The AspectJ™ Problem Diagnosis Guide

Table of Contents

by the AspectJ Team

This guide describes how to configure the AspectJ compiler/weaver to provide information for diagnosing problems in the input programs, the compiler/weaver or its configuration.

The AspectJ compiler and weaver can provide lots of information for diagnosing problems in building AspectJ programs. For problems in the input program, there are a number of default warning and error messages, as well as many configurable "lint" messages, all of which can be emitted normally, logged using standard facilities, or intercepted programmatically. These are discussed in Messages. Since most errors relate to writing pointcuts incorrectly, there is a section on Debugging Pointcuts.

For problems with the compiler/weaver itself there are three facilities that enable the AspectJ developers to resolve bugs even when it is too hard to deliver a reproducible test case:

- Tracing can be enabled to track progress up to the time of a failure;
- AspectJ Core Files can give a relatively complete picture of the state of the world at the time of a failure; and
- Dumping classes during load-time weaving is a way to capture both input and output classes during load-time weaving.

This guide describes how to configure messages to get the right information and how to configure traces, dumps, and core files. Although the compiler/weaver operates in roughly three modes (from the command-line, embedded in an IDE, and enabled as load-time weaving), the principles are basically the same for all modes. The differences lie in how to set up diagnostics and what information is likely to be relevant.

Messages

Introduction

Messages point out potential problems in the input program; some are clearly problems (errors), but many more may depend on what the programmer intends. To keep the noise down the latter are treated as warnings which can be ignored by the programmer or information which are hidden. However, when investigating unexpected behavior it's helpful to show them. This section describes how to configure messages, presents some problem scenarios when compiling or doing load-time weaving, and summarizes some of the more relevant messages.

Configuring Messages

The compiler offers `-verbose`, `-warning`, and `-XLint` options when invoked using the command-line, Ant, or embedded in an IDE. All options are listed in the AspectJ Development Environment Guide sections for Ajc and Ant Tasks. The Load-time Weaving section describes how to use XML configuration files and system properties to pass options to the weaver. (You can also pass options to the weaver using system properties in build- time weaving.) The `-verbose` option has the effect of including messages level "info", which are normally ignored. Both `warning` and `XLint` enable you to identify specific messages to emit, but warning messages tend to be the same provided by the underlying Eclipse JDT (Java) compiler, while XLint messages are emitted by the AspectJ compiler or weaver. Obviously, during load-time weaving only weaver messages will be emitted. Similarly, if aspects are compiled but not woven, then only compiler messages will be emitted. However, the usual case for the compiler/weaver working at build time is to emit both compiler and weaver messages.

The tables below list some options, System Properties (for LTW only) and Java 5 annotations used to control AspectJ messages. The method of configuration depends on your environment so please refer to the relevant documentation for ajc, Ant or LTW.

Option	Description
`-verbose`	Show informational messages including AspectJ version and build date.
`-debug`	(Load-time weaving only). Show debugging messages such as which classes are being woven or those that are excluded. (This is not related to the compiler -g option to include debug information in the output .class files.)
`-showWeaveInfo`	Show weaving messages.
`-Xlint`	Control level of lint messages.
`messageHolderClass`/ `-XmessageHolderClass:`	In Ant tasks and LTW respectively specify the class to receive all messages. See iajc task options or Weaver Options.

System Property	Description
`aj.weaving.verbose`	Show informational messages including AspectJ version and build date (same as `-verbose` option).
`org.aspectj.weaver.showWeaveInfo`	Show weaving messages (same as `-showWeaveInfo` option).
`org.aspectj.weaving.messages`	Set this system property to enable tracing of all compiler messages. See Configuring Tracing.

Annotation	Description
`@SuppressAjWarnings`	Include this is Java 5 code to suppress AspectJ warnings associated with the next line of code.

Message scenarios

Compile-time weaving scenarios

Advice not woven

This means that the pointcut for the advice did not match, and it should be debugged as described in Debugging Pointcuts.

Load-time weaving scenarios

You can use `META-INF/aop.xml` to control which messages are produced during LTW. The following example will produce basic informational messages about the lifecyle of the weaver in addition to any warning or error messages.

```
<aspectj>
    <weaver options="-verbose">
    </weaver>
</aspectj>
```

The messages indicate which `META-INF/aop.xml` configurations file(s) are being used. Each message is also preceeded by the name of the defining class loader associated with weaver. You can use this information in a large system to distinguish between different applications each of which will typically have its own class loader.

```
[AppClassLoader@92e78c] info AspectJ Weaver Version 1.5.3 built on Thursday Oct 26,
2006 at 17:22:31 GMT
[AppClassLoader@92e78c] info register classloader
sun.misc.Launcher$AppClassLoader@92e78c
[AppClassLoader@92e78c] info using configuration /C:/temp/META-INF/aop.xml
[AppClassLoader@92e78c] info using configuration /C:/temp/META-INF/aop-ajc.xml
[AppClassLoader@92e78c] info register aspect ExceptionHandler
```

```
[AppClassLoader@92e78c] info processing reweavable type ExceptionHandler:
ExceptionHandler.aj
```

Advice not woven

It is often difficult to determine, especially when using load-time weaving (LTW), why advice has not been woven. Here is a quick guide to the messages to look for. Firstly if you use the `-verbose` option you should see the following message when your aspect is registered:

```
info register aspect MyAspect
```

Secondly if you use the `-debug` option you should see a message indicating that you class is being woven:

```
debug weaving 'HelloWorld'
```

However this does not mean that advice has actually been woven into your class; it says that the class has been passed to the weaver. To determine whether your pointcuts match you can use the `-showWeaveInfo` option which will cause a message to be issued each time a join point is woven:

```
weaveinfo Join point 'method-execution(void HelloWorld.main(java.lang.String[]))' ...
```

If advice is woven at this join point you should get the corresponding message.

Lint messages

The table below lists some useful `-Xlint` messages.

Message	Default	Description
`aspectExcludedByConfiguration`	`ignore`	If an aspect is not being woven, despite being registered, it could be that it has been excluded by either an `include` or `exclude` element in the `aspects` section of `META-INF/aop.xml`. Enable this message to determine whether an aspect has been excluded.

Message	Default	Description
`adviceDidNotMatch`	`warning`	Issued when advice did not potentially affect any join points. This means the corresponding pointcut did not match any join points in the program. This may be valid e.g., in library aspects or code picking up error conditions, but often the programmer simply made a mistake in the pointcut. The best approach is to debug the pointcut.
`invalidAbsoluteTypeName`	`warning`	Issued when an exact type in a pointcut does not match any type in the system. Note that this can interact with the rules for resolving simple types, which permit unqualified names if they are imported.
`typeNotExposedToWeaver`	`warning`	This means that a type which could be affected by an aspect is not available for weaving. This happens when a class on the classpath should be woven.
`runtimeExceptionNotSoftened`	`warning`	Before AspectJ 5, declare soft used to soften runtime exceptions (unnecessarily). Since then, it does not but does issue this warning in case the programmer did intend for the exception to be wrapped.
`unmatchedSuperTypeInCall`	`warning`	Issued when a call pointcut specifies a defining type which is not matched at the call site (where the declared type of the reference is used, not the actual runtime type). Most people should use 'target(Foo) && call(void foo())' instead.

Debugging Pointcuts

Introduction

This section describes how to write and debug pointcuts using the usual approach of iteration and decomposition. New users are often stumped when their advice does not match. That means the pointcut doesn't match; they rewrite the pointcut and it still doesn't match, with no new information. This can be frustrating if each iteration involves building, deploying, and testing a complex application. Learning to break it down, particularly into parts that can be checked at compile-time, can save a lot of time.

Debugging pointcuts

Go at it top-down and then bottom-up.

Top-down

Top-down, draft significant aspects by first writing the comments to specify responsibilities. Advice responsibility usually takes the form, *"When X, do Y"*. Pointcut responsibility for *"When X"* often takes the form, *"When [join points] [in locations] [are ...]"*. These *[]*'s often translate to named pointcuts like `libraryCalls() && within(Client) && args(Context)`, which form a semantic bridge to the plain-text meaning in a comment, e.g. `// when the client passes only context into the library`. This gets you to a point where you can debug the parts of the pointcut independently.

Bottom-up

Bottom-up (to build each part), consider each primitive pointcut designator (PCD), then the composition, and then any implicit constraints:

1. What kinds of join points should it match? (constructor-call? field-get?)? This translates to using the kinded pointcuts (`call(..)`, `get(..)`, etc.).
2. Are these restricted to being lexically within something? This translates to using `within{code}(..)`. If this is true, it should always be used, to speed up weaving.
3. What runtime constraints and context should be true and available at each join point? This translates to `this()`, `target()`, `args()`, `cflow{below}()` and `if(..)`.
4. Are there any advice or implementation limitations at issue? This involves knowing the few constraints on AspectJ imposed by Java bytecode as listed in the AspectJ Programming Guide section on Implementation Notes.

It's much faster to iterate a pointcut at compile-time using declare warning (even better, some errors are identified at parse-time in the latest versions of AJDT). Start with the parts of the pointcut that are staticly-determinable (i.e., they do not involve the runtime PCD's listed above). If compiles themselves take too long because of all the AspectJ weaving, then try to only include the debugging aspect with the prototype pointcut, and limit the scope using `within(..)`.

Common pointcut mistakes

There are some typical types of mistakes developers make when designing pointcuts. Here are a few examples:

Mistakes in primitive pointcuts

- `this(Foo) && execution(static * *(..))`: There is no `this` in a static context, so `this()` or `target()` should not be used in a static context or when targetting a static context (respectively). This happens most often when you want to say things like "all calls to `Foo` from `Bar`" and you only pick out calls to instance methods of `Foo` or you try to pick out calls from static methods of `Bar`.
- `target(Foo) && call(new(..)`: This will never match. In constructor-call join points, there is no target because the object has not been created yet.
- `call(* Foo.*(..))`: `Foo` refers to the compile-time type of the invoking reference, not the implementing class. In Java before 1.4, the compile-time type was rendered as the defining type, not the reference type; this was corrected in 1.4 (as shown when using ajc with the -1.4 flag) Most people should use `target(Foo) && call(…)`.
- `execution(* Foo.bar(..))`: An execution join point for `Foo` is always within `Foo`, so this won't pick out any overrides of `bar(..)`. Use `target(Foo) && execution(* bar(..))` for instance methods.
- `within(Foo)`: anonymous types are not known at weave-time to be within the lexically-enclosing type (a limitation of Java bytecode).

Mistakes in composition

- `call(* foo(Bar, Foo)) && args(Foo)`: This will never match. The parameters in `args(..)` are position-dependent, so `args(Foo)` only picks out join points where there is only one argument possible, of type Foo. Use the indeterminate-arguments operator `..` as needed, e.g., `args(Foo, ..)`.
- `call(* foo()) && execution(* foo())`: This will never match. Each pointcut must be true at each join point matched. For a union of different kinds of join points (here, call or execution), use `||`. E.g., to match both method-call and field-get join points, use `call(* …) || get(…)`.

Mistakes in implicit advice constraints

- `after () returning (Foo foo) : …`: after advice can bind the returned object or exception thrown. That effectively acts like `target()`, `this()`, or `args()` in restricting when the advice runs based on the runtime type of the bound object, even though it is not explicitly part of the pointcut.

Mistakes in implementation requirements

- *ajc* has to control the code for a join point in order to implement the join point. This translates to an implicit `within({code under the control of the compiler})` for all join points, with additional caveat for some join points. Take exception handlers, for example: there is no way to be sure from the bytecode where the original handler ends, so *ajc* can't implement after advice

on handler join points. (Since these are on a per-join-point basis, they should be considered for each corresponding primitive pointcut designator.) Unlike the mistakes with the primitive PCDs above, the compiler will emit an error for these caveats.

- `call(@SuperAnnotation Subclass.meth()`: Annotations are not inherited by default, so e.g., if the pointcut specifies an annotation, then subclass implementations of that method will not be matched.

Tracing

Introduction

The AspectJ developers have instrumented the compiler/weaver with many "trace" messages for their own debugging use. These remain in the production releases because tracing helps when it is hard to isolate the problem in a test case. This sections describes how to enable tracing so you can provide trace information on bug reports.

The usual approach to opening a report on Bugzilla is to describe the symptoms of the problem and attach a simple testcase. This allows the AspectJ team to try and reproduce the problem in an attempt to fix it as well as improve the test suite. Unfortunately it may not be possible to produce such a testcase either because your program is too large or is commercially sensitive. Alternatively the problem may relate to your specific environment where AspectJ is being used and will not be reproducible by the AspectJ team. In each of these situations you can produce a trace of the compiler when the problem occurs instead. This can then be attached to the bug report.

Configuring Tracing

When available (Java 5 or later) AspectJ will use the java.util.logging infrastructure configured using a `logging.properties` file. By default only error and fatal events will be logged but less severe warnings as well as fine-grained method entry and exit events can be obtained using the appropriate configuration. All regular compiler messages can also be logged through the infrastructure by setting the `org.aspectj.weaving.messages` System property.

If you are running the AspectJ compiler/weaver under JDK 1.4 or earlier, AspectJ will use a simple built-in trace infrastructure that logs to stderr. This is enabled by setting the `org.aspectj.weaving.tracing.enabled` System property. You may also override the default behaviour or provide your own trace implementation using the `org.aspectj.weaving.tracing.factory` System property.

The table below lists the System properties that can be used to configure tracing.

Property	Description
`org.aspectj.tracing.debug`	Enable simple debugging of the trace infrastructure itself. Default: `false`.
`org.aspectj.tracing.enabled`	Enable the built-in AspectJ trace infrastructure. Default: `false`.

Property	Description
`org.aspectj.tracing.factory`	Select trace infrastructure. Specify the fully qualified class name of the `org.aspectj.weaver.tools.TraceFactory` interface to use a custom infrastructure. Specify a value of `default` to force AspectJ to use it's built-in infrastructure.
`org.aspectj.tracing.messages`	Enable tracing of compiler messages. The kind of messages logged is determined by the selected trace infrastructure not the message configuration. Default: `false`.

Examples

Using `-Dorg.aspectj.tracing.factory=default` to force AspectJ to use its internal infrastructure, `-Dorg.aspectj.tracing.enabled=true` to turn it on and `-Dorg.aspectj.tracing.messages=true` to include messages running a simple HelloWorld with LTW will generate tracing to stderr. Below is an extract from that trace with method arguments removed. You will notice the millisecond time stamp, thread id and indication of entry/exit/event or message type for each line of trace.

```
15:44:18.630 main > org.aspectj.weaver.loadtime.Aj.<init>
15:44:18.660 main < org.aspectj.weaver.loadtime.Aj.<init>
15:44:18.660 main > org.aspectj.weaver.loadtime.Aj.preProcess
15:44:18.660 main - org.aspectj.weaver.loadtime.Aj.preProcess
15:44:18.730 main > org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.<init>
15:44:18.730 main < org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.<init>
15:44:18.730 main > org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.initialize
15:44:18.821 main I [AppClassLoader@92e78c] info AspectJ Weaver Version DEVELOPMENT
...
15:44:18.821 main >
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.parseDefinitions
15:44:18.821 main I [AppClassLoader@92e78c] info register classloader ...
15:44:18.821 main -
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.parseDefinitions
15:44:18.841 main -
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.parseDefinitions
15:44:18.841 main I [AppClassLoader@92e78c] info using configuration ...
15:44:18.891 main <
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.parseDefinitions
15:44:19.021 main > org.aspectj.weaver.World$TypeMap.<init>
15:44:19.021 main < org.aspectj.weaver.World$TypeMap.<init>
15:44:19.021 main > org.aspectj.weaver.CrosscuttingMembersSet.<init>
15:44:19.021 main < org.aspectj.weaver.CrosscuttingMembersSet.<init>
15:44:19.021 main > org.aspectj.weaver.Lint.<init>
15:44:19.021 main < org.aspectj.weaver.Lint.<init>
15:44:19.021 main > org.aspectj.weaver.World.<init>
```

```
15:44:19.111 main < org.aspectj.weaver.World.<init>
15:44:19.201 main > org.aspectj.weaver.bcel.BcelWeaver.<init>
15:44:19.201 main < org.aspectj.weaver.bcel.BcelWeaver.<init>
15:44:19.201 main >
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.registerDefinitions
15:44:19.211 main > org.aspectj.weaver.bcel.BcelWeaver.setReweavableMode
15:44:19.351 main < org.aspectj.weaver.bcel.BcelWeaver.setReweavableMode
15:44:19.351 main >
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.registerAspects
15:44:19.351 main I [AppClassLoader@92e78c] info register aspect Aspect
15:44:19.351 main > org.aspectj.weaver.bcel.BcelWeaver.addLibraryAspect
15:44:19.501 main - org.aspectj.weaver.bcel.BcelWorld.lookupJavaClass
15:44:19.632 main > org.aspectj.weaver.CrosscuttingMembersSet.addOrReplaceAspect
15:44:19.792 main < org.aspectj.weaver.CrosscuttingMembersSet.addOrReplaceAspect
15:44:19.792 main < org.aspectj.weaver.bcel.BcelWeaver.addLibraryAspect
15:44:19.792 main <
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.registerAspects
15:44:19.792 main <
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.registerDefinitions
15:44:19.792 main > org.aspectj.weaver.bcel.BcelWeaver.prepareForWeave
15:44:19.822 main < org.aspectj.weaver.bcel.BcelWeaver.prepareForWeave
15:44:19.822 main >
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.weaveAndDefineConcete...
15:44:19.822 main <
org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.weaveAndDefineConcete...
15:44:19.822 main < org.aspectj.weaver.loadtime.ClassLoaderWeavingAdaptor.initialize
15:44:19.822 main > org.aspectj.weaver.tools.WeavingAdaptor.weaveClass
...
```

Alternatively when running under Java 5 the `logging.properties` file below could be used to configure Java Logging. The resulting file, just containing trace for the `org.aspectj.weaver.loadtime` package, will be written to `java0.log` in your `user.home` directory.

```
handlers= java.util.logging.FileHandler

.level= INFO

java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.level = FINER

org.aspectj.weaver.loadtime.level = FINER
```

By setting the System property `-Dorg.aspectj.tracing.debug=true` you should see a message confirming which trace infrastructure is being used.

```
TraceFactory.instance=org.aspectj.weaver.tools.Jdk14TraceFactory@12dacd1
```

AspectJ Core Files

Introduction

When the compiler terminates abnormally, either because a particular kind of message was issued or an exception was thrown, an AspectJ core file will be produced. You will find it the working directory of the compiler and it will have a name that contains the date and time that the file was produced e.g. `ajcore.20060810.173655.626.txt`. The file contains details of the problem such as the exception thrown as well as information about the environment such as operating system and Java version. When submitting a bug, include this file whenever it is available.

Configuring dump files

By default AspectJ will only create an `ajcore` file when an unexpected exception is thrown by the weaver or an `abort` message is issued. However it is possible to disable this feature or enable files to be produced under different circumstances. The table below lists the System properties that can be used to configure `ajcore` files.

Property	Default	Description
`org.aspectj.weaver.Dump.exception`	`true`	Generate an `ajcore` files when an exception thrown.
`org.aspectj.weaver.Dump.condition`	`abort`	Message kind for which to generate `ajcore` e.g. `error`.
`org.aspectj.dump.directory`	`none`	The directory used for ajcore files.

AJCore File Examples

Below is an extract from an `ajcore` file. You will see details of the dump configuration as well as the exception (with stack trace) that is the source of the problem and any messages issued by the compiler. Most importantly the exact version of AspectJ is included.

```
---- AspectJ Properties ---
AspectJ Compiler DEVELOPMENT built on Tuesday Jul 25, 2006 at 13:00:09 GMT
---- Dump Properties ---
Dump file: ajcore.20060810.173655.626.txt
Dump reason: java.lang.NoClassDefFoundError
Dump on exception: true
Dump at exit condition: abort
---- Exception Information ---
java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory
    at
org.aspectj.weaver.tools.CommonsTraceFactory.<init>(CommonsTraceFactory.java:17)
    at java.lang.Class.newInstance0(Native Method)
    at java.lang.Class.newInstance(Class.java:232)
    at org.aspectj.weaver.tools.TraceFactory.<clinit>(TraceFactory.java:35)
```

```
    at org.aspectj.weaver.World.<clinit>(World.java:114)
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.initBcelWorld(AjBuildManager.jav
a:679)
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.doBuild(AjBuildManager.java:224)
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.batchBuild(AjBuildManager.java:1
64)
    at org.aspectj.ajdt.ajc.AjdtCommand.doCommand(AjdtCommand.java:112)
    at org.aspectj.ajdt.ajc.AjdtCommand.runCommand(AjdtCommand.java:60)
    at org.aspectj.tools.ajc.Main.run(Main.java:367)
    at org.aspectj.tools.ajc.Main.runMain(Main.java:246)
    at org.aspectj.tools.ajc.Main.main(Main.java:86)
---- System Properties ---
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=C:\jdk1.3.1_16\jre\bin
java.vm.version=1.3.1_16-b06
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=https://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
java.vm.specification.name=Java Virtual Machine Specification
user.dir=C:\workspaces\org.aspectj\org.aspectj.ant.tests
java.runtime.version=1.3.1_16-b06
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
os.arch=x86
java.io.tmpdir=C:\DOCUME~1\IBM_user\LOCALS~1\Temp\
line.separator=

java.vm.specification.vendor=Sun Microsystems Inc.
java.awt.fonts=
os.name=Windows XP
java.library.path=C:\jdk1.3.1_16\jre\bin;...
java.specification.name=Java Platform API Specification
java.class.version=47.0
os.version=5.1
user.home=C:\Documents and Settings\IBM_user
user.timezone=Europe/London
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=Cp1252
java.specification.version=1.3
java.class.path=C:\workspaces\org.aspectj\aj-build\dist\tools\lib\aspectjtools.jar
user.name=IBM_user
java.vm.specification.version=1.0
java.home=C:\jdk1.3.1_16\jre
user.language=en
java.specification.vendor=Sun Microsystems Inc.
awt.toolkit=sun.awt.windows.WToolkit
java.vm.info=mixed mode
```

```
java.version=1.3.1_16
java.ext.dirs=C:\jdk1.3.1_16\jre\lib\ext
sun.boot.class.path=C:\jdk1.3.1_16\jre\lib\rt.jar;...
java.vendor=Sun Microsystems Inc.
file.separator=\
java.vendor.url.bug=https://java.sun.com/cgi-bin/bugreport.cgi
sun.io.unicode.encoding=UnicodeLittle
sun.cpu.endian=little
user.region=GB
sun.cpu.isalist=pentium i486 i386
---- Command Line ---
-d
C:\workspaces\org.aspectj\org.aspectj.ant.tests\IncrementalAjcTaskTest-temp
-g:none
-deprecation
-noExit
-warn:-unusedImport
-nowarn
-XterminateAfterCompilation
-preserveAllLocals
-proceedOnError
-referenceInfo
-source
1.3
-target
1.1
-time
-verbose
-classpath
C:\workspaces\org.aspectj\org.aspectj.ant.tests\..\lib\test\aspectjrt.jar
-argfile
C:\workspaces\org.aspectj\taskdefs\testdata\default.lst
-messageHolder
org.aspectj.bridge.MessageHandler
---- Full Classpath ---
Empty
---- Compiler Messages ---
abort ABORT -- (NoClassDefFoundError) org/apache/commons/logging/LogFactory
org/apache/commons/logging/LogFactory
java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory
    at
org.aspectj.weaver.tools.CommonsTraceFactory.<init>(CommonsTraceFactory.java:17)
    at java.lang.Class.newInstance0(Native Method)
    at java.lang.Class.newInstance(Class.java:232)
    at org.aspectj.weaver.tools.TraceFactory.<clinit>(TraceFactory.java:35)
    at org.aspectj.weaver.World.<clinit>(World.java:114)
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.initBcelWorld(AjBuildManager.jav
a:679)
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.doBuild(AjBuildManager.java:224)
```

```
    at
org.aspectj.ajdt.internal.core.builder.AjBuildManager.batchBuild(AjBuildManager.java:1
64)
    at org.aspectj.ajdt.ajc.AjdtCommand.doCommand(AjdtCommand.java:112)
    at org.aspectj.ajdt.ajc.AjdtCommand.runCommand(AjdtCommand.java:60)
    at org.aspectj.tools.ajc.Main.run(Main.java:367)
    at org.aspectj.tools.ajc.Main.runMain(Main.java:246)
    at org.aspectj.tools.ajc.Main.main(Main.java:86)
```

Dumping classes during load-time weaving

Introduction

Very rarely problems may be encountered with classes that have been load-time woven. Symptoms will include incorrect program function or a Java exception such as `java.lang.VerifyError`. In these situations it's most helpful to include the offending class in the bug report. When using load-time weaving the woven classes are in memory only so to save them to disk configure `META-INF/aop.xml` to dump the classes (by default to an `_ajdump` subdirectory of the current working directory). Also if the input class file is not available (e.g. it is a generated proxy or has already been instrumented by another agent) you can configure the weaver to dump the input classes as well.

Configuring bytecode dumping in load-time weaving

For details of how to configure byte-code dumping, see the AspectJ Development Environment Guide section on Configuring Load-time Weaving. Following is a simple example.

LTW Dump Examples

The following `META-INF/aop.xml` will weave classes in the `com.foo` package (and subpackages) but not CGLIB generated classes in the `com.foo.bar` package (and subpackages). It will also ensure all woven byte-code is dumped both before and after weaving.

```
<aspectj>
    <aspects>
        <aspect name="ataspectj.EmptyAspect"/>
    </aspects>
    <weaver options="-verbose -debug">
        <dump within="com.foo.bar..*" beforeandafter="true"/>
        <include within="com.foo..*"/>
        <exclude within="com.foo.bar..*CGLIB*"/>
    </weaver>
</aspectj>
```

You should see messages similar to this:

```
[WeavingURLClassLoader] info AspectJ Weaver Version 1.5.3 built on Thursday Oct 26,
2006 at 17:22:31 GMT
[WeavingURLClassLoader] info register classloader
org.aspectj.weaver.loadtime.WeavingURLClassLoader
[WeavingURLClassLoader] info using configuration /C:/tempMETA-INF/aop.xml
[WeavingURLClassLoader] info register aspect ataspectj.EmptyAspect
[WeavingURLClassLoader] debug not weaving 'com.foo.bar.Test$$EnhancerByCGLIB$$12345'
[WeavingURLClassLoader] debug weaving 'com.foo.bar.Test'
```

On disk you would find the following files:

```
_ajdump/_before/com/foo/bar/Test.class
_ajdump/com/foo/bar/Test.class
```