoss Retro tool (just like you do when you use aopc compiler), and now your bytecodes are ready to run using a JDK 1.4.2 virtual machine. Notice this solution allows not only the use of JBoss AOP annotations, but of most JDK 5 features and new API operations.

The simplest way to run JBoss Retro is using the ant task. You just need to declare it:

```
<taskdef name="retro" classname="org.jboss.ant.tasks.retro.Retro" classpathref="jboss.retro.classpath"/>
```

And use it as in the following example:

```
<retro compilerclasspathref="jboss.retro.classpath" destdir=".">
   <classpath refid="jboss.retro.classpath"/>
   <classpath path="."/>
   <src path="."/>
</retro>
```

This task takes the following parameters:

- `compilerclasspathref` - This represent the jars needed for the JBoss Retro processor to work. The `compilerclasspath` version takes the paths of the jar files, and the `compilerclasspathref` version takes the name of a predefined ant path.
- `classpath` or `classpathref` - Path to the compiled classes to be processed by JBoss Retro. The `classpath` version takes the path of the directory, and the `classpathref` version takes the name of a predefined ant path.
- `verbose` - Default is false. If true, verbose output is generated, which comes in handy for diagnosing unexpected results.

- `suppress` - Default is true. If false, error messages will not be suppressed.
- `maxmemory` - Sets the maximum memory of the java task that will be forked.
- `destdir` - the dir where JBoss Retro will write the resulting bytecodes.

It is also possible to run JBoss Retro with the following command line:

```
$ java -cp <all the JBoss AOP jars and the directory containing files we want to AOP> \
      -verbose <true/false> -suppress <true/false> -destdir <
      org.jboss.ant.tasks.retro.Weaver \
      [-bytecode]<files>+
```

JBoss Retro is the de facto standard solution JBoss Group provides to achieve JDK 1.4.2 backward compatibility. However, if you do not have the option to compile your code using JDK 5, you should go with the next solution, the annotation compiler.

8.2. JDK1.4.2 Annotation Compiler

Unlike JBoss Retro, the annotation compiler does not support all JDK 5 constructs and new APIs. Its functionality consists in supporting only annotations, that must be written in the form of doclets. Nevertheless, this is enough to allow the use of all JBoss AOP features, and doesn't require a JDK 5 compiler.

This way, if you can't use a JDK 5 compiler to compile your code, you should stick with the annotation compiler. It will process your application's bytecodes, transforming doclets into annotations. The result of this transformation is that your doclets will become viewable by JBoss AOP as if they were regular JDK 5 annotations.

In the next sections, we will see what is the format your doclets need to follow in order to be transformed into annotations, and how to use the annotation compiler.

8.1. Annotations with JDK 1.4.2

In JDK 5, annotations must map to an annotation type, which is defined using the following syntax:

```
package com.mypackage;

public @interface MyAnnotation
{
   String myString();
   int myInteger();
}
```

Similarly, annotations for use with the annotation compiler also need to map to a type. And this one is defined in exactly the same way as above, with the important difference that '@interface' is replaced by 'interface'. i.e. the simulated annotation type is a normal Java interface:

```
package com.mypackage;

public interface MyAnnotation extends org.jboss.lang.Annotation
```

```
{
   String myString();
   int myInteger();
}
```

One difference from AOP 1.x is that the interfaces defining the annotations must now extend `org.jboss.lang.Annotation`, this is because JBoss Retro is now the primary mechanism for using annotations in JDK 1.4.2.

The syntax for using annotations in JDK 1.4.2 is almost exactly the same as JDK 5 annotations except for these subtle differences:

- they are embedded as doclet tags
- You use a double at sign, i.e. '@@'
- You MUST have a space after the tag name otherwise you will get a compilation error. (This is the quirkiness of the QDox doclet compiler used to compile the annotations.')
- You cannot import the annotation type, you must use the fully qualified name of the interface.
- You can only annotate top-level and inner classes, and their constructors, methods and fields. Annotating anonyomus classes, local classes, and parameters for constructors or methods is not supported.
- You cannot specify default values for an annotation's value

This example shows an annotated class in JDK 1.4.2:

```
package com.mypackage;

/**
 * @@com.mypackage.MyAnnotation (myString="class", myInteger=5)
 */
public class MyClass
{
   /**
    * @@com.mypackage.MyAnnotation (myString="field", myInteger=4)
    */
   private String myField;

   /**
    * @@com.mypackage.MyAnnotation (myString="constructor", myInteger=3)
    */
   public MyClass()
   {
   }

   /**
    * @@com.mypackage.MyAnnotation (myString="method", myInteger=3)
    */
   public int myMethod()
   {
   }
}
```

The next aspect is the JDK1.4.2 version of the @Introduction exammple (Chapter 6). Notice the slight difference in the JDK 1.4.2 annotation: class values don't have the ".class" suffix:

```
package com.mypackage;

/*
 * @@org.jboss.aop.Aspect (scope = Scope.PER_VM)
 */
public class IntroAspect
{
   /*
    * @org.jboss.aop.Introduction (target=com.blah.SomeClass, \
          interfaces={java.io.Serializable})
    */
   public static Object pojoNoInterfacesIntro;
}
```

Now, look at the next example:

```
package com.mypackage;

import org.jboss.aop.introduction.AnnotationIntroduction;

/**
 * @@org.jboss.aop.InterceptorDef (scope=org.jboss.aop.advice.Scope.PER_VM)
 * @@org.jboss.aop.Bind (pointcut="all(com.blah.SomePOJO)")
 */
public class IntroducedAnnotationInterceptor implements Interceptor
{
   /**
    * @@org.jboss.aop.AnnotationIntroductionDef \
          (expr="method(* com.blah.SomePOJO->annotationIntroductionMethod())", \
          invisible=false, \
          annotation="@com.mypackage.MyAnnotation \
          (string='hello', integer=5, bool=true)")
    */
   public static AnnotationIntroduction annotationIntroduction;

   public String getName()
   {
      return "IntroducedAnnotationInterceptor";
   }

   public Object invoke(Invocation invocation) throws Throwable
   {
      return invocation.invokeNext();
   }
}
```

The code above is the jdk1.4.2 version equivalent to the @AnnotationIntroductionDef example we have seen in Chapter 6. Note that, in the version above, the reference to only uses one '@'. In addition,the value for its string parameter uses single quotes instead of double ones.

8.2. Enums in JDK 1.4.2

Another JDK 5 feature that JBoss AOP helps introduce to JBoss 1.4.2 are Enums. As an example we can look at the `org.jboss.aop.advice.Scope` enum that is used for the `@Aspect` annotation. Here is the JDK 5 version.

```
package org.jboss.aop.advice;

public enum Scope
{
   PER_VM, PER_CLASS, PER_INSTANCE, PER_JOINPOINT
}
```

And it's usage in JDK 5

```
package com.mypackage;

@Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
public class SomeAspect
{
}
```

The usage in JDK 1.4.2 is similar:

```
package com.mypackage;

/**
 * @@org.jboss.aop.Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
 */
public class SomeAspect
{
   //...
}
```

However the declaration of the enumeration is different in 1.4.2:

```
package org.jboss.aop.advice;

import java.io.ObjectStreamException;

public class Scope extends org.jboss.lang.Enum
{
   private Scope(String name, int v)
   {
      super(name, v);
   }

   public static final Scope PER_VM = new Scope("PER_VM", 0);
   public static final Scope PER_CLASS = new Scope("PER_CLASS", 1);
   public static final Scope PER_INSTANCE = new Scope("PER_INSTANCE", 2);
   public static final Scope PER_JOINPOINT = new Scope("PER_JOINPOINT", 3);

   private static final Scope[] values = {PER_VM, PER_CLASS, PER_INSTANCE, PER_JOINPOINT};

   Object readResolve() throws ObjectStreamException
   {
      return values[ordinal];
   }

}
```

To create your own enum class for use within annotations, you need to inherit from `org.jboss.lang.Enum`. Each

enum has two values, a String name, and an integer ordinal. The value used for the ordinal must be the same as it's index in the static array.

8.3. Using Annotations within Annotations

The annotation compiler allows you to use annotations within annotations. This is best illustrated with an example. The definitions of the annotation interfaces in JDK 1.4.2:

```
com.mypackage;

public interface Outer
{
   Inner[] values();
}
```

```
com.mypackage;

public interface Inner
{
   String str();
   int integer();
}
```

The annotations can be applied as follows

```
com.mypackage;

/**
 * @@com.mypackage.Outer ({@@com.mypackage.Inner (str="x", integer=1), \
       @@com.mypackage.Inner (str="y", integer=2)})
 */
public class Test
{
   Inner[] values();
}
```

8.4. Using the Annotation Compiler

In order to use the JDK 1.4.2 annotations you have to precompile your files with an annotation compiler.

To use the annotation compiler you can create a simple ant build.xml file

```
<?xml version="1.0" encoding="UTF-8"?>

<project default="run" name="JBoss/AOP">
   <target name="prepare">
```

Include the jars AOP depends on

```
     <path id="javassist.classpath">
```

```
         <pathelement path="../../../javassist.jar"/>
      </path>
      <path id="trove.classpath">
         <pathelement path="../../../trove.jar"/>
      </path>
      <path id="concurrent.classpath">
         <pathelement path="../../../concurrent.jar"/>
      </path>
      <path refid="jboss.common.core.classpath"/>
      <path refid="jboss.common.logging.spi.classpath"/>
      <path refid="jboss.common.logging.log4j.classpath"/>
      <path refid="jboss.common.logging.jdk.classpath"/>
         <pathelement path="../../../jboss-common.jar"/>
      </path>
      <path id="jboss.aop.classpath">
         <pathelement path="../../../jboss-aop.jar"/>
      </path>
      <path id="qdox.classpath">
         <pathelement path="../../../qdox.jar"/>
      </path>
      <path id="classpath">
         <path refid="javassist.classpath"/>
         <path refid="trove.classpath"/>
         <path refid="jboss.aop.classpath"/>
      <path refid="jboss.common.core.classpath"/>
      <path refid="jboss.common.logging.spi.classpath"/>
      <path refid="jboss.common.logging.log4j.classpath"/>
      <path refid="jboss.common.logging.jdk.classpath"/>
         <path refid="concurrent.classpath"/>
         <path refid="qdox.classpath"/>
      </path>
```

Define the ant task that does the annnotation compilation

```


      <taskdef
         name="annotationc"
         classname="org.jboss.aop.ant.AnnotationC"
         classpathref="jboss.aop.classpath"/>
   </target>

   <target name="compile" depends="prepare">]></programlisting>
         Compile the source files
         <programlisting><!CDATA[
      <javac srcdir="."
         destdir="."
         debug="on"
         deprecation="on"
         optimize="off"
         includes="**">
            <classpath refid="classpath"/>
      </javac>
```

Run the annotation compiler

```


      <annotationc compilerclasspathref="classpath" classpath="." bytecode="true">
         <src path="."/>
      </annotationc>
   </target>
</project>
```

The `org.jboss.aop.ant.AnnotationC` ant task takes several parameters.

- `compilerclasspath`, `compilerclasspathref`, `classpath`, `classpathref` and `verbose` - These have the same meaning as in the JBoss Retro task.
- `bytecode` - The default is false. If true the annotation compiler instruments (i.e. modifies) the class files with the annotations. In this case, the classes must be precompiled with javac and `classpath` or `classpathref` must be specified.
- `xml` - Default is false. If true an xml file is generated containing information of how to attach the annotations at a later stage in the aop process.
- `output` - If `xml`="true", this lets you specify the name you would like for the generated xml file. The default name is `metadata-aop.xml`

You cannot currently specify both `bytecode` and `xml`.

You can also run `org.jboss.aop.ant.AnnotationC` from the command line, you need

```
$ java -cp <all the JBoss AOP jars and the directory containing files we want to AOP> \
      org.jboss.aop.annotation.compiler.AnnotationCompiler \
      [-xml [-o outputfile ]] [-bytecode]<files>+
```

In the /bin folder of the distribution we have provided batch/script files to make this easier. It includes all the aop libs for you, so you just have to worry about your files. The usage is:

```
$ annotationc <classpath> [-verbose] [-xml [-o outputfile]] [-bytecode] <dir_or_file>+
```

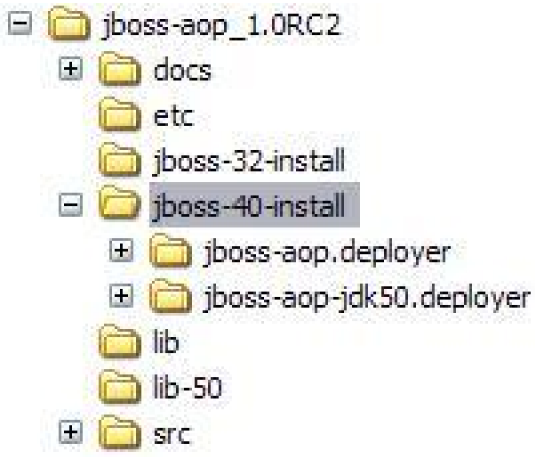
- `classpath` - path to your classes and any jars your code depends on

The other parameters are the same as above.

9

Installing

This section defines how to install JBoss AOP standalone, within JBoss 4.0.x, JBoss 4.2.x and within JBoss 5.x



9.1. Installing Standalone

There's nothing really to install if you're running outside the JBoss application server. If you are using JDK 1.4.x, use the libraries under the `lib-14/` directory to build your JBoss AOP applications. If you're using JDK 5.0, use the libraries under `lib-50/`.

9.2. Installing with JBoss 4.0.x amd JBoss 4.2.x Application Server for JDK 5

To install JBoss AOP in JBoss 4.0.x or JBoss 4.2.x Application Server: with JDK 5, there is an ant build script to install into the application server. It lives in `jboss-40-install/jboss-aop-jdk50.deployer/build.xml`. Modify `jboss-40-install/jboss-aop-jdk50.deployer/jboss.properties` to point to the the root of your JBoss installation and specify the application server configuration you want to upgrade. These are the steps taken by the ant script:

1. Back up the existing `jboss-40-install/jboss-aop-jdk50.deployer` to `server/<config-name>/deploy/jboss-aop-jdk50.deployer.bak`
2. Copy the files from `jboss-40-install/jboss-aop-jdk50.deployer` over the files that already exist in your existing JBoss Application Server distribution under `server/<config-name>/deploy/jboss-aop-jdk50.deployer`
3. In JBoss 4.0.4.GA and later, move `server/<config-name>/deploy/jboss-aop-jdk50.deployer/javassist.jar` to `server/<config-name>/lib/javassist.jar`. Any existing javassist.jar in that location is copied to `server/`

<config-name>/deploy/jboss-aop-jdk50.deployer.bak/lib/javassist.bak

4. If you NOT upgrading from a previous AOP 2 distribution, open up server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-aspect-library-jdk50.jar and delete all classes and subpackages under org.jboss.aop. In AOP 2.0 we changed the packaging, these classes now exist inside server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-aop-as4-deployer.jar. Also, we delete any files that also exist in server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-standalone-aspect-library-jdk14.jar

9.3. Installing with JBoss 4.0.x Application Server for JDK 1.4

To install JBoss AOP in JBoss 4.0.x with JDK 1.4 (JBoss 4.2.x requires JDK 5), there is an ant build script to install into the application server. It lives in jboss-40-install/jboss-aop-jdk14.deployer/build.xml. Modify jboss-40-install/jboss-aop-jdk14.deployer/jboss.properties to point to the the root of your JBoss installation and specify the application server configuration you want to upgrade. These are the steps taken by the ant script:

1. Back up the existing jboss-40-install/jboss-aop.deployer (if upgrading from AOP 1.x) or jboss-40-install/jboss-aop-jdk14.deployer (if upgrading from AOP 2.x) to server/<config-name>/deploy/jboss-aop.deployer.bak
2. Copy the files from jboss-40-install/jboss-aop-jdk14.deployer to server/<config-name>/deploy/jboss-aop-jdk50.deployer
3. In JBoss 4.0.4.GA and later, move server/<config-name>/deploy/jboss-aop-jdk50.deployer/javassist.jar to server/<config-name>/lib/javassist.jar. Any existing javassist.jar in that location is copied to server/<config-name>/deploy/jboss-aop-jdk50.deployer.bak/lib/javassist.bak
4. In JBoss 4.0.4.GA and later, move server/<config-name>/deploy/jboss-aop-jdk50.deployer/jbossretro-rt.jar to server/<config-name>/lib/jbossretro-rt.jar. Any existing jbossretro-rt.jar in that location is copied to server/<config-name>/deploy/jboss-aop-jdk50.deployer.bak/lib/jbossretro-rt.bak
5. In JBoss 4.0.4.GA and later, move client/jbossretro-rt.jar to client/jbossretro-rt.bak.
6. In JBoss 4.0.4.GA and later, move client/javassist.jar to client/javassist.bak.
7. If you NOT upgrading from a previous AOP 2 distribution, open up server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-aspect-library-jdk50.jar and delete all classes and subpackages under org.jboss.aop. In AOP 2.0 we changed the packaging, these classes now exist inside server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-aop-as4-deployer.jar. Also, we delete any files that also exist in server/<config-name>/deploy/jboss-aop-jdk50.deployer/jboss-standalone-aspect-library-jdk14.jar

9.4. Installing with JBoss Application Server 5

JBoss 5 is not released yet, this section will be modified when the packaging is finalised, but all you should need to do is copy the contents of the jboss-50-install/lib directory into $JBOSS_ROOT/lib, and copy the contents of the jboss-50-install/jboss-aop-jboss5.deployer folder into $JBOSS_ROOT/server/<config-name>/deployers/jboss-aop-jboss5.deployer

10

Building and Compiling Aspectized Java

10.1. Instrumentation modes

JBoss AOP works by instrumenting the classes you want to run. This means that modifications to the bytecode are made in order to add extra information to the classes to hook into the AOP library. JBoss AOP allows for two types of instrumentation

- Precompiled - The classes are instrumented in a separate aop compilation step before they are run.
- Loadtime - The classes are instrumented when they are first loaded.

This chapter describes the steps you need to take to precompile your classes with the aop precompiler.

10.2. Ant Integration

JBoss AOP comes with an ant task that you can use for precompiling your classes with the aop precompiler. An example build.xml file is the basis for the explanation. (It is quite similar to the one used in the previous chapter.) There will be differences in the build.xml file if you are using JDK 1.4.2 or JDK 5.0, these are outlined below:

```
<?xml version="1.0" encoding="UTF-8"?>

<project default="compile" name="JBoss/AOP">
   <target name="prepare">
```

Define the source directory, and the directory to compile classes to. If you're not fussy, they can point to the same directory.

```
        <property name="src.dir" value="PATH TO YOUR SOURCE DIR">
        <property name="classes.dir" value="PATH TO YOUR DIR FOR COMPILED CLASSES">
```

Include the jars AOP depends on, these are common to all JDK's

```
     <path id="javassist.classpath">
        <pathelement path="../../../javassist.jar"/>
     </path>

     <path id="trove.classpath">
        <pathelement path="../../../trove.jar"/>
```

```
    </path>

    <path id="concurrent.classpath">
       <pathelement path="../../../concurrent.jar"/>
    </path>

    <path refid="jboss.common.core.classpath"/>
    <path refid="jboss.common.logging.spi.classpath"/>
    <path refid="jboss.common.logging.log4j.classpath"/>
    <path refid="jboss.common.logging.jdk.classpath"/>
       <pathelement path="../../../jboss-common.jar"/>
    </path>

    <path id="lib.classpath">
       <path refid="javassist.classpath"/>
       <path refid="trove.classpath"/>
       <path refid="jboss.aop.classpath"/>
       <path refid="jboss.common.core.classpath"/>
       <path refid="jboss.common.logging.spi.classpath"/>
       <path refid="jboss.common.logging.log4j.classpath"/>
       <path refid="jboss.common.logging.jdk.classpath"/>
       <path refid="concurrent.classpath"/>
    </path>
```

This snippet shows what to do for JDK 1.4. It will also work with JDK 5.0 if your classes do not use JDK 5.0 style annotations and enums:

```
    <!--                  JDK version 1.4.2                          -->
    <!-- Do not include this if using JDK 5 with annotations!!!!  -->

    <path id="jboss.aop.classpath">
       <pathelement path="../../../jboss-aop.jar"/>
    </path>

    <!--            JDK version 1.4.2 - END                         -->
```

This snippet shows what to do for JDK 5.0 if you are using JDK 5.0 annotations:

```
    <!--            JDK version 1.5                                 -->
    <!-- Do not include this if using JDK 1.4.2!!!!                 -->

    <path id="jboss.aop.classpath">
       <pathelement path="../../../jboss-aop-jdk50.jar"/>
    </path>

    <!--            JDK version 1.5 - END                           -->
```

(You should only use one of the two previous snippets for setting up jboss.aop.classpath)

Now we set up the full classpath of all the needed libraries:

```
    <path id="classpath">
       <path refid="lib.classpath"/>
       <path refid="jboss.aop.classpath"/>
    </path id="classpath">
```

Define the `org.jboss.aop.ant.AopC` ant aop precompiler task:

```
      <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
         classpathref="jboss.aop.classpath"/>
   </target>
```

```
   <target name="compile" depends="prepare">
```

Compile the files (from the source directory to the compiled classes directory:

```
      <javac srcdir="${src.dir}"
         destdir="${classes.dir}"
         debug="on"
         deprecation="on"
         optimize="off"
         includes="**">
         <classpath refid="classpath"/>
      </javac>
```

Now use the ant aop precompiler task, it reads the files from the

```
      <aopc compilerclasspathref="classpath" verbose="true">
         <classpath path="${classes.dir}"/>
         <src path="${classes.dir}"/>
         <include name="**/*.class"/>
         <aoppath path="jboss-aop.xml"/>
         <aopclasspath path="aspects.jar"/>
      </aopc>
   </target>
</project>
```

If you are using JDK 1.4.2 and wish to use annotations, you need to define the `org.jboss.aop.ant.AnnotationC` ant task, and run that BEFORE you invoke the `org.jboss.aop.ant.AopC` task. How to do this is shown in the previous chapter.

The `org.jboss.aop.ant.AopC` ant task takes several parameters.

- `compilerclasspath` or `compilerclasspathref` - These are interchangable, and represent the jars needed for the aop precompiler to work. The `compilerclasspath` version takes the paths of the jar files, and the `compilerclasspathref` version takes the name of a predefined ant path. They can be specified as attributes of `aopc`, as shown above. `compilerclasspath` can also be specified as a child element of `aopc`, in which case you can use all the normal ant functionality for paths (e.g. fileset).

- `classpath` or `classpathref` - Path to the compiled classes to be instrumented. The `classpath` version takes the path of the directory, and the `classpathref` version takes the name of a predefined ant path. They both be specified as attributes of `aopc`. `classpath` can also be specified as a child element of `aopc`, as shown above, in which case you can use all the normal ant functionality for paths (e.g. fileset). The full classpath of the underly-

ing java process will be classpath + compilerclasspath.

- `src` - A directory containing files to be transformed. You can use multiple src elements to specify more that one root directory for transformation.

- `include` - This is optional and it serves as a filter to pick out which files within `src` should be transformed. You can use wildcards within the `name` expression, and you can also use multiple `include` elements.

- `verbose` - Default is false. If true, verbose output is generated, which comes in handy for diagnosing unexpected results.

- `report` - Default is false. If true, the classes are not instrumented, but a report called `aop-report.xml` is generated which shows all classes that have been loaded that pertain to AOP, what interceptors and advices that are attached, and also what metadata that has been attached. One particularly useful thing is the unbounded section. It specifys all bindings that are not bound. It allows you to debug when you might have a typo in one of your XML deployment descriptors.

  Report generation works on the instrumented classes, so to get valid data in your report, you have to to make two passes with `aopc`. First you run `aopc` with `report="false"` to instrument the classes, and then you run `aopc` with `report="true"` to generate the report.

- `aoppath` - The path of the `*-aop.xml` file containing the xml configuration of your bindings. Files or Directories can be specified. If it is a directory, JBoss AOP will take all `aop.xml` files from that directory. This gets used for the `jboss.aop.path` optional system property which is described in the "Command Line" section. If you have more than one xml file, for example if you have both a "normal" `jboss-aop.xml` file, and a `metadata-aop.xml` file having used the JDK 1.4.2 annotation compiler, you can specify these as follows:

```
<aoppath>
<pathelement path="jboss-aop.xml"/>
<pathelement path="metadata-aop.xml"/>
<pathelement path="xmldir"/>
</aoppath>
```

- `aopclasspath` - This should mirror your class path and contain all JARs/directories that may have annotated aspects (Ses Chapter "Annotated Bindings"). The AOPC compiler will browser each class file in this path to determine if any of them are annotationed with `@Aspect`. This gets used for the `jboss.aop.class.path` optional system property which is described in the "Command Line" section. If you have more than one jar file, you can specify these as follows:

```
<aopclasspath>
<pathelement path="aspects.jar"/>
<pathelement path="foo.jar"/>
</aopclasspath>
```

- `maxsrc` - The ant task expands any directories in `src` to list all class files, when creating the parameters for the java command that actually performs the compilation. On some operating systems there is a limit to the length of vaid command lines. The default value for `maxsrc` is 1000. If the total length of all the files used is greater than `maxsrc`, a temporary file listing the files to be transformed is used and passed in to the java command instead. If you have problems running the `aopc` task, try setting this value to a value smaller than 1000.

10.3. Command Line

To run the aop precompiler from the command line you need all the aop jars on your classpath, and the class files you are instrumenting must have everything they would need to run in the java classpath, including themselves, or the precompiler will not be able to run.

The `jboss.aop.path` optional system property points to XML files that contain your pointcut, advice bindings, and metadata definitions that the precompiler will use to instrument the .class files. The property can have one or files it points to delimited by the operating systems specific classpath delimiter (';' on windows, ':' on unix). Files or Directories can be specified. If it is a directory, JBoss AOP will take all `aop.xml` files from that directory.

The `jboss.aop.class.path` optional system property points to all JARs or directories that may have classes that are annotated as `@Aspect` (See Chapter "Annotated Bindings"). JBoss AOP will browse all classes in this path to see if they are annotated. The property can have one or files it points to delimited by the operating systems specific classpath delimiter (';' on windows, ':' on unix). Note for this to have effect with JDK 1.4, you first have to run the annotation compiler with bytecode=true.

It is invoked as:

```
$java -classpath ... [-Djboss.aop.path=...] [-Djboss.aop.class.path=...] \
                     org.jboss.aop.standalone.Compiler <class files or directories>
```

In the /bin folder of the distribution we have provided batch/script files to make this easier. It includes all the aop libs for you, so you just have to worry about your files. The usage for JDK 1.4 is:

```
$ aopc <classpath> [-aoppath ...] [-aopclasspath ...] [-report] [-verbose] \
      <class files or directories>+
```

And for JDK 5:

```
$ aopc15 <classpath> [-aoppath ...] [-aopclasspath ...] [-report] [-verbose] \
      <class files or directories>+
```

- `classpath` - path to your classes and any jars your code depends on

The other parameters are the same as above.

11

Running Aspectized Applications

This section will show you how to run JBoss AOP with standalone applications and how to run it integrated with the JBoss application server.

11.1. Loadtime, Compiletime and HotSwap Modes

There are 3 different modes to run your aspectized applications. Precompiled, loadtime or hotswap. JBoss AOP needs to weave your aspects into the classes which they aspectize. You can choose to use JBoss AOP's precompiler to accomplish this (Compiletime) or have this weavining happen at runtime either when the class is loaded (Loadtime) or after it (HotSwap).

Compiletime happens before you run your application. Compiletime weaving is done by using the JBoss AOP precompiler to weave in your aspects to existing .class files. The way it works is that you run the JBoss AOP precompiler on a set of .class files and those files will be modified based on what aspects you have defined. Compiletime weaving isn't always the best choice though. JSPs are a good instance where compiletime weaving may not be feasible. It is also perfectly reasonable to mix and match compile time and load time though. If you have load-time transformation enabled, precompiled aspects are not transformed when they are loaded and ignored by the classloader transformer.

Loadtime weaving offers the ultimate flexibility. JBoss AOP does not require a special classloader to do loadtime weaving, but there are some issues that you need to think about. JDK 1.4 does not have a standard simple way of transforming/instrumenting classes at runtime, so what JBoss AOP does is have a way to modify `java.lang.ClassLoader.class` to add the appropriate hooks. Its pretty simple. Take a look at the source under `org.jboss.aop.hooks` package and you'll see what we're doing is not that magical at all. Although this JDK 1.4 works with JDK 5, JDK5 actually has a simple standard mechanism of hooking in a class transformer through the `-javaagent`. JBoss AOP an additional load-time transformer that can hook into classloading via this standard mechanism.

Load-time weaving also has other serious side effects that you need to be aware of. JBoss AOP needs to do the same kinds of things that any standard Java profiling product needs to do. It needs to be able to process bytecode at runtime. This means that boot can end up being significantly slowed down because JBoss AOP has to do a lot of work before a class can be loaded. Once all classes are loaded though, load-time weaving has zero effect on the speed of your application. Besides boottime, load-time weaving has to create a lot of Javassist datastructure that represent the bytecode of a particular class. These datastructures consume a lot of memory. JBoss AOP does its best to flush and garbage collect these datastructures, but some must be kept in memory. We'll talk more about this later.

HotSwap weaving is a good choice if you need to enable aspects in runtime and don't want that the flow control of your classes be changed before that. When using this mode, your classes are instrumented a minimum necessary before getting loaded, without affecting the flow control. If any joinpoint becomes intercepted in runtime due to a

dynamic AOP operation, the affected classes are weaved, so that the added interceptors and aspects can be invoked. As the previous mode, hot swap contains some drawbacks that need to be considered.

11.2. Regular Java Applications

JBoss AOP does not require an application server to be used. Applications running JBoss AOP can be run standalone outside of an application server in any standard Java application. This section focuses on how to run JBoss AOP applications that don't run in the JBoss application server.

11.2.1. Precompiled instrumentation

Running a precompiled aop application is quite similar to running a normal java application. In addition to the classpath required for your application you need to specify the files required for aop:

- `javassist.jar`
- `trove.jar`
- `concurrent.jar`
- `jboss-common.jar`
- `jboss-aop.jar`
- or `jboss-aop-jdk50.jar`

- depending on if you are using JDK 1.4 (jboss-aop.jar) or JDK 5.0 (jboss-aop-jdk50.jar)

JBoss AOP finds XML configuration files in these two ways:

- You tell JBoss AOP where the XML files are. Set the `jboss.aop.path` system property. (You can specify multiple files or directories separated by ':' (*nix) or ';' (Windows), i.e. `-Djboss.aop.path=jboss-aop.xml;metadata-aop.xml`) If you specify a directory, all `aop.xml` files will be loaded from there as well.
- Let JBoss AOP figure out where XML files are. JBoss AOP will look for all XML files that match this pattern `/META-INF/jboss-aop.xml`. So, if you package your jars and put your JBoss AOP XML files within `/META-INF/jboss-aop.xml`, JBoss AOP will find these files.

If you are using annotated bindings (See Chapter "Annotated Bindings"), you must tell JBoss AOP which JARS or directories that may have annotated @Aspects. To do this you must set the `jboss.aop.class.path` system property. (You can specify multiple jars or directories separated by ':' (*nix) or ';' (Windows), i.e. `-Djboss.aop.class.path=aspects.jar;classes`)

So to run a precompiled AOP application, where your jboss-aop.xml file is not part of a jar, you enter this at a command prompt:

```
$ java -cp=<classpath as described above> -Djboss.aop.path=<path to jboss-aop.xml> \
        -Djboss.aop.class.path=aspects.jar
```

```
        com.blah.MyMainClass
```

To run a precompiled AOP application, where your application contains a jar with a META-INF/jboss-aop.xml file, you would need to do this from the command-line:

```
$ java -cp=<classpath as described above> com.blah.MyMainClass
```

In the /bin folder of the distribution we have provided batch/script files to make this easier. It includes all the aop libs for you, so you just have to worry about your files. The usage for JDK 1.4 is:

```
$ run-precompiled classpath [-aoppath path_to_aop.xml] [-aopclasspath path_to_annotated] \
      com.blah.MyMainClass [args...]
```

For JDK 5:

```
$ run-precompiled15 classpath [-aoppath path_to_aop.xml] [-aopclasspath path_to_annotated] \
      com.blah.MyMainClass [args...]
```

If your application is not in a jar with a META-INF/jboss-aop.xml file, you must specify the path to your `*-aop.xml` files in the `-aoppath` parameter, and if your class comtains aspects configured via annotations (`@Aspect` etc.) you must pass in this classpath via the `-aopclasspath` parameter. (For JDK 1.4, you must have compiled the annotations first).

11.2.2. Loadtime

This section describes how to use loadtime instrumentation of classes with aop. The classes themselves are just compiled using Java, but are not precompiled with the aop precompiler. (If you want to use annotations with JDK 1.4, you will still need to use the JDK 1.4 Annotation Compiler). In the examples given if your classes are contained in a jar with a META-INF/jboss-aop.xml file, you would omit the `-Djboss.aop.path` system property.

11.2.2.1. Loadtime JDK 1.4

In order to do loadtime weaving of aspects with JDK 1.4, we had to massage `java.lang.ClassLoader`. `java.lang.ClassLoader` is modified to add hooks for class transformation before class loading. It is very similar to JDK 5's built in ability to define class transformers. What you have to do is generate a modification of `java.lang.ClassLoader` and add this class to the default bootstrap class path (bootclasspath) for your classes to get instrumented at loadtime. The classes used are dependent upon the VM. At present this custom classloader has only been tested with Sun's J2SE 1.4.x and 5.0. The steps to compile and use the custom classloader are shown below.

```
$ java -cp=<classpath as described above>  \
       org.jboss.aop.hook.GenerateInstrumentedClassLoader <output dir>
```

For the following example, the `aop boot classpath` should be the `output dir` specified above, followed by the jars needed for AOP, i.e. `javassist.jar`, `trove.jar`, `concurrent.jar`, `jboss-common.jar` and `jboss-aop.jar`. You separate the classpath elements as normal, with ';' (Windows) or ':' (Unix). The path to your classes should NOT be included here! You then use this `aop boot classpath` as the argument for `-Xbootclasspath` option as shown here:

```
$ java -Xbootclasspath/p:<aop boot classpath as described> \
      -Djboss.aop.path=<path to jboss-aop.xml> \
      -classpath <path to your classes> com.blah.MyMainClass
```

In the /bin folder of the distribution we have provided batch/script files to make this easier. It includes all the aop libs for you, so you just have to worry about your files:

```
$ run-load-boot classpath [-aoppath path_to_aop.xml] [-aopclasspath path_to_annotated] \
      com.blah.MyMainClass [args...]
```

The parameters have the same meaning as for the run-precompiled scripts. (Since this is for JDK 1.4, you must have compiled the annotations first). This script both creates the instrumented class loader and makes sure that that the JAVA_HOME environment variable has been set (Your job is to make sure it points to a 1.4 distribution!).

11.2.2.2. Loadtime with JDK 5

JDK 5.0 has a pluggable way of defining a class transformer via the `java.lang.instrument` package. JBoss AOP uses this mechanism to weave aspects at class load time with JDK 5. Using loadtime with JDK 5 is really easy. All you have to do is define an additional standard switch on the Java command line. `-javaagent:jboss-aop-jdk50.jar`. For these examples make sure that you use `jboss-aop-jdk50.jar` and not `jboss-aop.jar` in your classpath. Here's how run an AOP application in JDK 5.0 with loadtime instrumentation, where your jboss-aop.xml file is not part of a jar:

```
$ java -cp=<classpath as described above> -Djboss.aop.path=<path to jboss-aop.xml> \
      -javaagent:jboss-aop-jdk50.jar com.blah.MyMainClass
```

And to run an AOP application in JDK 5.0 with loadtime instrumentation, where your application contains a jar with a META-INF/jboss-aop.xml file:

```
$ java -cp=<classpath as described above> -javaagent:jboss-aop-jdk50.jar \
      com.blah.MyMainClass
```

In the /bin folder of the distribution we have provided batch/script files to make this easier. It includes all the aop libs for you, so you just have to worry about your files. The usage for JDK 5 is:

```
$ run-load15 classpath [-aoppath path_to_aop.xml] [-aopclasspath path_to_annotated] \
      com.blah.MyMainClass [args...]
```

The parameters have the same meaning as for the run-precompiled scripts.

If you invoke the previous `java` examples with ant, by using the ant `java` task, make sure that you set `fork="true"` in the ant `java` task. Failure to do so, causes the `java` task to execute in the same VM as ant which is already running. This means that the special classloader used to do the loadtime transformations does not replace the standard one, so no instrumentation takes place.

11.2.2.3. Loadtime using JRockit

In JRockit the -Xbootclass/p option does not work, so we cannot replace the classloader. Instead we plug natively into its JVM using vendor specific hooks to provide transformation when a class is loaded. All you have to do is define an additional switch on the Java command line. `-Xmanagement:class=org.jboss.aop.hook.JRockitClassPreProcessor` Here's how run an AOP application in JDK 1.4 with loadtime instrumentation, with JRockit:

```
$ java -cp=<classpath as described above> -Djboss.aop.path=<path to jboss-aop.xml> \
      -Xmanagement:class=org.jboss.aop.hook.JRockitClassPreProcessor com.blah.MyMainClass
```

The above will also work with JRockit 5.0, but this can also use the "normal" -javaagent switch.

11.2.2.4. Improving Loadtime Performance

Boss AOP needs to do the same kinds of things that any standard Java profiling product needs to do. It needs to be able to process bytecode at runtime before a class is loaded. JBoss AOP has to do a lot of work before a class can be loaded. This means that boot time can end up being significantly slowed down. Once all classes are loaded though, load-time weaving has zero effect on the speed of your application.

Besides boottime, load-time weaving has to create a lot of Javassist datastructures that represent the bytecode of a particular class. These datastructures consume a lot of memory. JBoss AOP does its best to flush and garbage collect these datastructures, but some must be kept in memory. This section focuses on how you can improve the performance of Loadtime weaving.

Increase the Java Heapspace
: In Java, when your application is getting close to eating up all of its memory/heapspace, the Java Garbage Collector starts to run more frequently and aggressively. When the GC starts running more often the performance of your application will suffer. JBoss AOP does its best to balance bootup speed vs. memory consumption, but it does require loading bytecode into Javassist datastructures so it can analyze and transform a class. For speed purposes, the datastructures are cached thus leading to the extra memory consumption. Javassist structures of non-transformed classes are placed a SoftReference cache, so they are GC'd when memory is running low. Transformed classes, however, are locked in the cache. Transformed classes are help in memory, as they may effect pointcut matching on classes that haven't been loaded yet.

  To increase your Heap size, use the standard `-Xmx` switch.

Filtering
: Filtering probably has the greatest effect on overall boot-time speed. If you've ever worked with a Java profiling product before, you probably noticed that it has an option to filter classes that you are not interested in profiling. THis can speed up performance of the tool. JBoss AOP has to analyze every class in the system to make sure it does not need to be transformed. THis is one reason why load-time weaving can be so slow. You can give JBoss AOP a lot of help by specifying sets of classes that do not need to be transformed.

To enable filtering, you can use the `jboss.aop.exclude` System Property. This System Property is a comma delimited list. The strings in the list can be package names and/or classnames. No wildcards are allowed. Packages/classes within this list will ignored by JBoss AOP.

```
java -Djboss.aop.exclude=org.jboss,org.apache ...
```

There is also a mirror opposite of exclude. The System Property `jboss.aop.include` overrides any thing specified with exclude.

Turn off optimizations

To increase overall runtime performance, JBoss AOP has to dynamically create a lot of extra code. If you turn off these optimizations, JBoss AOP can weave a bit quicker. There is a good chance, depending on your application that you will not even notice that these optimizations are turned off. The `jboss.aop.optimized` system property can be set to turn off optimizations.

```
java -Djboss.aop.optimized=false ...
```

Turn off pruning

JBoss AOP tries to aggressive prune cached Javassist structures. This may, may not have a tiny effect on performance. The `jboss.aop.prune` system property can be set to turn off pruning.

```
java -Djboss.aop.prune=false ...
```

-client/-server

Strangely enough, it seems that the -client VM switch is a little faster for JBoss AOP loadtime weaving that -server. If you are using the -server VM, trying switching to -client (the default).

bootclasspath Vs. JDK5 -javaagent

It is significantly slower to use the -javaagent vs. the JDK 1.4 bootclasspath approach. So, if you are using JDK5, use the JDK1.4 bootclasspath approach.

Ignore

A way to completely ignore classes from being instrumented. This overrides whatever you have set up using the include/exclude filters. The system property is `jboss.aop.ignore`, and you can use wildcards in the classnames. As for include/exclude you may specify a comma separated list of class name patterns. This following example avoids instrumenting the cglib generated proxies for hibernate:

```
java -Djboss.aop.ignore=*$$EnhancerByCGLIB$$*
```

11.2.3. HotSwap

The HotSwap feature allows bytecode of your classes to be weaved in runtime. This results in application flow control changes to your classes only when joinpoints become intercepted (to do this, use the dynamic aop funcionality

provided by JBoss AOP). This is a mode to be considered when you want to assure the flow control of your classes will be kept intact until a binding or a interceptor is added.

This mode is currently provided through the `java.lang.instrument.Instrumentation` hot swap functionality, which is part of the JVMTI (Java Virtual Machine Tool Interface) added in JDK5. So, you cannot run JBoss AOP in this mode when using a previous JDK version.

To enable HotSwap, you have to add an argument to the Java command line in a very similar way to the "Loadtime with JDK5" mode: `-javaagent:jboss-aop-jdk50.jar=-hotSwap`. The difference is that the `-hotSwap` argument was added to the agent parameter list.

This way, if your jboss-aop.xml file is contained in a jar file, run:

```
$ java -cp=<classpath as described above> -Djboss.aop.path=<path to jboss-aop.xml> \
                -javaagent:jboss-aop-jdk50.jar=-hotSwap com.blah.MyMainClass
```

And if your jboss-aop.xml file is contained in a jar, run the following command line:

```
$ java -cp=<classpath as described above> -javaagent:jboss-aop-jdk50.jar=-hotSwap \
                com.blah.MyMainClass
```

The `run-load15HotSwap` batch/script files contained in the /bin folder of the distribution are similar to the `run-load15` ones, described in the previous subsection. All aop libs are included in these script files. To use them, run:

```
$ run-load15 classpath [-aoppath path_to_aop.xml] [-aopclasspath path_to_annotated] \
                com.blah.MyMainClass [args...]
```

When hotswap is enabled, the prunning of classes is turned off. Therefore, if you try to configure the jboss.aop.prune option as `true`, this setup will be ignored.

As with the "Loadtime with JDK5" mode, the HotSwap mode results in a boot time delay. Besides this drawback, the execution of some dynamic aop operations may be slower than in the other modes, when classes need to be hot swapped. The available options to tune performance are the same as described in the "Improving Loadtime Performance" subsection, except the pruning of classes.

11.3. JBoss Application Server

JBoss AOP is integrated with JBoss 4.0.1+ and JBoss 3.2.6+ application server. The integration steps are different depending on what version of JBoss AS you are using and what JDK version you are using. It is also dependent on whether you want to use loadtime or compiletime instrumentation.

If you wish to use JBoss AS 4.0.0 you will need to use JBoss AOP 1.0 Final since later releases of JBoss AOP leverage improvements in JBoss's deployement architecture. If you do this please consult the docs for JBoss AOP 1.0 Final. It is recommended though that you use the latest versions of JBoss AOP and AS.

Based on what JDK you are on and what loadtime weaving option you want to you, you must configure JBoss AS differently.

11.3.1. Packaging AOP Applications

To deploy an AOP application in JBoss you need to package it. AOP is packaged similarly to SARs(MBeans). You can either deploy an XML file directly in the deploy/ directory with the signature *-aop.xml along with your package (this is how the base-aop.xml, included in the `jboss-aop.deployer` file works) or you can include it in the jar file containing your classes. If you include your xml file in your jar, it must have the file extension .aop and a jboss-aop.xml file must be contained in a META-INF directory, i.e. `META-INF/jboss-aop.xml`.

If you want to create anything more than a non-trivial example, using the .aop jar files, you can make any top-level deployment contain a .aop file containing the xml binding configuration. That is you can have a .aop file in an .ear file, or a .aop file in a war file etc. The bindings specified in the `META-INF/jboss-aop.xml` file contained in the .aop file will affect all the classes in the whole war!

To pick up a .aop file in an .ear file, it must be listed in the `.ear/META-INF/application.xml` as a java module, e.g.:

```
<?xml version='1.0'  encoding='UTF-8'?>
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN'
                             'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
    <display-name>AOP in JBoss example</display-name>
    <module>
        <java>example.aop</java>
    </module>
    <module>
        <ejb>aopexampleejb.jar</ejb>
    </module>
    <module>
        <web>
           <web-uri>aopexample.war</web-uri>
          <context-root>/aopexample</context-root>
       </web>
   </module>
</application>
```

Note that in newer versions of JBoss (>= 4.0.5), the contents of the .ear file are deployed in the order they are listed in the application.xml. When using loadtime weaving the bindings listed in the example.aop file must be deployed before the classes being advised are deployed, so that the bindings exist in the system before the ejb, servlet etc. classes are loaded. This is acheived by listing the .aop file at the start of the application.xml. Older versions of JBoss did not have this issue since the contained .aop files were deployed before anything else, and this still holds true for other types of archives such as .sar and .war files.

11.3.2. JBoss 4.x and JDK 1.4

JBoss AOP comes distributed with the JBoss 4.x Application Server. It is best to download the latest version and update your JBoss Application Server installation as described in the "Installing" chapter of this guide. Also, the

version distributed with JBoss 4.x Application Server may not be up to date. Check http://www.jboss.org/products/aop to see if a new version of JBoss AOP is available. To install a new version remove the jboss-aop.deployer file from the JBoss AS deploy/ directory and copy the jboss-aop.deployer directory from the JBoss AOP distribution to the JBoss AS deploy/ directory. This jboss-aop.deployer/ is in the JBoss AOP distribution within the jboss-40-install/ directory.

JBoss 4.x Application Server works out of the box with precompiled applications. If you want to do load-time transformations, you must edit jboss-aop.deployer/META-INF/jboss-service.xml as follows:

The `jboss-aop.deployer` file contains some MBeans that deploy and manage the AOP framework.

```
                <mbean code="org.jboss.aop.deployment.AspectManagerService"
        name="jboss.aop:service=AspectManager">
        <attribute name="EnableLoadtimeWeaving">false</attribute>
        <!-- These switches are only relevant when EnableLoadtimeWeaving is true -->
        <attribute name="SuppressTransformationErrors">true</attribute>
        <attribute name="Prune">true</attribute>
        <attribute name="Include">org.jboss.test</attribute>
        <attribute name="Exclude">org.jboss.</attribute>
        <attribute name="Optimized">true</attribute>
        <attribute name="Verbose">false</attribute>
     </mbean>

     <mbean code="org.jboss.aop.deployment.AspectDeployer"
        name="jboss.aop:service=AspectDeployer">
     </mbean>
```

By default, JBoss application server will not do load-time bytecode manipulation of AOP files. You can turn load-time on by setting the EnableLoadtimeWeaving attribute to true. If SuppressTransformationErrors is true failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not have all the classes a class references.

The next thing you have to do is create a new `java.lang.ClassLoader.class`. This new class will bytecode modify a copy of java.lang.ClassLoader.class to put in the appropriate hooks for loadtime transformation. There is a script in the bin/ directory of the JBoss-AOP distribution to create this class and also create a jar from it.

```
$ cd jboss-aop1.3.bin
$ create-pluggable-jboss-classloader.sh
```

This will create a jboss-classloader-transformer.jar. Copy this jar to the bin/ directory of your JBoss Application server distribution.

Next, you need to copy the jdk14-pluggable-instrumentor.jar from the lib-14/ directory of your JBoss AOP distribution to the bin/ directory of your JBoss application server installation. Next edit run.sh or run.bat (depending on what OS you're on) and add the following to the JAVA_OPTS environment variable

On Unix/linux edit run.sh (note the : separating the bootclasspath entries)

```
JAVA_OPTS="$JAVA_OPTS -Dprogram.name=%PROGNAME% \
-Xbootclasspath/p:jboss-classloader-transformer.jar:jdk14-pluggable-instrumentor.jar"
```

Note that if you are using a cygwin shell on Windows, you will need to use a semicolon instead of a colon to separate the bootclasspath jars:

```
JAVA_OPTS="$JAVA_OPTS -Dprogram.name=%PROGNAME% \
-Xbootclasspath/p:jboss-classloader-transformer.jar;jdk14-pluggable-instrumentor.jar"
```

On Windows edit run.bat (note the `;` separating the bootclasspath entries)

```
set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME% \
-Xbootclasspath/p:jboss-classloader-transformer.jar;jdk14-pluggable-instrumentor.jar
```

After modifying JAVA_OPTS and setting the EnableLoadtimeWeaving to true, then you should be ready to go.

11.3.3. JBoss 4.x and JDK 5

JBoss AS has special integration with JDK 5.0 to do loadtime transformations. This section explains how to use it.

JBoss AOP comes distributed with the JBoss 4.x Application Server. The version that comes with JBoss 4.x does not take advantage of JDK 5.0 features. It is best to install the jboss-aop-jdk50.deployer/ distribution into your JBoss Application Server install base. See the "Installing" chapter for more details.

If you want to do load-time transformations with JBoss 4 and JDK 5, there are two steps you must take.

The `jboss-aop-jdk50.deployer` file contains some MBeans that deploy and manage the AOP framework.

```
                <mbean code="org.jboss.aop.deployment.AspectManagerServiceJDK5"
      name="jboss.aop:service=AspectManager">
      <attribute name="EnableLoadtimeWeaving">true</attribute>
      <!-- only relevant when EnableLoadtimeWeaving is true -->
      <attribute name="SuppressTransformationErrors">true</attribute>
      <attribute name="Prune">true</attribute>
      <attribute name="Include">org.jboss.test</attribute>
      <attribute name="Exclude">org.jboss.</attribute>
      <attribute name="Optimized">true</attribute>
      <attribute name="Verbose">false</attribute>
   </mbean>

   <mbean code="org.jboss.aop.deployment.AspectDeployer"
      name="jboss.aop:service=AspectDeployer">
   </mbean>
```

By default, JBoss application server will not do load-time bytecode manipulation of AOP files. You can turn load-time on by setting the EnableLoadtimeWeaving attribute to true. If SuppressTransformationErrors is true failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not have all the classes a class references.

The next step is to copy the pluggable-instrumentor.jar from the lib-50/ directory of your JBoss AOP distribution to the bin/ directory of your JBoss AOP application server installation. Next edit run.sh or run.bat (depending on what OS you're on) and add the following to the JAVA_OPTS environment variable

```
set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME% -javaagent:pluggable-instrumentor.jar
```

After modifying JAVA_OPTS and setting the EnableLoadtimeWeaving to true, then you should be ready to go.

Note that the `code` attribute of the AspectManager mbean must be `org.jboss.aop.deployment.AspectManagerServiceJDK5` as that is what works with the -javaagent weaver.

11.3.4. JBoss 4.x and JRockit

To use loadtime transformations with JRockit we can instruct Jrockit to use its native classloader hooks. Note that with JRockit 1.4.2 this is your only option to do loadtime transformations.

If you are using JRockit 5.0 and you wish to use the JDK 5 features of JBoss AOP, you should replace jboss-aop.deployer with jboss-aop-jdk50.deployer as mentioned in "JBoss 4.x and JDK 5.0".

If you want to do load-time transformations with JBoss 4 and JRockit, there are two steps you must take.

The `jboss-aop.deployer` or `jboss-aop-jdk50.deployer` file (depending on which you are using) contains some MBeans that deploy and manage the AOP framework.

```
                <mbean code="org.jboss.aop.deployment.AspectManagerService"
        name="jboss.aop:service=AspectManager">
        <attribute name="EnableLoadtimeWeaving">true</attribute>
        <!-- only relevant when EnableLoadtimeWeaving is true -->
        <attribute name="SuppressTransformationErrors">true</attribute>
        <attribute name="Prune">true</attribute>
        <attribute name="Include">org.jboss.test</attribute>
        <attribute name="Exclude">org.jboss.</attribute>
        <attribute name="Optimized">true</attribute>
        <attribute name="Verbose">false</attribute>
     </mbean>

     <mbean code="org.jboss.aop.deployment.AspectDeployer"
        name="jboss.aop:service=AspectDeployer">
     </mbean>
```

By default, JBoss application server will not do load-time bytecode manipulation of AOP files. You can turn load-time on by setting the EnableLoadtimeWeaving attribute to true. If SuppressTransformationErrors is true failed bytecode transformation will only give an error warning. This flag is needed because sometimes a JBoss deployment will not have all the classes a class references.

The next step is to copy the jrockit-pluggable-instrumentor.jar from the lib-50/ directory of your JBoss AOP distribution to the bin/ directory of your JBoss AOP application server installation. Next edit run.sh or run.bat (depending on what OS you're on) and add the following to the JAVA_OPTS and JBOSS_CLASSPATH environment variables

```
# Setup JBoss sepecific properties
JAVA_OPTS="$JAVA_OPTS -Dprogram.name=$PROGNAME \
        -Xmanagement:class=org.jboss.aop.hook.JRockitPluggableClassPreProcessor"
JBOSS_CLASSPATH="$JBOSS_CLASSPATH:jrockit-pluggable-instrumentor.jar"
```

After modifying JAVA_OPTS, JBOSS_CLASSPATH and setting the EnableLoadtimeWeaving to true, then you should be ready to go.

Note that the `code` attribute of the AspectManager mbean must be `org.jboss.aop.deployment.AspectManagerService` as that is what works with the JRockit special hooks.

11.3.5. Improving Loadtime Performance in a JBoss AS Environment

The same rules apply to JBoss AS for tuning loadtime weaving performance as standalone Java. See the previous chapter on tips and hints. YOU CANNOT USE THE SAME SYSTEM PROPERTIES THOUGH! Switches like pruning, optimized, and include/exclude are configured through the jboss-aop.deployer/META-INF/jboss-service.xml file talked about earlier in this chapter. You should be able to figure out how to turn the switches on/off from the above documentation.

11.4. Scoping aop to the classloader

By default all deployments in JBoss are global to the whole application server. That means that any ear, sar, jar etc. that is put in the deploy directory can see the classes from any other deployed archive. Similarly, aop bindings are global to the whole virtual machine. This "global" visibility can be turned off per top-level deployment.

11.4.1. Deploying as part of a scoped classloader

How the following works may be changed in future versions of jboss-aop. If you deploy a .aop file as part of a scoped archive, the bindings etc. applied within the .aop/META-INF/jboss-aop.xml file will only apply to the classes within the scoped archive and not to anything else in the application server. Another alternative is to deploy -aop.xml files as part of a service archive (SAR). Again if the SAR is scoped, the bindings contained in the -aop.xml files will only apply to the contents of the SAR file. It is not currently possible to deploy a standalone -aop.xml file and have that attach to a scoped deployment. Standalone -aop.xml files will apply to classes in the whole application server.

11.4.2. Attaching to a scoped deployment

If you have an application using classloader isolation, as long as you have "prepared your classes" you can later attach a .aop file to that deployment. If we have a .ear file scoped using a jboss-app.xml file, with the scoped loader repository `jboss.test:service=scoped`:

```
<jboss-app>
  <loader-repository>
      jboss.test:service=scoped
  </loader-repository>
</jboss-app>
```

We can later deploy a .aop file containing aspects and configuration to attach that deployment to the scoped .ear. This is done using the `loader-repository` tag in the .aop files `META-INF/jboss-aop.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
   <loader-repository>jboss.test:service=scoped</loader-repository>
```

```
   <!-- Aspects and bindings -->
</aop>
```

This has the same effect as deploying the .aop file as part of the .ear as we saw previously, but allows you to hot deploy aspects into your scoped application.

12

Reflection and AOP

While AOP works fine for normal access to fields, methods and constructors, there are some problems with using the Reflection API for this using JBoss. The problems are:

- Intereptors/aspects bound to execution pointcuts for fields and constructors don't get invoked.
- Intereptors/aspects bound to caller pointcuts for methods and constructors don't get invoked.
- Reflection Methods such as `Class.getMethods()` and `Class.getField()` return extra JBoss AOP "plumbing" information.

12.1. Force interception via reflection

To address the issues with interceptors not being invoked when you use reflection, we have provided a reflection aspect. You bind it to a set of caller pointcuts, and it mounts the pre-defined interceptor/aspect chains. The jboss-aop.xml entries are:

```
<aspect class="org.jboss.aop.reflection.ReflectionAspect" scope="PER_VM"/>

<bind pointcut="call(* java.lang.Class->newInstance())">
   <advice name="interceptNewInstance" \
      aspect="org.jboss.aop.reflection.ReflectionAspect"/>
</bind>

<bind pointcut="call(* java.lang.reflect.Constructor->newInstance(java.lang.Object[]))">
   <advice name="interceptNewInstance" \
      aspect="org.jboss.aop.reflection.ReflectionAspect"/>
</bind>

<bind pointcut="call(* java.lang.reflect.Method->invoke(java.lang.Object, java.lang.Object[]))">
   <advice name="interceptMethodInvoke" \
      aspect="org.jboss.aop.reflection.ReflectionAspect"/>
</bind>

<bind pointcut="call(* java.lang.reflect.Field->get*(..))">
   <advice name="interceptFieldGet" \
      aspect="org.jboss.aop.reflection.ReflectionAspect"/>
</bind>

<bind pointcut="call(* java.lang.reflect.Field->set*(..))">
   <advice name="interceptFieldSet" \
      aspect="org.jboss.aop.reflection.ReflectionAspect"/>
</bind>
```

The `ReflectionAspect` class provides a few hooks for you to override from a subclass if you like. These methods described below.

```
protected Object interceptConstructor(
             Invocation invocation,
             Constructor constructor,
             Object[] args)
             throws Throwable;
```

Calls to `Class.newInstance()` and `Constructor.newInstance()` end up here. The default behavior is to mount any constructor execution or caller interceptor chains. If you want to override the behaviour, the parameters are:

- `invocation` - The invocation driving the chain of advices.
- `constructor` - The constructor being called
- `args` - the arguments being passed in to the constructor (in the case of Class.newInstance(), a zero-length array since it takes no parameters)

```
protected Object interceptFieldRead(
   Invocation invocation,
   Field field,
   Object instance)
   throws Throwable;
```

Calls to `Field.getXXX()` end up here. The default behavior is to mount any field read interceptor chains. If you want to override the behaviour, the parameters are:

- `invocation` - The invocation driving the chain of advices.
- `field` - The field being read
- `instance` - The instance from which we are reading a non-static field.

```
protected Object interceptFieldWrite(
   Invocation invocation,
   Field field,
   Object instance,
   Object arg)
   throws Throwable;
```

Calls to `Field.setXXX()` end up here. The default behavior is to mount any field write interceptor chains. If you want to override the behaviour, the parameters are:

- `invocation` - The invocation driving the chain of advices.

- `field` - The field being written
- `instance` - The instance on which we are writing a non-static field.
- `arg` - The value we are setting the field to

```
protected Object interceptMethod(
             Invocation invocation,
             Method method,
             Object instance,
             Object[] args)
             throws Throwable;
```

Calls to `Method.invoke()` end up here. The default behavior is to mount any method caller interceptor chains (method execution chains are handled correctly by default). If you want to override the behaviour, the parameters are:

- `invocation` - The invocation driving the chain of advices.
- `method` - The method being invoked
- `instance` - The instance on which we are invoking a non-static method.
- `args` - Values for the method arguments.

12.2. Clean results from reflection info methods

The `ReflectionAspect` also helps with getting rid of the JBoss AOP "plumbing" information. You bind it to a set of caller pointcuts, using the followingjboss-aop.xml entries :

```
<bind pointcut="call(* java.lang.Class->getInterfaces())">
   <advice name="interceptGetInterfaces" \
      aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
</bind>

<bind pointcut="call(* java.lang.Class->getDeclaredMethods())">
   <advice name="interceptGetDeclaredMethods" \
      aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
</bind>

<bind pointcut="call(* java.lang.Class->getDeclaredMethod(..))">
   <advice name="interceptGetDeclaredMethod" \
      aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
</bind>

<bind pointcut="call(* java.lang.Class->getMethods())">
   <advice name="interceptGetMethods" \
      aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
</bind>

<bind pointcut="call(* java.lang.Class->getMethod(..))">
```

```
      <advice name="interceptGetMethod" \
         aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
   </bind>

   <bind pointcut="call(* java.lang.Class->getDeclaredFields())">
      <advice name="interceptGetDeclaredFields" \
         aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
   </bind>

   <bind pointcut="call(* java.lang.Class->getDeclaredClasses())">
      <advice name="interceptGetDeclaredClasses" \
         aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
   </bind>

   <bind pointcut="call(* java.lang.Class->getDeclaredField(..))">
      <advice name="interceptGetDeclaredField" \
         aspect="org.jboss.test.aop.reflection.ReflectionAspectTester"/>
   </bind>
```

This way the calls to `Class.getMethods()` etc. only return information that is present in the "raw" class, by filtering out the stuff added to the class by JBoss AOP.

13

JBoss AOP IDE

13.1. The AOP IDE

JBoss AOP comes with an Eclipse plugin that helps you define interceptors to an eclipse project via a GUI, and to run the appication from within Eclipse. This is a new project, and expect the feature set to grow quickly!

13.2. Installing

You install the JBoss AOP IDE in the same way as any other Eclipse plugin.

- Make sure you have Eclipse 3.0.x installed, and start it up.
- Select Help > Software Updates > Find and Install in the Eclipse workbench.
- In the wizard that opens, click on the "Search for new features to install" radio button, and click Next.
- On the next page you will need to add a new update site for JBossIDE. Click the "New Remote Site.." button.
- Type in "JBossIDE" for the name, and "http://jboss.sourceforge.net/jbosside/updates" for the URL, and click OK.
- You should see a new site in the list now called JBossIDE. click the "+" sign next to it to show the platforms available.
- Now, depending if you just want to install the AOP IDE (if you don't know what JBoss-IDE is, go for this set of options):

  - Check the "JBoss-IDE AOP Standalone" checkbox.
  - In the feature list you should check the "JBoss-IDE AOP Standalone 1.0" checkbox.

  If you have JBoss-IDE installed, or want to use all the other (non-AOP) features of JBoss-IDE:

  - If you don't have JBossIDE installed, check the "JBoss-IDE 1.4/Eclipse 3.0" checkbox.
  - Check the "JBoss-IDE AOP Extension" checkbox.
  - In the feature list you should check the "JBoss-IDE AOP Extension 1.0" checkbox, and the JBoss-IDE (1.4.0) checkbox if you don't have JBossIDE installed.

- At this point you should only need to accept the license agreement(s) and wait for the install process to finish.

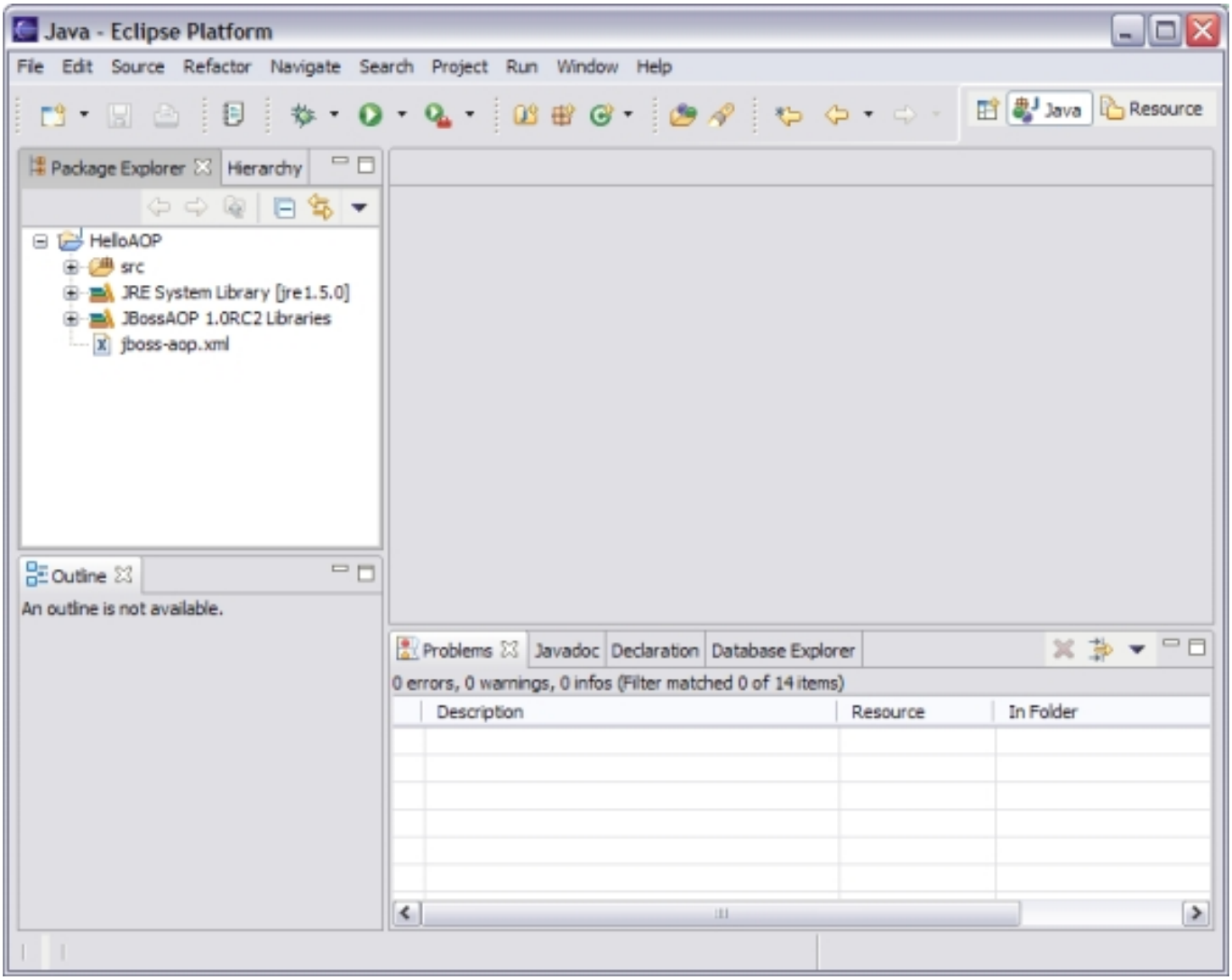
13.3. Tutorial

This tutorial is meant to guide you through creating a new AOP project in eclipse using the AOP extension to JBossIDE. It assumes that you have some working knowledge of AOP, and Java.. and possibly some minimal experience dealing with eclipse as well.

13.3.1. Create Project

- From eclipse's main menu, you can click on the File Menu, and under it, New > Project...
- Double click on JBoss AOP Project under the JBossAOP folder
- In the Project Name text box, let's enter `HelloAOP`.
- `Use Default` should be fine for the project location. (If you want to use an external location, make sure there are no spaces in the path.)
- Click `Finish`

At this point, your eclipse workbench should look something like this:



13.3.2. Create Class

Next step is to create a normal Java class.

- Right click on the "src" directory in the Package Explorer and in the menu, click New > Class.
- The only thing you should need to change is the Name of the class. Enter `HelloAOP` without quotes into the Name textbox, and click `Finish`

Modify the code for your class so it loks like

```
public class HelloAOP {

        public void callMe ()
        {
                System.out.println("AOP!");
        }

        public static void main (String args[])
        {
                new HelloAOP().callMe();
```

```
        }
}
```

13.3.3. Create Interceptor

Next we want to create an interceptor to the class.

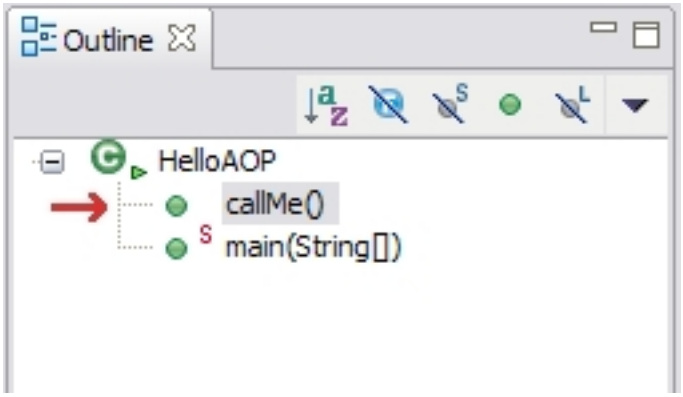
- Right click on the "src" directory in the Package Explorer and in the menu, click New > Class. In the resulting dialog:
  - Name the class `HelloAOPInterceptor`
  - Add `org.jboss.aop.advice.Interceptor` to the list of interceptors.

Then modify the class so it looks like:

```
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;

public class HelloAOPInterceptor implements Interceptor {

        public String getName() {
                return "HelloAOPInterceptor";
        }

   //We renamed the arg0 parameter to invocation
        public Object invoke(Invocation invocation) throws Throwable {
                System.out.print("Hello, ");
      //Here we invoke the next in the chain
                return invocation.invokeNext();
        }
}
```

13.3.4. Applying the Interceptor

In order to apply your Interceptor to the callMe() method, we'll first need to switch back to the `HelloAOP.java` editor. Once the editor is active, you should be able to see the callMe() method in the Outline view (If you cannot see the outline view, go to Window > Show View > Outline).



Right click on this method, and click JBoss AOP > Apply Interceptor(s)... A dialog should open, with a list of available Interceptors. Click on `HelloAOPInterceptor`, and click `Finish`.

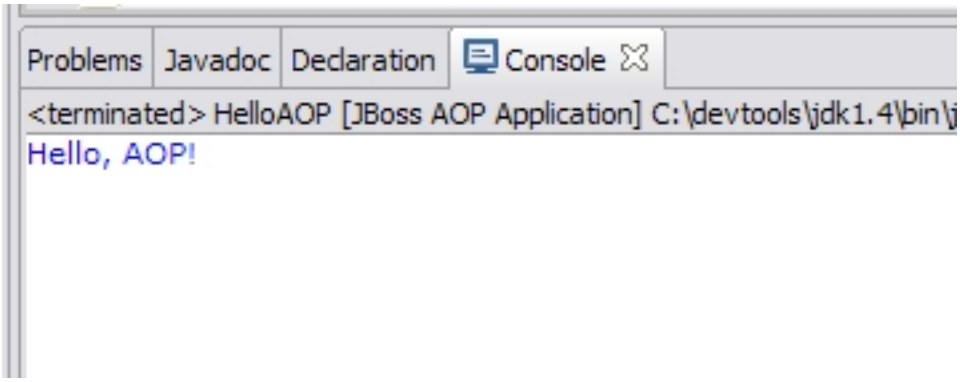
You should see in your Package Explorer that the file "jboss-aop.xml" now exists under your project root.

13.3.5. Running

Now all that's left is running the application! Similar to running a normal Java Application from Eclipse, you must create a Run Configuration for your project.

- From the Run menu of eclipse, and choose "Run..."
- In the dialog that opens, you should see a few choices in a list on the left. Double click on "JBoss AOP Application".
- Once it is finished loading, you should have a new Run Configuration under JBoss AOP Application called "Hello AOP".
- Click the "Run" button

The Eclipse console should now say: `Hello, AOP!`, where the `Hello,` bit has been added by the interceptor.

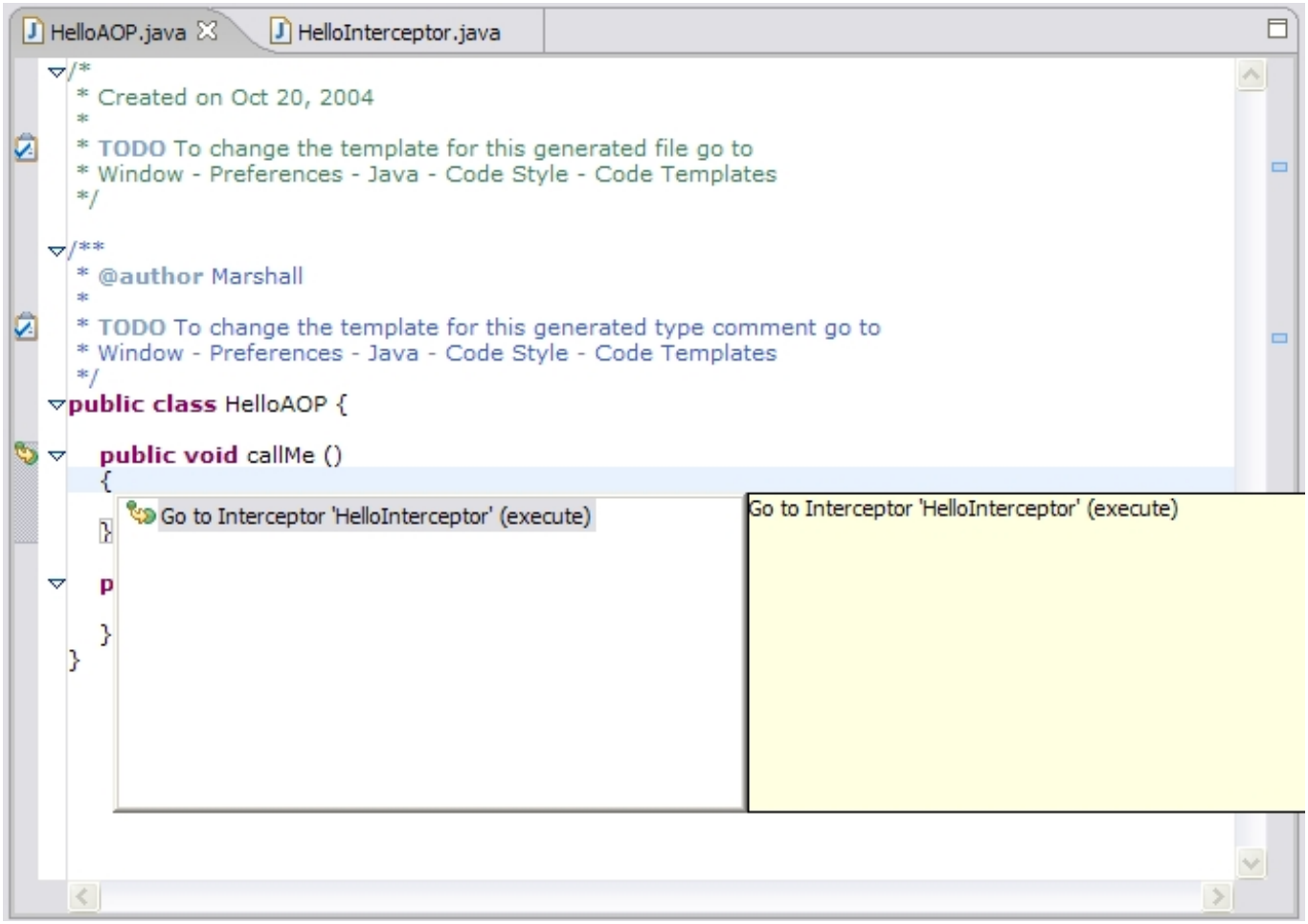


13.3.6. Navigation

In the real world, when developing AOP application across a development team, you can expect it will be hard to understand when and where aspects are applied in your codebase. JBoss-IDE/AOP has a few different strategies for notifying developers when an aspect is applied to a certain part of code.

13.3.6.1. Advised Markers

A marker in eclipse is a small icon that appears on the left side of the editor. Most developers are familiar with the Java Error and Bookmark markers. The AOP IDE provides markers for methods and fields which are intercepted. To further facilitate this marking, anytime the developer presses Ctrl + 1 (the default key combination for the Eclipse Quick Fix functionality)), a list of interceptors and advice will be given for that method or field. This makes navigation between methods and their interceptors extremeley easy!

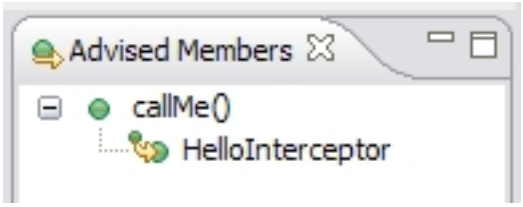


13.3.6.2. The Advised Members View

The Advised Members view gives the developer an overview of every single method and field in the current class that is advised by an Aspect or Interceptor. Let's have a look.

- From the Eclipse main menu, click on Window > Show View > Other...
- In the window that opens, you should see a folder called "JBoss AOP". Press the "+" to expand it.
- Double click on "Advised Members"

Once you've done this, you should now make sure you are currently editing the `HelloAOP` class we created in the last tutorial. Once you have that class open in an editor, you should see something similar to this in the Advised Members view:



Here we see that the method "callMe()" is intercepted by the interceptor `HelloInterceptor`. Double clicking on `HelloInterceptor` will take you straight to it. This view is similar to the Outline view, except it only shows mem-

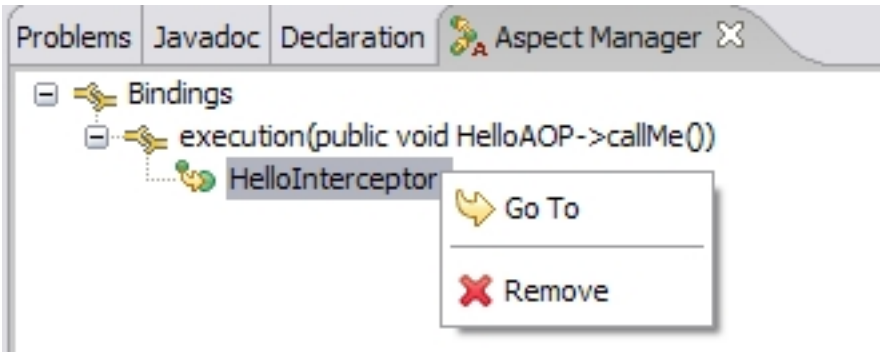
bers in your class which are intercepted.

13.3.6.3. The Aspect Manager View

The Aspect Manager View is a graphical representation of the AOP descriptor file (jboss-aop.xml). It allows you to remove an Interceptor or advice from a pointcut, as well as apply new Interceptors and Advice to existing pointcuts.

- From the Eclipse main menu, click on Window > Show View > Other...
- In the window that opens, you should see a folder called "JBoss AOP". Press the "+" to expand it.
- Double click on "Aspect Manager"

Under Bindings, you'll notice that a pointcut is already defined that matches our "callMe()" method, and our `HelloInterceptor` is directly under it. Right Click on `HelloInterceptor` will provide you with this menu:



You can remove the interceptor, or jump to it directly in code. If you right click on the binding (pointcut) itself, you'll be able to apply more interceptors and advice just like when right clicking on a field or method in the outline view. You can also remove the entire binding altogether (which subsequently removes all child interceptors and advice, be warned)