JBoss AOP Reference Documentation

Version: 1.0

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
```

Chapter 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.

Chapter 2. Implementing Aspects

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 sections walks through implementing aspects.

2.2. Invocation Object

Invocation objects are the runtime encapsulation of their joinpoint. They contain runtime information of their joinpoint (args, java.lang.reflect.*, etc..), and they also drive the flow of aspects.

Table 2.1. Invocation class types

Class	Description
org.jboss.aop.joinpoint.MethodInvocation	Method execution. Created and used when a method is intercepted.
org.jboss.aop.joinpoint.ConstructorInvocation	Constructor execution. Created and used when a constructor is intercepted.
org.jboss.aop.joinpoint.FieldInvocation	Field execution. This is an abstract base class that encapsulates field access.
org.jboss.aop.joinpoint.FieldReadInvocation	Field read access. Extends FieldInvocation. Created when a field is read.
org.jboss.aop.joinpoint.FieldWriteInvocation	Field modification. Extends FieldInvocation. Created when a field is written to.
org.jboss.aop.joinpoint.MethodCalledByMethod	Caller pointcuts. This invocation object is allocated when you are using "call" pointcut expressions. This particular class encapsulates a method that is calling another method so that you can access the caller and callee.
org.jboss.aop.joinpoint.MethodCalledByConstructor	Caller pointcuts. This invocation object is allocated when you are using "call" pointcut expressions. This particular class encapsulates a constructor that is calling another method so that you can access the caller and callee.
org.jboss.aop.joinpoint.ConstructorCalledByMethod	Caller pointcuts. This invocation object is allocated when you are using "call" pointcut expressions. This particular class encapsulates a method that is calling a constructor so that you can access the caller and callee.
org.jboss.aop.joinpoint.ConstructorCalledByConstructor	Caller pointcuts. This invocation object is allocated when you are using "call" pointcut expressions. This particular class encapsulates a constructor that is calling a constructor so that you can access the caller and callee.

2.3. 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.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.

2.4. 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.

Method names can be overloaded for different invocation types. 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 concreate invocation type.

```
public class Aspect
 {
   public Object trace(MethodInvocation invocation) throws Throwable {
      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()");
      }
   }
}
```

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);
}
```

Chapter 3. Pointcut and Type Expressions

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.

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, the `$instanceof{}` expression can be used. Wildcards and annotations are not allowed within `$instanceof{}` expressions.

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

is allowed.

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

are NOT 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{}` in the class name.

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

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

```
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(..)
```

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(..)
```

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(..)
```

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 constructors 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 of a tagged `@javax.ejb.Entity` class.

```
* @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
```

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.

`ge(field expression)t`

```
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 call.

`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})
```

Chapter 4. XML Bindings

4.1. 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.

4.2. 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.

4.2.1. 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.

4.2.2. 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.

4.3. 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)+>
<!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>
```

4.4. 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.

4.4.1. 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.

4.4.2. 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 4.1. 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.
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.

4.4.3. 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 4.2. 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 4.3. Injecting AOP runtime constructs

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

4.4.3.1. Names

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

4.4.3.2. 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) {...}
}
```

4.4.4. 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.

4.5. 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.

4.6. bind

```
<bind pointcut="execution(void Foo->bar())">
     <interceptor-ref name="org.jboss.MyInterceptor/>
     <advice name="trace" aspect="org.jboss.MyAspect"/>
</bind>
```

In the above example, the MyInterceptor interceptor and the trace method 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.

advice

The `advice` tag takes a `name` attribute that should map to a method within the aspect class. The `aspect` attribute should be the name of the aspect definition.

4.7. 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>
```

4.8. 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>
```

4.9. introduction

4.9.1. 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 expres-

sions, 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

4.9.2. 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.

4.10. 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, bo
</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.

4.11. 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>
```

4.12. 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})">
```

4.13. 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">
```

4.14. 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())"/>
```

4.15. 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.

4.16. 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 Except

   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.

Chapter 5. 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 JBoss AOP uses an annotation compiler allowing you to create the same annotations in javadoc style comments.

The JDK 1.5 form has been used for the declarations of each annotation type shown below. For clarity both types of annotations are shown in the usage examples contained in this chapter. A point worth mentioning is that in JDK 1.5 annotations are part of language and can thus be imported, so that just the classname can be used. In JDK 1.4.2 the annotations are not part of "Java" so fully qualified classnames are needed. To keep things short and sweet the listings only import classes that are new for each listing.

5.1. @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 the "XML Bindings" chapter for a description of the various scopes.

In JDK 1.5 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)
}
```

And in JDK 1.4.2:

```
package com.mypackage;

/**
 * @@Aspect (scope = Scope.PER_VM)
 */
public class MyAspect
{
   public Object myAdvice(Invocation invocation)
   {
      return invocation.invokeNext();
   }
}
```

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>
```

5.2. @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.

5.2.1. Interceptor Example

In JDK 1.5 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();
   }
}
```

And in JDK 1.4.2:

```
package com.mypackage;

/**
 * @@org.jboss.aop.InterceptorDef (scope = org.jboss.aop.advice.Scope.PER_VM)
 * @@org.jboss.aop.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>
```

5.2.2. AspectFactory Example

In JDK 1.5 the `@IntroductionDef` 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
}
```

And in JDK 1.4.2:

```
```

```
package com.mypackage;

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

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

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

5.3. @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 in JDK 1.5:

```
   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!).

The same example in JDK 1.4.2:

```
package com.mypackage;

import org.jboss.aop.pointcut.Pointcut;

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

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

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>
```

5.4. @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 in JDK 1.5:

```
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();
   }

}
```

And in JDK 1.4.2:

```
package com.mypackage;

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

   /**
    * @@org.jboss.aop.Bind (pointcut="com.mypackage.MyAspect.fooMethods")
```

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

     /**
      * @@org.jboss.aop.Bind (pointcut="execution("* org.blah.Bar->someMethod())")
      */
     public Object otherAdvice(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>
```

5.5. @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 `@Introduction` in JDK 1.5:

```
   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;
   }
```

And in JDK 1.4.2:

```
   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;
   }
```

Notice the slight difference in the JDK 1.4.2 annotation, the class values don't have the ".class" suffix.

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>
```

5.6. @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
- 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` in JDK 1.5:

```
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);
   }
```

```
   }
```

Here's the JDK 1.4.2 version:

```
package com.mypackage;

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

/**
 * @@org.jboss.aop.Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
 */
public class IntroductionAspect
{
   /**
    * @org.jboss.aop.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 `Externalizable-Mixin` 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>
```

5.7. @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}) @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` in JDK 1.5:

```
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();
   }
}
```

And in JDK 1.4.2:

```
package com.mypackage;

/**
 * @@org.jboss.aop.InterceptorDef (scope = org.jboss.aop.advice.Scope.PER_VM)
 * @@org.jboss.aop.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.

5.8. @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 with JDK 1.5:

```
   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();
      }
```

```
}
```

And with JDK 1.4.2:

```
package com.mypackage;

import org.jboss.aop.TypeDef;
import org.jboss.aop.pointcut.Typedef;

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

   /**
    * @@org.jboss.aop.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>
```

5.9. @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 this in JDK 1.5:

```
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();
   }
}
```

And in 1.4.2:

```
package com.mypackage;

import org.jboss.aop.pointcut.CFlowStack;
/**
 * @@org.jboss.aop.Aspect (scope=org.jboss.aop.advice.Scope.PER_VM)
 */
public class CFlowAspect
{

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

   /**
```

```
     * @@org.jboss.aop.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>
```

5.10. @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 in JDK 1.5:

```
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;
   }
}
```

And the same in JDK 1.4.2:

```
package com.mypackage;

import org.jboss.aop.pointcut.DynamicCFlow;

/**
```

```
 * @org.jboss.aop.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 in JDK 1.5:

```
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();
   }
}
```

To use the dynamic cflow we just defined in JDK 1.5:

```
package com.mypackage;

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

5.11. @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` in JDK 1.5:

```
   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 use of `@AnnotationIntroductionDef` in JDK 1.4.2:

```
   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();
   }
}
```

Note that the reference to only uses one '@', 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>
```

Chapter 6. Dynamic AOP

6.1. 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.

6.2. 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/ap-

pend full stacks into the pre/post chains.

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

6.3. Preparation

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

Chapter 7. Annotation Compiler for JDK 1.4

7.1. Annotations with JDK 1.4.2

Annotations are only available in JDK 1.5, but using our annotation compiler you can acheive similar functionality with JDK 1.4.2.

Annotations must map to an annotation type, in JDK 1.5 they are defined as:

```
package com.mypackage;

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

Annotation types for use with the annotation compiler are defined in exactly the same way for JDK 1.4.2, with the important difference that '@interface' is replaced by 'interface'. i.e. the simulater annotation type is a normal Java interface:

```
package com.mypackage;

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

The syntax for using annotations in JDK 1.4.2 is almost exactly the same as JDK 1.5 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 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()
    {
    }
}
```

7.2. Enums in JDK 1.4.2

Another JDK 1.5 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 1.5 version.

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

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

And it's usage in JDK 1.5

```
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.

7.3. 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();
}
```

7.4. 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 id="jboss.common.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.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">
```

Compile the source files

```
   <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` or `compilerclasspathref` - These are interchangable, and represent the jars needed for the annotation compiler to work. The `compilerclasspath` version takes the paths of the jar files, and the `compilerclasspathref` version takes the name of a predefined ant path.
- `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 `classpath-ref` must be specified.
- `classpath` or `classpathref` - Path to the compiled classes to be instrumented with annotations, if `bytecode`="true". The `classpath` version takes the path of the directory, and the `classpathref` version takes the name of a predefined ant path.
- `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`
- `verbose` - Default is false. If true, verbose output is generated, which comes in handy for diagnosing unexpected results.

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:

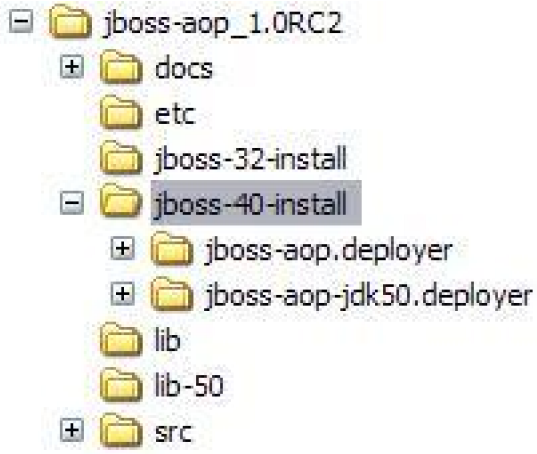
```
$ annotationc <classpath> [-verbose] [-xml [-o outputfile]] [-bytecode] <dir_or_file>+
```

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

The other parameters are the same as above.

Chapter 8. Installing

This section defines how to install JBoss AOP standalone, within JBoss 4.0 and within JBoss 3.2.6



8.1. 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/` directory to build your JBoss AOP applications. If you're using JDK 5.0, use the libraries under `lib-50/`.

8.2. Installing with JBoss 4.x Application Server

To install JBoss AOP in JBoss 4.x Application Server:

1. Delete `jboss-aop.deployer` file or directory from the existing JBoss Application Server distribution under `server/<config-name>/deploy`
2. From the JBoss AOP distribution, from the `jboss-40-install` directory copy `jboss-aop.deployer/` or `jboss-aop-jdk50.deployer` to the JBoss Application Server distribution under `server/<config-name>/deploy` depending on which JDK you are running with.

8.3. Installing with JBoss 3.2.6 Application Server

JBoss AOP only works with JBoss Application Server 3.2.6RC1+ and only with the precompiler. Load-time AOP is not supported at this time for JBoss 3.2.x application server. To install JBoss AOP to JBoss Application Server, copy all jars in the `jboss-32-install/` directory to the `server/<config-name>/lib`

1. copy all jars in the `jboss-32-install/` directory to the `server/<config-name>/lib`
2. Copy `etc/base-aop.xml` to the JBoss Application Server distribution under `server/<config-name>/deploy`.
3. Edit `server/<config-name>/conf/jboss-service.xml` to add the appropriate configuration defined in section 10.2 "JBoss Application Server".

Chapter 9. Building and Compiling Aspectized Java

9.1. 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.

9.2. 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 id="jboss.common.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.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 1.5 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-jdk15.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}"/>
         <aoppath path="jboss-aop.xml"/>
         <aopclasspath path="aspects.jar"/>
      </aopc>
```

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 underlying java process will be classpath + compilerclasspath.

- `src` - Which files to be transformed. Either a directory containing the files to be instrumented, or an ant fileset containing which files should be precompiled. In the example above, we specified a directory so all our classes are passed in to the precompiler.

- `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.

- `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.

9.3. 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 1.5:

```
$ 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.

Chapter 10. Running Aspectized Applications

10.1. Regular Java Applications

Applications using JBoss AOP can be run standalone, or they can be run as part of the JBoss application server. When running AOP standalone, you can either use classes that have been instrumented using the AOP precompiler, or classes that have only simply compiled using javac to be instrumented by AOP when they are first loaded.

10.1.1. 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)

You need to tell JBoss AOP where the xml configuration files of your bindings are. This can be done in one of these two ways:

- 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.
- Package your classes in a jar where `/META-INF/jboss-aop.xml` contains your bindings.

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 1.5:

```
$ 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).

10.1.2. 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. (For JDK 1.4, if you are using annotations, you will also need to use the annotation compiler to take advantage of the annotations.) 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.

How to run this is quite similar to running the precompiled examples, you need to specify one more thing. In JDK 1.4, you must also set the `java.system.class.loader` system property to `org.jboss.aop.standalone.SystemClassLoader`. Here's how run an AOP application in JDK 1.4 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> \
      -Djava.system.class.loader=org.jboss.aop.standalone.SystemClassLoader \
      com.blah.MyMainClass
```

And to run an AOP application in JDK 1.4 with loadtime instrumentation, where your application contains a jar with a META-INF/jboss-aop.xml file:

```
$ java -cp=<classpath as described above> \
      -Djava.system.class.loader=org.jboss.aop.standalone.SystemClassLoader \
      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-load-system 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).

JBoss AOP uses the JDK 5.0 `java.lang.instrument` package to enable loadtime transformations. To run loadtime transformations with JDK 5.0, you do not specify the `java.system.class.loader` system property. Instead you pass in the JVM command line switch `javaagent` with a value of `-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 1.5 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. (Since this is for JDK 1.5, there is no need to compile the annotations first).

10.1.3. Loadtime Alternative

A better way to run loadtime transformations using JDK 1.4, which is more in line with what is done when running loadtime transformations for JDK 1.5, uses some non-standard options of Java to override `java.lang.ClassLoader` with a custom classloader that hooks into the library. The custom classloader must be added 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 for J2SE 1.4, and does NOT work with J2SE 5.0 (For 1.5 use the `-javaagent` switch as mentioned above. Due to licensing issues we are not allowed to ship the precompiled version of the custom classloader. The steps to compile and use the custom classloader are shown below.

As implied above, the java executable used in this example MUST be the one that ships with JDK 1.4. If you also have, say, the JDK 1.5 runtime installed, you may run into problems. To be absolutely sure, in the console you are using set the `JAVA_HOME` environment variable to point to the root of your JDK 1.4 installation. Then on Windows invoke java as:

```
$ %JAVA_HOME%\bin\java ...
```

On Unix invoke java as:

```
$ $JAVA_HOME/bin/java ...
```

To use this you first need to generate the classloader:

```
$ 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. The usage for JDK 1.5 is:

```
$ 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!).

10.2. JBoss Application Server 4.x

JBoss AOP is integrated with JBoss 4.0 and JBoss 3.2.6RC1+ application server. You can deploy beans in two ways, precompiled for aop, or for loadtime instrumentation. The definitions of these are the same as above.

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.

To enable loadtime transformations in JBoss you need to perform the following steps:

- Unjar `deploy/jboss-aop.deployer`
- Edit `META-INF/jboss-service.xml` as described below
- Rejar `deploy/jboss-aop.deployer`

Later versions of JBoss Application Server will be distributed with this archive exploded as a directory so you do not have to unjar.

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="EnableTransformer">false</attribute>
   <!-- only relevant when EnableTransformer is true -->
   <attribute name="SuppressTransformationErrors">true</attribute>
   <!-- only relevant when Enabletransformer is true.  Optimization is optional
   only just in case there is a bug in it -->
   <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 and you will need to use the precompiler. You can turn this on by setting the EnableTransformer attribute to true. If SuppressTransformationErrors is true failed bytecode transformation will only give an error warning. This flag is needed because the JSP compiler does not run within a JBoss classloader and the AOP loader cannot resolve classes from this JSP classloader.

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 `aop.deployer` file works) or you can include it in the jar fail 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!

10.3. JBoss Application Server 3.2.x

JBoss AOP can also work with JBoss 3.2.6RC1+ and higher in the JBoss 3.2 series. Look in the Installing chapter on how to install the JAR files.

After installing, you need to modify the `jboss-3.2.6/server/xxx/conf/jboss-service.xml` file to add the mbean definitions mentioned for the 4.0 release, and copy the `base-aop.xml` file into the server/xxx/deploy/ directory if you want to use any of JBoss Aspects.

NOTE!!! JBoss 3.2.x series does not support class-load-time transformations. You MUST use the JBoss AOP precompiler if you want to run in JBoss 3.2.x.

Chapter 11. 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 exexcution 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.

11.1. 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.

11.2. 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.

Chapter 12. JBoss AOP IDE

12.1. 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!

12.2. 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.

12.3. Tutorial

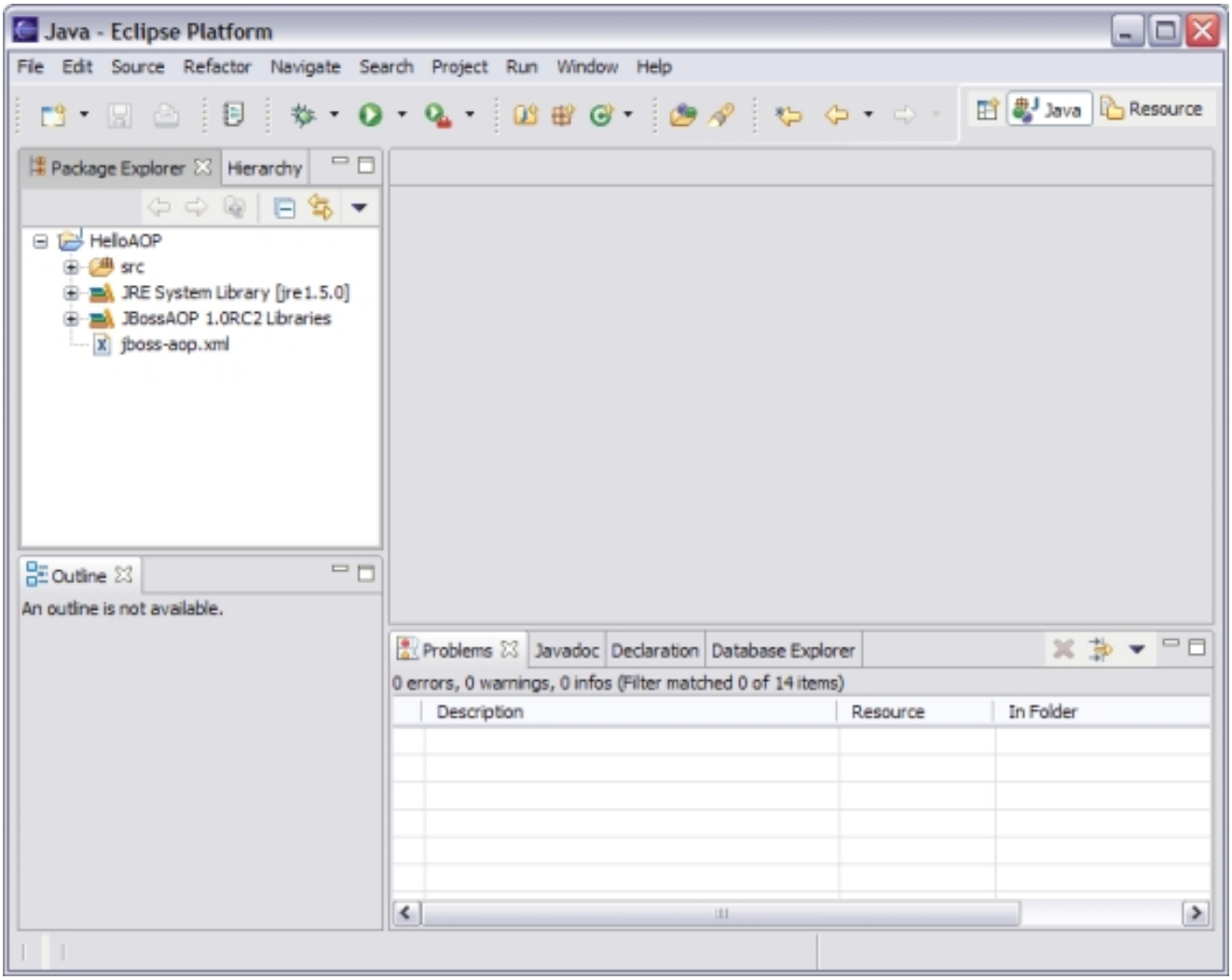
This tutorial is meant to guide you through creating a new AOP project in eclipse using the AOP extension to JBossIDE. It assumes that you have some working knowledge of AOP, and Java.. and possibly some minimal

experience dealing with eclipse as well.

12.3.1. 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:



12.3.2. 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();
        }
}
```

12.3.3. Create Interceptor

Next we want to create an interceptor to the class.

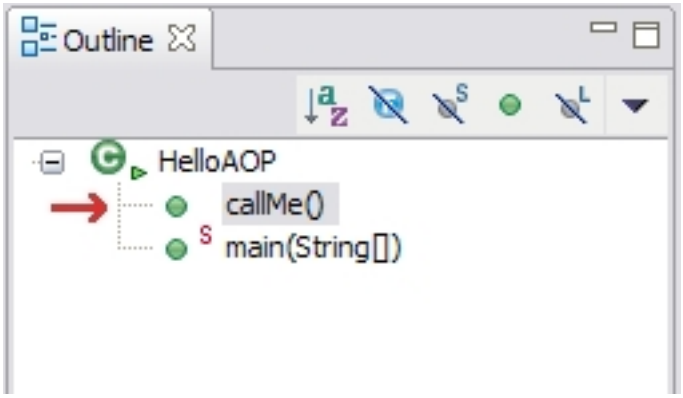
- Right click on the "src" directory in the Package Explorer and in the menu, click New > Class. In the resulting dialog:
  - Name the class `HelloAOPInterceptor`
  - Add `org.jboss.aop.advice.Interceptor` to the list of interceptors.

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();
          }
  }
```

12.3.4. 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`.

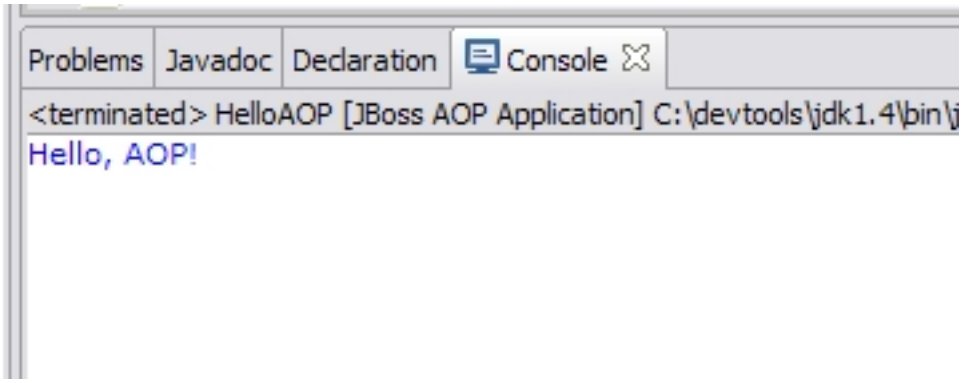
You should see in your Package Explorer that the file "jboss-aop.xml" now exists under your project root.

12.3.5. 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.



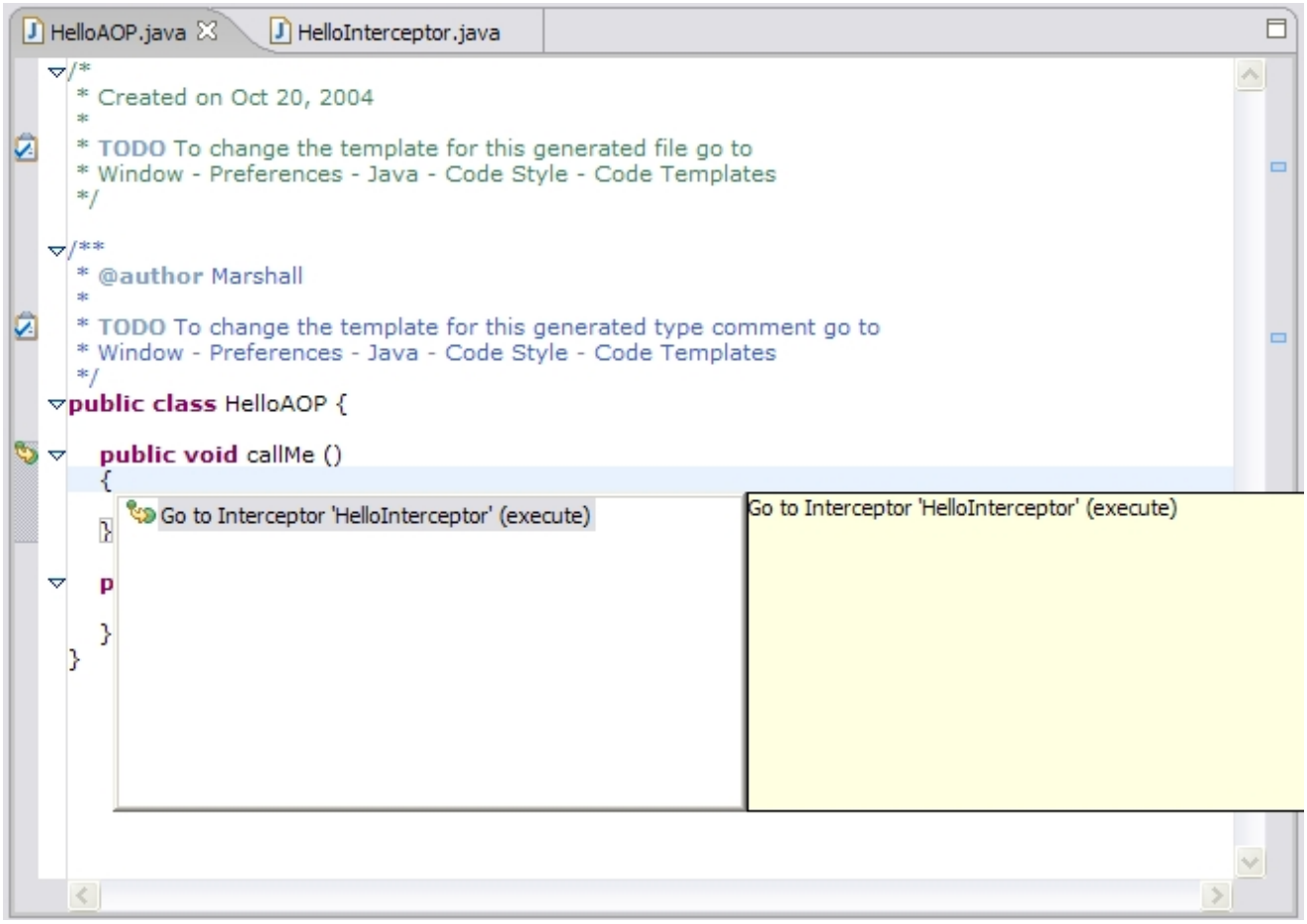
12.3.6. Navigation

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.

12.3.6.1. 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 inter-

cepted. 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!

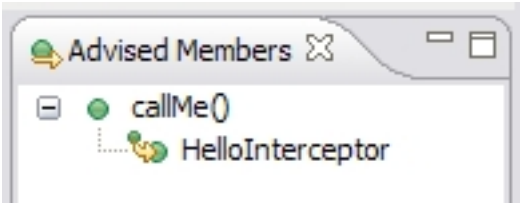


12.3.6.2. The Advised Members View

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:



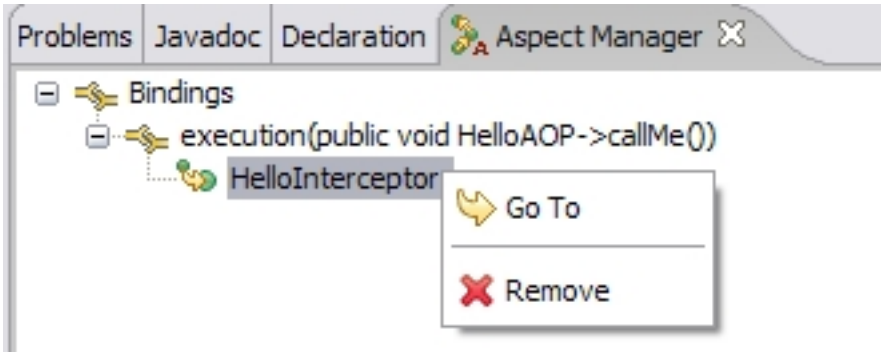
Here we see that the method "callMe()" is intercepted by the interceptor `HelloInterceptor`. Double clicking on `HelloInterceptor` will take you straight to it. This view is similar to the Outline view, except it only shows members in your class which are intercepted.

12.3.6.3. 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)