The AspectJ™ Development Environment Guide

Table of Contents

by the AspectJ Team

This guide describes how to build and deploy AspectJ programs using the AspectJ tools and facilities. See also the AspectJ Programming Guide, the documentation available with the AspectJ support available for various integrated development environments (e.g. Eclipse AJDT), and the most-recent documentation available from the AspectJ project page.

Introduction to the AspectJ tools

The Eclipse AspectJ implementation

The AspectJ Programming Guide describes the AspectJ language. This guide describes the AspectJ tools produced by the AspectJ team on https://eclipse.org/aspectj. The AspectJ tools include - ajc, the compiler/weaver; ajdoc, a documentation tool; Ant support for ajc; and load-time weaving support. These tools are delivered in the library folder of the AspectJ tools installation, mainly in `aspectjtools.jar` (tools) and `aspectjrt.jar` (runtime). This guide does not describe the Eclipse AspectJ development tools (AJDT). That is produced by another team (sharing some members) on https://eclipse.org/ajdt. AJDT is delivered as an Eclipse plugin, incorporating the classes in the AspectJ tools libraries along with the Eclipse plugin interface classes.

Since AspectJ 1.1, the tools have implemented the AspectJ language using bytecode weaving, which combines aspects and classes to produce .class files that run in a Java VM. There are other ways to implement the language (e.g., compiler preprocessor, VM support); the AspectJ team has always tried to distinguish the language and the implementation so other groups could build alternative implementations of AspectJ. To that end, The AspectJ Programming Guide, Implementation Notes describes how the Java bytecode form affects language semantics. VM- or source-based implementations may be free of these limits or impose limits of their own, but most should be fairly close to what's possible in Java bytecode.

Please be careful not to confuse any description of weaving or of this implementation of the AspectJ language with the AspectJ language semantics. If you do, you might find yourself writing code that doesn't work as expected when you compile or run it on other systems. More importantly, if you think about aspects in terms of weaving or of inserting or merging code, then you can lose many of the design benefits of thinking about an aspect as a single crosscutting module. When the text below introduces an implementation detail, it will warn if users make mistakes by applying it in lieu of the language semantics.

Bytecode weaving, incremental compilation, and memory usage

Bytecode weaving takes classes and aspects in .class form and weaves them together to produce binary-compatible .class files that run in any Java VM and implement the AspectJ semantics. This process supports not only the compiler but also IDE's. The compiler, given an aspect in source form, produces a binary aspect and runs the weaver. IDE's can get information about crosscutting in the program by subscribing to information produced by weaver as a side-effect of weaving.

Incremental compilation involves recompiling only what is necessary to bring the binary form of a program up-to-date with the source form in the shortest time possible. Incremental weaving supports this by weaving on a per-class basis. (Some implementations of AOP (including AspectJ 1.0) make use of whole-program analysis that can't be done in incremental mode.) Weaving per-class means that if the source for a pure Java class is updated, only that class needs to be produced. However, if some crosscutting specification may have been updated, then all code potentially affected by it may need to be woven. The AspectJ tools are getting better at minimizing this effect,

but it is to some degree unavoidable due to the crosscutting semantics.

Memory usage can seem higher with AspectJ tools. Some aspects are written to potentially affect many classes, so each class must be checked during the process of weaving. Programmers can minimize this by writing the crosscutting specifications as narrowly as possible while maintaining correctness. (While it may seem like more memory, the proper comparison would with with a Java program that had the same crosscutting, with changes made to each code segment. That would likely require more memory and more time to recompile than the corresponding AspectJ program.)

Classpath, inpath, and aspectpath

AspectJ introduces two new paths for the binary input to the weaver which you'll find referenced in `ajc`, the AspectJ compiler/weaver, AspectJ Ant Tasks, and Load-Time Weaving.

As in Java, the `classpath` is where the AspectJ tools resolve types specified in the program. When running an AspectJ program, the classpath should contain the classes and aspects along with the AspectJ runtime library, `aspectjrt.jar`.

In AspectJ tools, the `aspectpath` is where to find binary aspects. Like the classpath, it can include archives (.jar and .zip files) and directories containing .class files in a package layout (since binary aspects are in .class files). These aspects affect other classes in exactly the same way as source-level aspects, but are themselves not affected. When deploying programs, the original aspects must be included on the runtime classpath.

In AspectJ tools, the `inpath` is where to find binary input - aspects and classes that weave and may be woven. Like the classpath, it can include archives and class directories. Like the aspectpath, it can include aspects that affect other classes and aspects. However, unlike the aspectpath, an aspect on the inpath may itself be affected by aspects, as if the source were all compiled together. When deploying aspects that were put on the inpath, only the woven output should be on the runtime classpath.

Although types in the inpath and the aspectpath need to be resolved by the AspectJ tools, you usually do not need to place them on the classpath because this is done automatically by the compiler/weaver. But when using the `WeavingURLClassLoader`, your code must explicitly add the aspects to the classpath so they can be resolved (as you'll see in the sample code and the `aj.bat` script).

The most common mistake is failing to add `aspectjrt.jar` to the classpath. Also, when weaving with binary aspects, users forget to deploy the aspect itself along with any classes it requires. A more subtle mistake is putting a binary aspect (BA) on the inpath instead of the aspectpath. In this case the aspect BA might be affected by an aspect, even itself; this can cause the program to fail, e.g., when an aspect uses exclusion to avoid infinite recursion but fails to exclude advice in aspect BA.

The latter is one of many ways that mistakes in the build process can affect aspects that are written poorly. Aspects should never rely on the boundaries of the build specification to narrow the scope of their crosscutting, since the build can be changed without notice to the aspect developer. Careful users might even avoid relying on the implementation scope, to ensure their AspectJ code will run on other implementations.

`ajc`, the AspectJ compiler/weaver

Name

`ajc` - compiler and bytecode weaver for the AspectJ and Java languages

Synopsis

```
ajc [option...] [file... | @file... | -argfile file...]
```

Description

The `ajc` command compiles and weaves AspectJ and Java source and .class files, producing .class files compliant with any Java VM (1.1 or later). It combines compilation and bytecode weaving and supports incremental builds; you can also weave bytecode at run-time using Load-Time Weaving.

The arguments after the options specify the source file(s) to compile. To specify source classes, use `-inpath` (below). Files may be listed directly on the command line or in a file. The `-argfile file` and `@file` forms are equivalent, and are interpreted as meaning all the arguments listed in the specified file.

`Note:` You must explicitly pass `ajc` all necessary sources. Be sure to include the source not only for the aspects or pointcuts but also for any affected types. Specifying all sources is necessary because, unlike javac, ajc does not search the sourcepath for classes. (For a discussion of what affected types might be required, see The AspectJ Programming Guide, Implementation Appendix.)

To specify sources, you can list source files as arguments or use the options `-sourceroots` or `-inpath`. If there are multiple sources for any type, the result is undefined since ajc has no way to determine which source is correct. (This happens most often when users include the destination directory on the inpath and rebuild.)

Options

`-injars <JarList>`

deprecated: since 1.2, use -inpath, which also takes directories.

`-inpath <Path>`

Accept as source bytecode any .class files in the .jar files or directories on Path. The output will include these classes, possibly as woven with any applicable aspects. Path is a single argument containing a list of paths to zip files or directories, delimited by the platform-specific path delimiter.

`-aspectpath <Path>`

Weave binary aspects from jar files and directories on path into all sources. The aspects should have been output by the same version of the compiler. When running the output classes, the run

classpath should contain all aspectpath entries. Path, like classpath, is a single argument containing a list of paths to jar files, delimited by the platform- specific classpath delimiter.

`-argfile <File>`

The file contains a line-delimited list of arguments. Each line in the file should contain one option, filename, or argument string (e.g., a classpath or inpath). Arguments read from the file are inserted into the argument list for the command. Relative paths in the file are calculated from the directory containing the file (not the current working directory). Comments, as in Java, start with `//` and extend to the end of the line. Options specified in argument files may override rather than extending existing option values, so avoid specifying options like ←classpath> in argument files unlike the argument file is the only build specification. The form <@file> is the same as specifying ←argfile file>.

`-outjar <output.jar>`

Put output classes in zip file output.jar.

`-outxml`

Generate aop.xml file for load-time weaving with default name.

`-outxmlfile <custom/aop.xml>`

Generate aop.xml file for load-time weaving with custom name.

`-incremental`

Run the compiler continuously. After the initial compilation, the compiler will wait to recompile until it reads a newline from the standard input, and will quit when it reads a 'q'. It will only recompile necessary components, so a recompile should be much faster than doing a second compile. This requires -sourceroots.

`-sourceroots <DirPaths>`

Find and build all .java or .aj source files under any directory listed in DirPaths. DirPaths, like classpath, is a single argument containing a list of paths to directories, delimited by the platform-specific classpath delimiter. Required by -incremental.

`-crossrefs`

Generate a build .ajsym file into the output directory. Used for viewing crosscutting references by tools like the AspectJ Browser.

`-emacssym`

Generate .ajesym symbol files for emacs support (deprecated).

`-Xlint`

Same as -Xlint:warning (enabled by default)

`-Xlint:{level}`

Set default level for messages about potential programming mistakes in crosscutting code. {level} may be ignore, warning, or error. This overrides entries in org/aspectj/weaver/XlintDefault.properties from aspectjtools.jar, but does not override levels set using the -Xlintfile option.

`-Xlintfile <PropertyFile>`

Specify properties file to set levels for specific crosscutting messages. PropertyFile is a path to a Java .properties file that takes the same property names and values as org/aspectj/weaver/XlintDefault.properties from aspectjtools.jar, which it also overrides.

`-help`

Emit information on compiler options and usage

`-version`

Emit the version of the AspectJ compiler

`-classpath <Path>`

Specify where to find user class files. Path is a single argument containing a list of paths to zip files or directories, delimited by the platform-specific path delimiter.

`-bootclasspath <Path>`

Override location of VM's bootclasspath for purposes of evaluating types when compiling. Path is a single argument containing a list of paths to zip files or directories, delimited by the platform-specific path delimiter.

`-extdirs <Path>`

Override location of VM's extension directories for purposes of evaluating types when compiling. Path is a single argument containing a list of paths to directories, delimited by the platform-specific path delimiter.

`-d <Directory>`

Specify where to place generated .class files. If not specified, <Directory> defaults to the current working dir.

`-target <[1.1 to 1.5]>`

Specify classfile target setting (1.1 to 1.5, default is 1.2)

`-1.3`

Set compliance level to 1.3 This implies -source 1.3 and -target 1.1.

`-1.4`

Set compliance level to 1.4 (default) This implies -source 1.4 and -target 1.2.

`-1.5`

Set compliance level to 1.5. This implies -source 1.5 and -target 1.5.

`-source <[1.3|1.4|1.5]>`

Toggle assertions (1.3, 1.4, or 1.5 - default is 1.4). When using -source 1.3, an assert() statement valid under Java 1.4 will result in a compiler error. When using -source 1.4, treat `assert` as a keyword and implement assertions according to the 1.4 language spec. When using -source 1.5, Java 5 language features are permitted.

`-nowarn`

Emit no warnings (equivalent to '-warn:none') This does not suppress messages generated by `declare warning` or `Xlint`.

`-warn: <items>`

Emit warnings for any instances of the comma-delimited list of questionable code (eg '-warn:unusedLocals,deprecation'):

```
constructorName         method with constructor name
packageDefaultMethod    attempt to override package-default method
deprecation             usage of deprecated type or member
maskedCatchBlocks       hidden catch block
unusedLocals            local variable never read
unusedArguments         method argument never read
unusedImports           import statement not used by code in file
none                    suppress all compiler warnings
```

`-warn:none` does not suppress messages generated by `declare warning` or `Xlint`.

`-deprecation`

Same as -warn:deprecation

`-noImportError`

Emit no errors for unresolved imports

`-proceedOnError`

Keep compiling after error, dumping class files with problem methods

`-g<:[lines,vars,source]>`

debug attributes level, that may take three forms:

```
-g          all debug info ('-g:lines,vars,source')
-g:none     no debug info
-g:{items} debug info for any/all of [lines, vars, source], e.g.,
            -g:lines,source
```

`-preserveAllLocals`

Preserve all local variables during code generation (to facilitate debugging).

`-referenceInfo`

Compute reference information.

`-encoding <format>`

Specify default source encoding format. Specify custom encoding on a per file basis by suffixing each input source file/folder name with '[encoding]'.

`-verbose`

Emit messages about accessed/processed compilation units

`-showWeaveInfo`

Emit messages about weaving

`-log <file>`

Specify a log file for compiler messages.

`-progress`

Show progress (requires -log mode).

`-time`

Display speed information.

`-noExit`

Do not call System.exit(n) at end of compilation (n=0 if no error)

`-repeat <N>`

Repeat compilation process N times (typically to do performance analysis).

`-XterminateAfterCompilation`

Causes compiler to terminate before weaving

`-XaddSerialVersionUID`

Causes the compiler to calculate and add the SerialVersionUID field to any type implementing Serializable that is affected by an aspect. The field is calculated based on the class before weaving has taken place.

`-Xreweavable[:compress]`

(Experimental - deprecated as now default) Runs weaver in reweavable mode which causes it to create woven classes that can be rewoven, subject to the restriction that on attempting a reweave all the types that advised the woven type must be accessible.

`-XnoInline`

(Experimental) do not inline around advice

`-XincrementalFile <file>`

(Experimental) This works like incremental mode, but using a file rather than standard input to control the compiler. It will recompile each time file is changed and and halt when file is deleted.

`-XserializableAspects`

(Experimental) Normally it is an error to declare aspects Serializable. This option removes that restriction.

`-XnotReweavable`

(Experimental) Create class files that can't be subsequently rewoven by AspectJ.

`-Xajruntimelevel:1.2, ajruntimelevel:1.5`

(Experimental) Allows code to be generated that targets a 1.2 or a 1.5 level AspectJ runtime (default 1.5)

File names

ajc accepts source files with either the `.java` extension or the `.aj` extension. We normally use `.java` for all of our files in an AspectJ system — files that contain aspects as well as files that contain classes. However, if you have a need to mechanically distinguish files that use AspectJ's additional functionality from those that are pure Java we recommend using the `.aj` extension for those files.

We'd like to discourage other means of mechanical distinction such as naming conventions or sub-packages in favor of the `.aj` extension.

- Filename conventions are hard to enforce and lead to awkward names for your aspects. Instead of `TracingAspect.java` we recommend using `Tracing.aj` (or just `Tracing.java`) instead.
- Sub-packages move aspects out of their natural place in a system and can create an artificial need for privileged aspects. Instead of adding a sub-package like `aspects` we recommend using the `.aj` extension and including these files in your existing packages instead.

Compatibility

AspectJ is a compatible extension to the Java programming language. The AspectJ compiler adheres to the The Java Language Specification, Second Edition and to the The Java Virtual Machine Specification, Second Edition and runs on any Java 2 compatible platform. The code it generates runs on any Java 1.1 or later compatible platform. For more information on compatibility with Java and with previous releases of AspectJ, see Version Compatibility.

Examples

Compile two files:

```
ajc HelloWorld.java Trace.java
```

To avoid specifying file names on the command line, list source files in a line-delimited text argfile. Source file paths may be absolute or relative to the argfile, and may include other argfiles by @-reference. The following file `sources.lst` contains absolute and relative files and @-references:

```
Gui.java
/home/user/src/Library.java
data/Repository.java
data/Access.java
@../../common/common.lst
@/home/user/src/lib.lst
view/body/ArrayView.java
```

Compile the files using either the -argfile or @ form:

```
ajc -argfile sources.lst
ajc @sources.lst
```

Argfiles are also supported by jikes and javac, so you can use the files in hybrid builds. However, the support varies:

- Only ajc accepts command-line options
- Jikes and Javac do not accept internal @argfile references.
- Jikes and Javac only accept the @file form on the command line.

Bytecode weaving using -inpath: AspectJ 1.2 supports weaving .class files in input zip/jar files and directories. Using input jars is like compiling the corresponding source files, and all binaries are emitted to output. Although Java-compliant compilers may differ in their output, ajc should take as input any class files produced by javac, jikes, eclipse, and, of course, ajc. Aspects included in -inpath will be woven into like other .class files, and they will affect other types as usual.

Aspect libraries using -aspectpath: AspectJ 1.1 supports weaving from read-only libraries containing aspects. Like input jars, they affect all input; unlike input jars, they themselves are not affected or emitted as output. Sources compiled with aspect libraries must be run with the same aspect libraries on their classpath.

The following example builds the tracing example in a command-line environment; it creates a read-only aspect library, compiles some classes for use as input bytecode, and compiles the classes and other sources with the aspect library.

The tracing example is in the AspectJ distribution ({aspectj}/doc/examples/tracing). This uses the following files:

```
aspectj1.1/
  bin/
    ajc
  lib/
    aspectjrt.jar
  examples/
    tracing/
      Circle.java
      ExampleMain.java
      lib/
        AbstractTrace.java
        TraceMyClasses.java
      notrace.lst
      Square.java
      tracelib.lst
      tracev3.lst
      TwoDShape.java
      version3/
        Trace.java
```

```
        TraceMyClasses.java
```

Below, the path separator is taken as ";", but file separators are "/". All commands are on one line. Adjust paths and commands to your environment as needed.

Setup the path, classpath, and current directory:

```
cd examples
export ajrt=../lib/aspectjrt.jar
export CLASSPATH="$ajrt"
export PATH="../bin:$PATH"
```

Build a read-only tracing library:

```
ajc -argfile tracing/tracelib.lst -outjar tracelib.jar
```

Build the application with tracing in one step:

```
ajc -aspectpath tracelib.jar -argfile tracing/notrace.lst -outjar tracedapp.jar
```

Run the application with tracing:

```
java -classpath "$ajrt;tracedapp.jar;tracelib.jar" tracing.ExampleMain
```

Build the application with tracing from binaries in two steps:

- (a) Build the application classes (using javac for demonstration's sake):

  ```
  mkdir classes
  javac -d classes tracing/*.java
  jar cfM app.jar -C classes .
  ```

- (b) Build the application with tracing:

  ```
  ajc -inpath app.jar -aspectpath tracelib.jar -outjar tracedapp.jar
  ```

Run the application with tracing (same as above):

```
java -classpath "$ajrt;tracedapp.jar;tracelib.jar" tracing.ExampleMain
```

Run the application without tracing:

```
java -classpath "app.jar" tracing.ExampleMain
```

The AspectJ compiler API

The AspectJ compiler is implemented completely in Java and can be called as a Java class. The only interface that should be considered public are the public methods in `org.aspectj.tools.ajc.Main`. E.g., `main(String[] args)` takes the the standard `ajc` command line arguments. This means that an alternative way to run the compiler is

```
java org.aspectj.tools.ajc.Main [option...] [file...]
```

To access compiler messages programmatically, use the methods `setHolder(IMessageHolder holder)` and/or `run(String[] args, IMessageHolder holder)`. `ajc` reports each message to the holder using `IMessageHolder.handleMessage(..)`. If you just want to collect the messages, use `MessageHandler` as your `IMessageHolder`. For example, compile and run the following with `aspectjtools.jar` on the classpath:

```
import org.aspectj.bridge.*;
import org.aspectj.tools.ajc.Main;
import java.util.Arrays;

public class WrapAjc {
  public static void main(String[] args) {
    Main compiler = new Main();
    MessageHandler m = new MessageHandler();
    compiler.run(args, m);
    IMessage[] ms = m.getMessages(null, true);
    System.out.println("messages: " + Arrays.asList(ms));
  }
}
```

Stack Traces and the SourceFile attribute

Unlike traditional java compilers, the AspectJ compiler may in certain cases generate classfiles from multiple source files. Unfortunately, the original Java class file format does not support multiple SourceFile attributes. In order to make sure all source file information is available, the AspectJ compiler may in some cases encode multiple filenames in the SourceFile attribute. When the Java VM generates stack traces, it uses this attribute to specify the source file.

(The AspectJ 1.0 compiler also supports the .class file extensions of JSR-45. These permit compliant debuggers (such as jdb in Java 1.4.1) to identify the right file and line even given many source files for a single class. JSR-45 support is planned for ajc in AspectJ 1.1, but is not in the initial release. To get fully debuggable .class files, use the -XnoInline option.)

Probably the only time you may see this format is when you view stack traces, where you may encounter traces of the format

```
java.lang.NullPointerException
  at Main.new$constructor_call37(Main.java;SynchAspect.java[1k]:1030)
```

where instead of the usual

```
File:LineNumber
```

format, you see

```
File0;File1[Number1];File2[Number2] ... :LineNumber
```

In this case, LineNumber is the usual offset in lines plus the "start line" of the actual source file. That means you use LineNumber both to identify the source file and to find the line at issue. The number in [brackets] after each file tells you the virtual "start line" for that file (the first file has a start of 0).

In our example from the null pointer exception trace, the virtual start line is 1030. Since the file SynchAspect.java "starts" at line 1000 [1k], the LineNumber points to line 30 of SynchAspect.java.

So, when faced with such stack traces, the way to find the actual source location is to look through the list of "start line" numbers to find the one just under the shown line number. That is the file where the source location can actually be found. Then, subtract that "start line" from the shown line number to find the actual line number within that file.

In a class file that comes from only a single source file, the AspectJ compiler generates SourceFile attributes consistent with traditional Java compilers.

ajdoc, the AspectJ API documentation generator

Name

ajdoc - generate HTML API documentation, including crosscutting structure

Synopsis

```
ajdoc [ -bootclasspath classpathlist ] [ -classpath classpathlist ] [-d path] [-help]
[-package] [-protected] [-private] [-public] [-overview overviewFile] [ -sourcepath
sourcepathlist ] [-verbose] [-version] [sourcefiles... | packages... | @file... |
-argfile file...] [ ajc options ]
```

Description

ajdoc renders HTML documentation for AspectJ constructs as well as the Java constructs that javadoc renders. In addition ajdoc displays the crosscutting nature in the form of links. That means, for example, that you can see everything affecting a method when reading the documentation for the method.

To run ajdoc, use one of the scripts in the AspectJ bin directory. The ajdoc implementation builds on Sun's javadoc command line tool, and you use it in the same way with many of the same options (javadoc options are not documented here; for more information on javadoc usage, see the Javadoc homepage.)

As with ajc (but unlike javadoc), you pass ajdoc all your aspect source files and any files containing types affected by the aspects; it's often easiest to just pass all the .java and .aj files in your system. Unlike ajc, ajdoc will try to find package sources using the specified sourcepath if you list packages on the command line.

To provide an argfile listing the source files, you can use use the same argfile (@filename) conventions as with ajc. For example, the following documents all the source files listed in argfile.lst, sending the output to the docDir output directory.

```
ajdoc -d docDir @argfile.lst
```

See ajc, the AspectJ compiler/weaver for details on the text file format.

ajdoc honours ajc options. See the ajc documentation for details on these options.

ajdoc currently requires the tools.jar from J2SE 1.3 to be on the classpath. Normally the scripts set this up, assuming that your JAVA_HOME variable points to an appropriate installation of Java. You may need to provide this jar when using a different version of Java or a JRE.

Examples

- Change into the `examples` directory.
- Type `mkdir doc` to create the destination directory for the documentation.
- Type `ajdoc -private -d doc spacewar coordination` to generate the documentation. (Use `-private` to get all members, since many of the interesting ones in spacewar are not public.)
- Type `ajdoc -private -d doc @spacewar/demo.lst` to use the argfile associated with Spacewar.
- To view the documentation, open the file `index.html` in the `doc` directory using a web browser.

`aj`, the AspectJ load-time weaving launcher

Name

`aj` - command-line launcher for basic load-time weaving

Synopsis

aj [*Options*] [*arg*...]

Description

The `aj` command runs Java programs in Java 1.4 or later by setting up `WeavingURLClassLoader` as the system class loader, to do load-time bytecode weaving.

The arguments are the same as those used to launch the Java program. Users should define the environment variables `CLASSPATH` and `ASPECTPATH`.

For more information and alternatives for load-time weaving, see Load-Time Weaving.

Examples

Use ajc to build a library, then weave at load time

```
REM compile library
${ASPECTJ_HOME}\bin\ajc.bat -outjar lib\aspects.jar @aspects.lst

REM run, weaving into application at load-time set
ASPECTPATH=lib\aspects.jar set CLASSPATH=app\app.jar
${ASPECTJ_HOME}\bin\aj.bat com.company.app.Main "Hello, World!"
```

AspectJ Ant Tasks

Introduction

AspectJ contains a compiler, `ajc`, that can be run from Ant. Included in the `aspectjtools.jar` are Ant binaries to support three ways of running the compiler:

1. AjcTask (iajc), a task to run the AspectJ post-1.1 compiler, which supports all the eclipse and ajc options, including incremental mode.
2. Ajc11CompilerAdapter (javac), an adapter class to run the new compiler using Javac tasks by setting the build.compiler property
3. Ajc10 (ajc), a task to run build scripts compatible with the AspectJ 1.0 tasks

This describes how to install and use the tasks and the adapter. For an example Ant script, see examples/build.xml.

Installing Ant Tasks

Install Jakarta Ant 1.5.1: Please see the official Jakarta Ant website for more information and the 1.5.1 distribution. This release is source-compatible with Ant 1.3 and Ant 1.4, but the task sources must be compiled with those versions of the Ant libraries to be used under those versions of Ant. Sources are available under the Eclipse Public License v 2.0 at https://eclipse.org/aspectj.

In Ant 1.5, third-party tasks can be declared using a taskdef entry in the build script, to identify the name and classes. When declaring a task, include the `aspectjtools.jar` either in the taskdef classpath or in `${ANT_HOME}/lib` where it will be added to the system class path by the ant script. You may specify the task script names directly, or use the "resource" attribute to specify the default names:

```
<taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties"/>
```

The current resource file retains the name "ajc" for the Ajc10 task, and uses "iajc" for the AspectJ post-1.1 task.

In Ant 1.6, third-party tasks are declared in their own namespace using `antlib.xml`. For example, the following script would build and run the spacewar example, if you put the script in the examples directory and `aspectjtools.jar` in the `${ANT_HOME}/lib` directory.

```
<project name="aspectj-ant1.6" default="spacewar"
    xmlns:aspectj="antlib:org.aspectj" basedir=".">
    <target name="spacewar">
        <aspectj:iajc
            argfiles="spacewar/debug.lst"
            outjar="spacewar.jar"
            classpath="../../lib/aspectjrt.jar"
       />
```

```
        <java classname="spacewar.Game"
         classpath="spacewar.jar:../../lib/aspectjrt.jar"/>
    </target>
</project>
```

For more information on using Ant, please refer to Jakarta's documentation on integrating user-defined Ant tasks into builds.

AjcTask (iajc)

This task uses the AspectJ post-1.1 compiler ajc. The AspectJ compiler can be used like Javac to compile Java sources, but it can also compile AspectJ sources or weave binary aspects with Java bytecode. It can run in normal "batch" mode or in an "incremental" mode, where it only recompiles files it has to revisit. For more information on ajc, see `ajc`, the AspectJ compiler/weaver. Unlike Javac or the Javac Ant task, this task always compiles the specified files since aspects can apply to other (updated) files. For a workaround, see Avoiding clean compiles.

Beyond the normal ajc compiler options, this task also supports an experimental option for an incremental "tag" file, and it can copy resources from source directories or input jars to the output jar or directory.

This task is named iajc to avoid conflict with the 1.0 task ajc.

AjcTask (iajc) Options

The following tables list the supported parameters. For any parameter specified as a Path, a single path can be specified directly as an attribute, multiple paths can be specified using a nested element of the same name, and a common path can be reused by defining it as a global and passing the id to the corresponding {name}ref attribute. See Path below for more details.

Most attributes and nested elements are optional. The compiler requires that the same version of `aspectjrt.jar` be specified on the classpath, and that some sources be be specified (using one or more of `sourceroots`, `injars`, `inpath`, `argfiles`, and/or `srcdir` (with patterns)). When in incremental mode, only `sourceroots` may be specified.

Boolean parameters default to `false` unless otherwise stated.

AjcTask (iajc) options for specifying sources

Attribute	Description
argfiles, argfilesRef (Path)	An argument file contains a list of arguments read by the compiler. Each line is read into one element of the argument array and may include another argfile by reference.
sourceRoots, sourceRootsRef (Path)	Directories containing source files (ending with .java or .aj) to compile.

Attribute	Description
srcdir (Path)	Base directory of sources to compile, assuming there are nested matches. This approach uses the Ant process for matching .java files and is not compatible with incremental mode. Unless using filters to limit the sources included, use sourceroots instead.
injars, injarsRef (Path)	Deprecated - use inpath instead. Read .class files for bytecode weaving from zip files (only).
inpath, inpathRef (Path)	Read .class files for bytecode weaving from directories or zip files (like classpath).
classpath, classpathRef (Path)	The classpath used by the sources being compiled. When compiling aspects, include the same version of the `aspectjrt.jar`.
bootclasspath, bootclasspathRef (Path)	The bootclasspath specifies types to use instead of the invoking VM's when seeking types during compilation.
extDirs, extDirsRef (Path)	The extension directories to use instead of those in the invoking VM when seeking types during compilation.
aspectPath, aspectPathRef (Path)	Similar to classpath, aspectpath contains read-only, binary aspect libraries that are woven into sources but not included in the output. `aspectpath` accepts jar/zip files (but, unlike classpath, not directories).

AjcTask (iajc) options for specifying output

Attribute	Description
destDir	The directory in which to place the generated class files. Only one of `destDir` and `outJar` may be set.
outJar	The zip file in which to place the generated output class files. Only one of `destDir` and `outJar` may be set.
copyInjars	(Deprecated/ignored; ajc does this.) If true, copy all non-.class files from input jar(s) to the output jar or destination directory after the compile (or incremental compile) completes. In forked mode, this copies only after the process completes, not after incremental compiles.

Attribute	Description
sourceRootCopyFilter	When set, copy all files from the sourceroot directories to the output jar or destination directory except those specified in the filter pattern. The pattern should be compatible with an Ant fileset excludes filter; when using this, most developers pass `/CVS/,/.java` to exclude any CVS directories or source files. See `inpathDirCopyFilter`. Requires `destDir` or `outJar`.
inpathDirCopyFilter	When set, copy all files from the inpath directories to the output jar or destination directory except those specified in the filter pattern. The pattern should be compatible with an Ant fileset excludes filter; when using this, most developers pass `/CVS/,/.java,/.class to exclude any CVS directories, source files, or unwoven .class files. (If/.class` is not specified, it will be prepended to the filter.) See `sourceRootCopyFilter`. (Note that ajc itself copies all resources from input jar/zip files on the inpath.) Requires `destDir` or `outJar`.

AjcTask (iajc) options for specifying compiler behavior

Attribute	Description
fork	Run process in another VM. This gets the forking classpath either explicitly from a `forkclasspath` entry or by searching the task or system/Ant classpath for the first readable file with a name of the form `aspectj{-}tools{.*}.jar`. When forking you can specify the amount of memory used with `maxmem`. Fork cannot be used in incremental mode, unless using a tag file.
forkclasspath, forkclasspathRef (Path)	Specify the classpath to use for the compiler when forking.
maxmem	The maximum memory to use for the new VM when fork is true. Values should have the same form as accepted by the VM, e.g., "128m".
incremental	incremental mode: Build once, then recompile only required source files when user provides input. Requires that source files be specified only using `sourceroots`. Incompatible with forking.

Attribute	Description
tagfile	incremental mode: Build once, then recompile only required source files when the tag file is updated, finally exiting when tag file is deleted. Requires that source files be specified only using `sourceroots`.
X	Set experimental option(s), using comma-separated list of accepted options Options should not contain the leading X. Some commonly-used experimental options have their own entries. The other permitted ones (currently) are serializableAspects, incrementalFile, lazyTjp, reweavable, notReweavable, noInline, terminateAfterCompilation, ajruntimelevel:1.2, and ajruntimelevel:1.5. Of these, some were deprecated in AspectJ 5 (reweavable, terminateAfterCompilation, etc.).
XterminateAfterCompilation	Terminates before the weaving process, dumping out unfinished class files.

AjcTask (iajc) options for specifying compiler side-effects and messages

Attribute	Description
emacssym	If true, emit `.ajesym` symbol files for Emacs support.
crossref	If true, emit `.ajsym` file into the output directory.
verbose	If true, log compiler verbose messages as Project.INFO during the compile.
logCommand	If true, log compiler command elements as Project.INFO (rather than the usual Project.VERBOSE level).
Xlistfileargs	If true, emit list of file arguments during the compile (but behaves now like verbose).
version	If true, do not compile - just print AspectJ version.
help	If true, just print help for the command-line compiler.
Xlintwarnings	Same as `xlint:warning`: if true, set default level of all language usage messages to warning.
Xlint	Specify default level of all language usage messages to one of [`error warning ignore`].

Attribute	Description
XlintFile	Specify property file containing `name:level` associations setting level for language messages emitted during compilation. Any levels set override the default associations in `org/aspectj/weaver/XLintDefault.properties`.
failonerror	If true, throw BuildException to halt build if there are any compiler errors. If false, continue notwithstanding compile errors. Defaults to `true`.
messageHolderClass	Specify a class to use as the message holder for the compile process. The entry must be a fully-qualified name of a class resolveable from the task classpath complying with the `org.aspectj.bridge.IMessageHolder` interface and having a public no-argument constructor.
showWeaveInfo	If true, emit weaver messages. Defaults to `false`.

AjcTask (iajc) options for specifying Eclipse compiler options

Attribute	Description
nowarn	If true, same as `warn:none`.
deprecation	If true, same as `warn:deprecation`
warn	One or more comma-separated warning specifications from [`constructorName packageDefaultMethod deprecation, maskedCatchBlocks unusedLocals unusedArguments, unusedImports syntheticAccess assertIdentifier`].
debug	If true, same as `debug:lines,vars,source`
debugLevel	One or more comma-separated debug specifications from [`lines vars source`].
PreserveAllLocals	If true, code gen preserves all local variables (for debug purposes).
noimporterror	If true, emit no errors for unresolved imports.
referenceinfo	If true, compute reference info.
log	File to log compiler messages to.
encoding	Default source encoding format (per-file encoding not supported in Ant tasks).
proceedOnError	If true, keep compiling after errors encountered, dumping class files with problem methods.
progress	If true, emit progress (requires log).

Attribute	Description
time	If true, display speed information.
target	Specify target class file format as one of [1.1 1.2]. Defaults to 1.1 class file.
source	Set source compliance level to one of [1.3 1.4 1.5] (default is 1.4). 1.3 implies -source 1.3 and -target 1.1. 1.4 implies -source 1.4 and -target 1.2. 1.5 implies -source 1.5 and -target 1.5.
source	Set source assertion mode to one of [1.3 1.4]. Default depends on compliance mode.

AjcTask matching parameters specified as nested elements

This task forms an implicit FileSet and supports all attributes of `fileset` (dir becomes srcdir) as well as the nested `include`, `exclude`, and `patternset` elements. These can be used to specify source files. However, it is better to use `sourceroots` to specify source directories unless using filters to exclude some files from compilation.

AjcTask Path-like Structures

Some parameters are path-like structures containing one or more elements; these are `sourceroots`, `argfiles`, `injars`, `inpath`, `classpath`, `bootclasspath`, `forkclasspath`, and `aspectpath`. In all cases, these may be specified as nested elements, something like this:

```
<iajc {attributes..} />
    <{name}>
        <pathelement path="{first-location}"/>
        <pathelement path="{second-location}"/>
        ...
    <{name}>
    ...
</iajc>
```

As with other Path-like structures, they may be defined elsewhere and specified using the refid attribute:

```
<path id="aspect.path">
    <pathelement path="${home}/lib/persist.jar"/>
    <pathelement path="${home}/lib/trace.jar"/>
</path>
...
<iajc {attributes..} />
    <aspectpath refid="aspect.path"/>
    ...
</iajc>
```

The task also supports an attribute `{name}ref` for each such parameter. E.g., for `aspectpath`:

```
<iajc {attributes..} aspectpathref="aspect.path"/>
```

Sample of iajc task

A minimal build script defines the task and runs it, specifying the sources:

```
<project name="simple-example" default="compile" >
  <taskdef
      resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
    <classpath>
      <pathelement location="${home.dir}/tools/aspectj/lib/aspectjtools.jar"/>
    </classpath>
  </taskdef>

  <target name="compile" >
    <iajc sourceroots="${home.dir}/ec/project/src"
        classpath="${home.dir}/tools/aspectj/lib/aspectjrt.jar"/>
  </target>
</project>
```

Below is script with most everything in it. The compile process...

1. Runs in incremental mode, recompiling when the user hits return;
2. Reads all the source files from two directories;
3. Reads binary .class files from input jar and directory;
4. Uses a binary aspect library for persistence;
5. Outputs to an application jar; and
6. Copies resources from the source directories and binary input jar and directories to the application jar.

When this target is built, the compiler will build once and then wait for input from the user. Messages are printed as usual. When the user has quit, then this runs the application.

```
<target name="build-test" >
  <iajc outjar="${home.dir}/output/application.jar"
          sourceRootCopyFilter="**/CVS/*,**/*.java"
          inpathDirCopyFilter="**/CVS/*,**/*.java,**/*.class"
          incremental="true" >
    <sourceroots>
      <pathelement location="${home.dir}/ec/project/src"/>
      <pathelement location="${home.dir}/ec/project/testsrc"/>
    </sourceroots>
    <inpath>
```

```
      <pathelement location="${home.dir}/build/module.jar"/>
      <pathelement location="${home.dir}/build/binary-input"/>
    </inpath>
    <aspectpath>
      <pathelement location="${home.dir}/ec/int/persist.jar"/>
    </aspectpath>
    <classpath>
      <pathelement location="${home.dir}/tools/aspectj/lib/aspectjrt.jar"/>
    </classpath>
  </iajc>

  <java classname="org.smart.app.Main">
    <classpath>
      <pathelement location="${home.dir}/tools/aspectj/lib/aspectjrt.jar"/>
      <pathelement location="${home.dir}/ec/int/persist.jar"/>
      <pathelement location="${home.dir}/output/application.jar"/>
    </classpath>
  </java>
</target>
```

For an example of a build script, see ../examples/build.xml.

Avoiding clean compiles

Unlike javac, the ajc compiler always processes all input because new aspects can apply to updated classes and vice-versa. However, in the case where no files have been updated, there is no reason to recompile sources. One way to implement that is with an explicit dependency check using the uptodate task:

```
<target name="check.aspects.jar">
  <uptodate property="build.unnecessary"
      targetfile="${aspects.module-jar}" >
     <srcfiles dir="${src1}" includes="**/*.aj"/>
     <srcfiles dir="${src2}/" includes="**/*.aj"/>
  </uptodate>
</target>

<target name="compile.aspects" depends="prepare,check.aspects.jar"
        unless="build.unnecessary">
   <iajc ...
```

When using this technique, be careful to verify that binary input jars are themselves up-to-date after they would have been modified by any build commands.

Programmatically handling compiler messages

Users may specify a message holder to which the compiler will pass all messages as they are generated. This will override all of the normal message printing, but does not prevent the task from

failing if exceptions were thrown or if failonerror is true and the compiler detected errors in the sources.

Handling messages programmatically could be useful when using the compiler to verify code. If aspects consist of declare [error|warning], then the compiler can act to detect invariants in the code being processed. For code to compare expected and actual messages, see the AspectJ testing module (which is not included in the binary distribution).

Ajc11CompilerAdapter (javac)

This CompilerAdapter can be used in javac task calls by setting the `build.compiler` property. This enables users to to easily switch between the Javac and AspectJ compilers. However, because there are differences in source file handling between the Javac task and the ajc compiler, not all Javac task invocations can be turned over to iajc. However, ajc can compile anything that Javac can, so it should be possible for any given compile job to restate the Javac task in a way that can be handled by iajc/ajc.

Sample of compiler adapter

To build using the adapter, put the `aspectjtools.jar` on the system/ant classpath (e.g., in `${ANT_HOME}/lib`) and define the `build.compiler` property as the fully-qualified name of the class, `org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter`.

The AspectJ compiler should run for any compile using the Javac task (for options, see the Ant documentation for the Javac task). For example, the call below passes all out-of-date source files in the `src/org/aspectj` subdirectories to the `ajc` command along with the destination directory:

```
-- command:

    cp aspectj1.1/lib/aspectjtools.jar ant/lib
    ant/bin/ant -Dbuild.compiler=org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter
...

-- task invocation in the build script:

  <javac srcdir="src" includes="org/aspectj/**/*.java" destdir="dest" />
```

To pass ajc-specific arguments, use a compilerarg entry.

```
-- command

  Ant -Dbuild.compiler=org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter

-- build script

  <property name="ajc"
            value="org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter"/>
```

```
<javac srcdir="src" includes="org/aspectj/**/*.java" destdir="dest" >
  <compilerarg compiler="${ajc}" line="-argfile src/args.lst"/>
<javac/>
```

The Javac task does special handling of source files that can interfere with ajc. It removes any files that are not out-of-date with respect to the corresponding .class files. But ajc requires all source files, since an aspect may affect a source file that is not out of date. (For a solution to this, see the `build.compiler.clean` property described below.) Conversely, developers sometimes specify a source directory to javac, and let it search for files for types it cannot find. AspectJ will not do this kind of searching under the source directory (since the programmer needs to control which sources are affected). (Don't confuse the source directory used by Javac with the source root used by ajc; if you specify a source root to ajc, it will compile any source file under that source root (without exception or filtering).) To replace source dir searching in Javac, use an Ant filter to specify the source files.

Compiler adapter compilerarg options

The adapter supports any ajc command-line option passed using compilerarg, as well as the following options available only in AjcTask. Find more details on the following options in AjcTask (iajc).

- `-Xmaxmem`: set maximum memory for forking (also settable in javac).
- `-Xlistfileargs`: list file arguments (also settable in javac).
- `-Xfailonerror`: throw BuildException on compiler error (also settable in javac).
- `-Xmessageholderclass`: specify fully-qualified name of class to use as the message holder.
- `-Xcopyinjars`: copy resources from any input jars to output (default behavior since 1.1.1)
- `-Xsourcerootcopyfilter {filter}`: copy resources from source directories to output (minus files specified in filter)
- `-Xtagfile {file}`: use file to control incremental compilation
- `-Xsrcdir {dir}`: add to list of ajc source roots (all source files will be included).

Special considerations when using Javac and compilerarg:

- The names above may differ slightly from what you might expect from AjcTask; use these forms when specifying compilerarg.
- By default the adapter will mimic the Javac task's copying of resource files by specifying `"/CVS/,/.java,/.aj" for the sourceroot copy filter. To change this behavior, supply your own value (e.g., "/"` to copy nothing).
- Warning - define the system property `build.compiler.clean` to compile all files, when available. Javac prunes the source file list of "up-to-date" source files based on the timestamps of corresponding .class files, and will not compile if no sources are out of date. This is wrong for ajc which requires all the files for each compile and which may refer indirectly to sources using argument files.

  To work around this, set the global property `build.compiler.clean`. This tells the compiler adapter to delete all .class files in the destination directory and re-execute the javac task so

javac can recalculate the list of source files. e.g.,

```
Ant -Dbuild.compiler=org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter
    -Dbuild.compiler.clean=anything ...
```

Caveats to consider when using this global `build.compiler.clean` property:

1. If javac believes there are no out-of-date source files, then the adapter is never called and cannot clean up, and the "compile" will appear to complete successfully though it did nothing.
2. Cleaning will makes stepwise build processes fail if they depend on the results of the prior compilation being in the same directory, since cleaning deletes all .class files.
3. This clean process only permits one compile process at a time for each destination directory because it tracks recursion by writing a tag file to the destination directory.
4. When running incrementally, the clean happens only before the initial compile.

Ajc10 (ajc)

This task handles the same arguments as those used by the AspectJ 1.0 task. This should permit those with existing build scripts using the Ajc Ant task to continue using the same scripts when compiling with 1.1. This will list any use of options no longer supported in 1.1 (e.g., `lenient, strict, workingdir, preprocess, usejavac`,...), and does not provide access to the new features of AspectJ 1.1. (Developers using AspectJ 1.1 only should upgrade their scripts to use AjcTask instead. This will not work for AspectJ 1.2 or later.)

Ajc10 (ajc) Options

Most attributes and nested elements are optional. The compiler requires that the same version of `aspectjrt.jar` be specified on the classpath, and that some sources be be specified (using one or more of `argfiles` and `srcdir` (with patterns)).

Boolean parameters default to `false` unless otherwise stated.

Table 1. AjcTask (ajc) options for specifying sources

Attribute	Description
srcdir	The base directory of the java files. See
destdir	The target directory for the output .class files
includes	Comma-separated list of patterns of files that must be included. No files are included when omitted.
includesfile	The path to a file containing include patterns.
excludes	Comma-separated list of patterns of files that must be excluded. No files (except default excludes) are excluded when omitted.

Attribute	Description
excludesfile	The path to a file containing exclude patterns.
defaultexcludes	If true, then default excludes are used. Default excludes are used when omitted (i.e., defaults to `true`).
classpath, classpathref	The classpath to use, optionally given as a reference to a classpath Path element defined elsewhere.
bootclasspath, bootclasspathref	The bootclasspath to use, optionally given as a reference to a bootclasspath Path element defined elsewhere.
extdirs	Paths to directories containting installed extensions.
debug	If true, emit debug info in the .class files.
deprecation	If true, emit messages about use of deprecated API.
verbose	Emit compiler status messages during the compile.
version	Emit version information and quit.
failonerror	If true, throw BuildException to halt build if there are any compiler errors. If false, continue notwithstanding compile errors. Defaults to `true`.
source	Value of -source option - ignored unless `1.4`.

Table 2. Parameters ignored by the old ajc taskdef, but now supported or buggy

Attribute	Description	Supported?
encoding	Default encoding of source files.	yes
optimize	Whether source should be compiled with optimization.	yes?
target	Generate class files for specific VM version, one of [`1.1` `1.2`].	yes
depend	Enables dependency-tracking.	no
includeAntRuntime	Whether to include the Ant run-time libraries.	no
includeJavaRuntime	Whether to include the run-time libraries from the executing VM.	no
threads	Multi-threaded compilation	no

The following table shows that many of the unique parameters in AspectJ 1.0 are no longer supported.

Table 3. Parameters unique to ajc

Attribute	Description
X	deprecated X options include reweavable (on by default) reweavable:compress (compressed by default)
emacssym	Generate symbols for Emacs IDE support.
argfiles	A comma-delimited list of argfiles that contain a line-delimited list of source file paths (absolute or relative to the argfile).

argfiles - argument list files

An argument file is a file (usually `{file}.lst`) containing a list of source file paths (absolute or relative to the argfile). You can use it to specify all source files to be compiled, which ajc requires to avoid searching every possible source file in the source path when building aspects. If you specify an argfile to the ajc task, it will not include all files in any specified source directory (which is the default behavior for the Javac task when no includes are specified). Conversely, if you specify excludes, they will be removed from the list of files compiled even if they were specified in an argument file.

The compiler also accepts arguments that are not source files, but the IDE support for such files varies, and Javac does not support them. Be sure to include exactly one argument on each line.

Ajc10 parameters specified as nested elements

This task forms an implicit FileSet and supports all attributes of `fileset` (dir becomes srcdir) as well as the nested `include`, `exclude`, and `patternset` elements. These can be used to specify source files.

`ajc`'s `srcdir`, `classpath`, `bootclasspath`, `extdirs`, and `jvmarg` attributes are path-like structures and can also be set via nested `src`, `classpath`, `bootclasspath`, `extdirs`, and `jvmargs` elements, respectively.

Sample of ajc task

Following is a declaration for the ajc task and a sample invocation that uses the ajc compiler to compile the files listed in `default.lst` into the dest dir:

```
<project name="example" default="compile" >
  <taskdef name="ajc"
    classname="org.aspectj.tools.ant.taskdefs.Ajc10" >
    <!-- declare classes needed to run the tasks and tools -->
    <classpath>
      <pathelement location="${home.dir}/tools/aspectj/lib/aspectjtools.jar"/>
    </classpath>
  </taskdef>
```

```
  <target name="compile" >
    <mkdir dir="dest" />
    <ajc destdir="dest" argfiles="default.lst" >
      <!-- declare classes needed to compile the target files -->
      <classpath>
        <pathelement location="${home.dir}/tools/aspectj/lib/aspectjrt.jar"/>
      </classpath>
    </ajc>
  </target>
</project>
```

This build script snippet

```
<ajc srcdir="${src}"
     destdir="${build}"
     argfiles="demo.lst"
/>
```

compiles all .java files specified in the demo.lst and stores the .class files in the ${build} directory. Unlike the Javac task, the includes attribute is empty by default, so only those files specified in demo.lst are included.

This next example

```
<ajc srcdir="${src}"
     destdir="${build}"
     includes="spacewar/*,coordination/*"
     excludes="spacewar/Debug.java"
/>
```

compiles .java files under the `${src}` directory in the spacewar and coordination packages, and stores the .class files in the `${build}` directory. All source files under spacewar/ and coordination/ are used, except Debug.java.

See ../examples/build.xml for an example build script.

Isolating problems running the Ant tasks

If you have problems with the tasks not solved by the documentation, please try to see if you have the same problems when running ajc directly on the command line.

- If the problem occurs on the command line also, then the problem is not in the task. (It may be in the tools; please send bug reports.)
- If the problem does not occur on the command line, then it may lie in the parameters you are supplying in Ant or in the task's handling of them.

- If the build script looks correct and the problem only occurs when building from Ant, then please send a report (including your build file, if possible).

Known issues with the Ant tasks

For the most up-to-date information on known problems, see the bug database for unresolved compiler bugs or taskdef bugs .

When running Ant build scripts under Eclipse 2.x variants, you will get a VerifyError because the Eclipse Ant support fails to isolate the Ant runtime properly. To run in this context, set up iajc to fork (and use forkclasspath). Eclipse 3.0 will fork Ant processes to avoid problems like this.

Memory and forking: Users email most often about the ajc task running out of memory. This is not a problem with the task; some compiles take a lot of memory, often more than similar compiles using javac.

Forking is now supported in both the Ajc11CompilerAdapter (javac) and AjcTask (iajc), and you can set the maximum memory available. You can also not fork and increase the memory available to Ant (see the Ant documentation, searching for ANT_OPTS, the variable they use in their scripts to pass VM options, e.g., ANT_OPTS=-Xmx128m).

Ant task questions and bugs

For questions, you can send email to aspectj-users@dev.eclipse.org. (Do join the list to participate!) We also welcome any bug reports, patches, and features; you can submit them to the bug database at https://bugs.eclipse.org/bugs using the AspectJ product and Ant component.

Load-Time Weaving

Introduction

The AspectJ weaver takes class files as input and produces class files as output. The weaving process itself can take place at one of three different times: compile-time, post-compile time, and load-time. The class files produced by the weaving process (and hence the run-time behaviour of an application) are the same regardless of the approach chosen.

- Compile-time weaving is the simplest approach. When you have the source code for an application, ajc will compile from source and produce woven class files as output. The invocation of the weaver is integral to the ajc compilation process. The aspects themselves may be in source or binary form. If the aspects are required for the affected classes to compile, then you must weave at compile-time. Aspects are required, e.g., when they add members to a class and other classes being compiled reference the added members.
- Post-compile weaving (also sometimes called binary weaving) is used to weave existing class files and JAR files. As with compile-time weaving, the aspects used for weaving may be in source or binary form, and may themselves be woven by aspects.
- Load-time weaving (LTW) is simply binary weaving defered until the point that a class loader loads a class file and defines the class to the JVM. To support this, one or more "weaving class loaders", either provided explicitly by the run-time environment or enabled through a "weaving agent" are required.

You may also hear the term "run-time weaving". We define this as the weaving of classes that have already been defined to the JVM (without reloading those classes). AspectJ 5 does not provide explicit support for run-time weaving although simple coding patterns can support dynamically enabling and disabling advice in aspects.

Weaving class files more than once

As of AspectJ 5 aspects (code style or annotation style) and woven classes are reweavable by default. If you are developing AspectJ applications that are to be used in a load-time weaving environment with an older version of the compiler you need to specify the `-Xreweavable` compiler option when building them. This causes AspectJ to save additional state in the class files that is used to support subsequent reweaving.

Load-time Weaving Requirements

All load-time weaving is done in the context of a class loader, and hence the set of aspects used for weaving and the types that can be woven are affected by the class loader delegation model. This ensures that LTW complies with the Java 2 security model. The following rules govern the interaction of load-time weaving with class loading:

1. All aspects to be used for weaving must be defined to the weaver before any types to be woven are loaded. This avoids types being "missed" by aspects added later, with the result that invariants across types fail.

2. All aspects visible to the weaver are usable. A visible aspect is one defined by the weaving class loader or one of its parent class loaders. All concrete visible aspects are woven and all abstract visible aspects may be extended.
3. A class loader may only weave classes that it defines. It may not weave classes loaded by a delegate or parent class loader.

Configuration

New in AspectJ 5 are a number of mechanisms to make load-time weaving easy to use. The load-time weaving mechanism is chosen through JVM startup options. Configuration files determine the set of aspects to be used for weaving and which types will be woven. Additional diagnostic options allow the user to debug the configuration and weaving process.

Enabling Load-time Weaving

AspectJ 5 supports several ways of enabling load-time weaving for an application: agents, a command-line launch script, and a set of interfaces for integration of AspectJ load-time weaving in custom environments.

Agents

AspectJ 5 ships with a load-time weaving agent that enables load-time weaving. This agent and its configuration is execution environment dependent. Configuration for the supported environments is discussed later in this chapter.

Using Java 5 JVMTI you can specify the `-javaagent:pathto/aspectjweaver.jar` option to the JVM.

Since AspectJ 1.9.7, the obsolete Oracle/BEA JRockit agent is no longer part of AspectJ. JRockit JDK never supported Java versions higher than 1.6. Several JRockit JVM features are now part of HotSpot and tools like Mission Control available for OpenJDK and Oracle JDK.

Command-line wrapper scripts `aj`

The `aj` command runs Java programs in Java 1.4 or later by setting up `WeavingURLClassLoader` as the system class loader. For more information, see `aj`, the AspectJ load-time weaving launcher.

The `aj5` command runs Java programs in Java 5 by using the `-javaagent:pathto/aspectjweaver.jar` option described above. For more information, see `aj`, the AspectJ load-time weaving launcher.

Custom class loader

A public interface is provided to allow a user written class loader to instantiate a weaver and weave classes after loading and before defining them in the JVM. This enables load-time weaving to be supported in environments where no weaving agent is available. It also allows the user to explicitly restrict by class loader which classes can be woven. For more information, see `aj`, the AspectJ load-time weaving launcher and the API documentation and source for `WeavingURLClassLoader` and `WeavingAdapter`.

Configuring Load-time Weaving with aop.xml files

The weaver is configured using one or more `META-INF/aop.xml` files located on the class loader search path. Each file may declare a list of aspects to be used for weaving, type patterns describing which types should woven, and a set of options to be passed to the weaver. In addition AspectJ 5 supports the definition of concrete aspects in XML. Aspects defined in this way must extend an abstract aspect visible to the weaver. The abstract aspect may define abstract pointcuts (but not abstract methods). The following example shows a simple aop.xml file:

```
<aspectj>

  <aspects>
    <!-- declare two existing aspects to the weaver -->
    <aspect name="com.MyAspect"/>
    <aspect name="com.MyAspect.Inner"/>

    <!-- define a concrete aspect inline -->
    <concrete-aspect name="com.xyz.tracing.MyTracing"
                     extends="tracing.AbstractTracing"
                     precedence="com.xyz.first, *">
      <pointcut name="tracingScope" expression="within(org.maw.*)"/>
    </concrete-aspect>

    <!-- Of the set of aspects declared to the weaver
         use aspects matching the type pattern "com..*" for weaving. -->
    <include within="com..*"/>

    <!-- Of the set of aspects declared to the weaver
         do not use any aspects with the @CoolAspect annotation for weaving -->
    <exclude within="@CoolAspect *"/>

  </aspects>

  <weaver options="-verbose">
    <!-- Weave types that are within the javax.* or org.aspectj.*
         packages. Also weave all types in the foo package that do
         not have the @NoWeave annotation. -->
    <include within="javax.*"/>
    <include within="org.aspectj.*"/>
    <include within="(!@NoWeave foo.*) AND foo.*"/>

    <!-- Do not weave types within the "bar" pakage -->
    <exclude within="bar.*"/>

    <!-- Dump all types within the "com.foo.bar" package
         to the "./_ajdump" folder on disk (for diagnostic purposes) -->
    <dump within="com.foo.bar.*"/>

    <!-- Dump all types within the "com.foo.bar" package and sub-packages,
         both before are after they are woven,
```

```
        which can be used for byte-code generated at runtime
    <dump within="com.foo.bar..*" beforeandafter="true"/>
  </weaver>

</aspectj>
```

The DTD defining the format of this file is available here: https://www.eclipse.org/aspectj/dtd/aspectj.dtd.

An aop.xml file contains two key sections: `aspects` defines one or more aspects to the weaver and controls which aspects are to be used in the weaving process; `weaver` defines weaver options and which types should be woven.

The simplest way to define an aspect to the weaver is to specify the fully-qualified name of the aspect type in an aspect element. You can also declare (and define to the weaver) aspects inline in the aop.xml file. This is done using the `concrete-aspect` element. A concrete-aspect declaration must provide a pointcut definition for every abstract pointcut in the abstract aspect it extends. This mechanism is a useful way of externalizing configuration for infrastructure and auxiliary aspects where the pointcut definitions themselves can be considered part of the configuration of the service. Refer to the next section for more details.

The `aspects` element may optionally contain one or more `include` and `exclude` elements (by default, all defined aspects are used for weaving). Specifying include or exclude elements restricts the set of defined aspects to be used for weaving to those that are matched by an include pattern, but not by an exclude pattern. The `within` attribute accepts a type pattern of the same form as a within pcd, except that && and || are replaced by 'AND' and 'OR'.

Note that `include` and `exclude` elements affect all aspects declared to the weaver including those in other aop.xml files. To help avoid unexpected behaviour a lint warning is issued if an aspect is not declared as a result of of applying these filters. Also note `aspect` and `concrete-aspect` elements must be used to declare aspects to the weaver i.e. `include` and `exclude` elements cannot be used find aspects on the class loader search path.

The `weaver` element is used to pass options to the weaver and to specify the set of types that should be woven. If no include elements are specified then all types visible to the weaver will be woven. In addition the `dump` element can be used capture on disk byte-code of woven classes for diagnostic purposes both before, in the case of those generated at runtime, and after the weaving process.

When several configuration files are visible from a given weaving class loader their contents are conceptually merged. The files are merged in the order they are found on the search path (with a regular `getResourceAsStream` lookup) according to the following rules:

- The set of available aspects is the set of all declared and defined aspects (`aspect` and `concrete-aspect` elements of the `aspects` section).
- The set of aspects used for weaving is the subset of the available aspects that are matched by at least one include statement and are not matched by any exclude statements. If there are no include statements then all non-excluded aspects are included.
- The set of types to be woven are those types matched by at least one weaver `include` element

and not matched by any weaver `exclude` element. If there are no weaver include statements then all non-excluded types are included.

- The weaver options are derived by taking the union of the options specified in each of the weaver options attribute specifications. Where an option takes a value e.g. `-warn:none` the most recently defined value will be used.

It is not an error for the same aspect to be defined to the weaver in more than one visible `META-INF/aop.xml` file. However, if the same concrete aspect is defined in more than one aop.xml file then an error will be issued. A concrete aspect defined in this way will be used to weave types loaded by the class loader that loaded the aop.xml file in which it was defined.

A `META-INF/aop.xml` can be generated by using either the `-outxml` or `-outxmlfile` options of the AspectJ compiler. It will simply contain a (possibly empty) set of aspect elements; one for each abstract or concrete aspect defined. When used in conjuction with the `-outjar` option a JAR is produced that can be used with the `aj5` command or a load-time weaving environment.

Using Concrete Aspects

It is possible to make an abstract aspect concrete by means of the `META-INF/aop.xml` file. This is useful way to implement abstract pointcuts at deployment time, and also gives control over precedence through the `precedence` attribute of the `concrete-aspect` XML element. Consider the following:

```
package mypack;

@Aspect
public abstract class AbstractAspect {

    // abstract pointcut: no expression is defined
    @Pointcut
    abstract void scope();

    @Before("scope() && execution(* *..doSome(..))")
    public void before(JoinPoint jp) {
       // ...
    }
}
```

This aspect is equivalent to the following in code style:

```
package mypack;

public abstract aspect AbstractAspect {

    // abstract pointcut: no expression is defined
    abstract pointcut scope();

    before() : scope() && execution(* *..doSome(..)) {
```

```
        // ...
    }
}
```

This aspect (in either style) can be made concrete using `META-INF/aop.xml`. It defines the abstract pointcut `scope()`. When using this mechanism the following rules apply:

- The parent aspect must be abstract. It can be an @AspectJ or a regular code style aspect.
- Only a simple abstract pointcut can be implemented i.e. a pointcut that doesn't expose state (through `args(), this(), target(), if()`). In @AspectJ syntax as illustrated in this sample, this means the method that hosts the pointcut must be abstract, have no arguments, and return void.
- The concrete aspect must implement all inherited abstract pointcuts.
- The concrete aspect may not implement methods so the abstract aspect it extends may not contain any abstract methods.

A limitation of the implementation of this feature in AspectJ 1.5.0 is that aspects defined using aop.xml are not exposed to the weaver. This means that they are not affected by advice and ITDs defined in other aspects. Support for this capability will be considered in a future release.

If more complex aspect inheritance is required use regular aspect inheritance instead of XML. The following XML definition shows a valid concrete sub-aspect for the abstract aspects above:

```
<aspectj>
    <aspects>
        <concrete-aspect name="mypack.__My__AbstractAspect" extends=
"mypack.AbstractAspect">
            <pointcut name="scope" expression="within(yourpackage..*)"/>
        </concrete-aspect>
    <aspects>
</aspectj>
```

It is important to remember that the `name` attribute in the `concrete-aspect` directive defines the fully qualified name that will be given to the concrete aspect. It must a valid class name because the aspect will be generated on the fly by the weaver. You must also ensure that there are no name collisions. Note that the concrete aspect will be defined at the classloader level for which the aop.xml is visible. This implies that if you need to use the `aspectof` methods to access the aspect instance(s) (depending on the perclause of the aspect it extends) you have to use the helper API `org.aspectj.lang.Aspects.aspectOf(..)` as in:

```
// exception handling omitted
Class myConcreteAspectClass = Class.forName("mypack.__My__AbstractAspect");

// here we are using a singleton aspect
AbstractAspect concreteInstance = Aspects.aspectOf(myConcreteAspectClass);
```

Using Concrete Aspects to define precedence

As described in the previous section, the `concrete-aspect` element in `META-INF/aop.xml` gives the option to declare the precedence, just as `@DeclarePrecedence` or `declare precedence` do in aspect source code.

Sometimes it is necessary to declare precedence without extending any abstract aspect. It is therefore possible to use the `concrete-aspect` element without the `extends` attribute and without any `pointcut` nested elements, just a `precedence` attribute. Consider the following:

```
<aspectj>
    <aspects>
        <concrete-aspect name="mypack.__MyDeclarePrecedence"
                         precedence="*..*Security*, Logging+, *"/>
    </aspects>
</aspectj>
```

This deployment time definitions is only declaring a precedence rule. You have to remember that the `name` attribute must be a valid fully qualified class name that will be then reserved for this concrete-aspect and must not conflict with other classes you deploy.

Weaver Options

The table below lists the AspectJ options supported by LTW. All other options will be ignored and a warning issued.

Option	Purpose
`-verbose`	Issue informational messages about the weaving process. Messages issued while the weaver is being bootstrapped are accumulated until all options are parsed. If the messages are required to be output immediately you can use the option `-Daj.weaving.verbose=true` on the JVM startup command line.
`-debug`	Issue a messages for each class passed to the weaver indicating whether it was woven, excluded or ignored. Also issue messages for classes defined during the weaving process such as around advice closures and concrete aspects defined in `META-INF/aop.xml`.
`-showWeaveInfo`	Issue informational messages whenever the weaver touches a class file. This option may also be enabled using the System property `-Dorg.aspectj.weaver.showWeaveInfo=true`.

Option	Purpose
`-Xlintfile:pathToAResource`	Configure lint messages as specified in the given resource (visible from this aop.xml file' classloader)
`-Xlint:default, -Xlint:ignore, ⋯`	Configure lint messages, refer to documentation for meaningfull values
`-nowarn, -warn:none`	Suppress warning messages
`-Xreweavable`	Produce class files that can subsequently be rewoven
`-XnoInline`	Don't inline around advice.
`-XmessageHandlerClass:⋯`	Provide alternative output destination to stdout/stderr for all weaver messages. The given value must be the full qualified class name of a class that implements the `org.aspectj.bridge.IMessageHandler` interface and is visible to the classloader with which the weaver being configured is associated. Exercise caution when packaging a custom message handler with an application that is to be woven. The handler (as well as classes on which it depends) cannot itself be woven by the aspects that are declared to the same weaver.

Special cases

The following classes are not exposed to the LTW infrastructure regardless of the `aop.xml` file(s) used:

- All `org.aspectj.*` classes (and subpackages) - as those are needed by the infrastructure itself
- All `java. and javax.` classes (and subpackages)
- All `sun.reflect.*` classes - as those are JDK specific classes used when reflective calls occurs

Despite these restrictions, it is perfectly possible to match call join points for calls to these types providing the calling class is exposed to the weaver. Subtypes of these excluded types that are exposed to the weaver may of course be woven.

Note that dynamic proxy representations are exposed to the LTW infrastructure and are not considered a special case.

Some lint options behave differently when used under load-time weaving. The `adviceDidNotMatch` won't be handled as a warn (as during compile time) but as an info message.

Runtime Requirements for Load-time Weaving

To use LTW the `aspectjweaver.jar` library must be added to the classpath. This contains the AspectJ 5 runtime, weaver, weaving class loader and weaving agents. It also contains the DTD for parsing XML weaving configuration files.

Supported Agents

JVMTI

When using Java 5 the JVMTI agent can be used by starting the JVM with the following option (adapt according to the path to aspectjweaver.jar):

```
-javaagent:pathto/aspectjweaver.jar
```

JRockit with Java 1.3/1.4 (use JVMTI on Java 5)

Since AspectJ 1.9.7, the obsolete Oracle/BEA JRockit agent is no longer part of AspectJ. JRockit JDK never supported Java versions higher than 1.6. Several JRockit JVM features are now part of HotSpot and tools like Mission Control available for OpenJDK and Oracle JDK.

AspectJ version compatibility

Version Compatibility

Systems, code, and build tools change over time, often not in step. Generally, later versions of the build tools understand earlier versions of the code, but systems should include versions of the runtime used to build the AspectJ program.

Java compatibility

AspectJ programs can run on any Java VM of the required version. The AspectJ tools produce Java bytecode .class files that run on Java compatible VM's. If a Java class is changed by an aspect, the resulting class is binary compatible (as defined in the Java Language Specification). Further, the AspectJ compiler and weaving do all the exception checking required of Java compilers by the Java specifications.

Like other Java compilers, the AspectJ compiler can target particular Java versions. Obviously, code targeted at one version cannot be run in a VM of a lesser version. The `aspectjrt.jar` is designed to take advantage of features available in Java 2 or Java 5, but will run in a JDK 1.1.x environment, so you can use AspectJ to target older or restricted versions of Java. However, there may be restricted variants of JDK 1.1.x that do not have API's used by the AspectJ runtime. If you deploy to one of those, you can email aspectj-dev@eclipse.org or download the runtime code to modify it for your environment.

Aside from the runtime, running the AspectJ tools themselves will require a more recent version of Java. You might use Java 5 to run the AspectJ compiler to produce code for Java 1.1.8.

Runtime library compatibility

When deploying AspectJ programs, include on the classpath the classes, aspects, and the AspectJ runtime library (`aspectjrt.jar`). Use the version of the runtime that came with the tools used to build the program. If the runtime is earlier than the build tools used, it's very likely to fail. If the runtime is later than the build tools used, it's possible (but not guaranteed) that it will work.

Given that, three scenarios cause problems. First, you deploy new aspects into an an existing system that already has aspects that were built with a different version. Second, the runtime is already deployed in your system and cannot be changed (e.g., some application servers put `aspectjrt.jar` on the bootclasspath). Third, you (unintentionally) deploy two versions of the runtime, and the one loaded by a parent loader is used).

In earlier versions of AspectJ, these problems present in obscure ways (e.g., unable to resolve a class). In later versions, a stack trace might even specify that the runtime version is out of sync with an aspect. To find out if the runtime you deployed is the one actually being used, log the defining class loader for the aspects and runtime.

Aspect binary compatibility

Generally, binary aspects can be read by later versions of the weaver if the aspects were built by

version 1.2.1 or later. (Some future weavers might have documented limitations in how far back they go.) If a post-1.2.1 weaver reads an aspect built by a later version, it will emit a message. If the weaver reads in a binary aspect and writes it out again, the result will be in the form produced by that weaver, not the original form of the aspect (just like other weaver output).

With unreleased or development versions of the tools, there are no guarantees for binary compatibility, unless they are stated in the release notes. If you use aspects built with development versions of the weaver, be careful to rebuild and redeploy with the next released version.

Aspect source compatibility

Generally, AspectJ source files can be read by later versions of the compiler. Language features do not change in dot releases (e.g., from 1.2.1 to 1.2.2). In some very rare cases, a language feature will no longer be supported or may change its meaning; these cases are documented in the release notes for that version. Some changes like this were necessary when moving to binary weaving in the 1.1 release, but at this time we don't anticipate more in the future. You might also find that the program behaves differently if you relied on behavior specific to that compiler/weaver, but which is not specified in the Semantics appendix to the Programming Guide.

Problems when upgrading to new AspectJ versions

Let's say your program behaves differently after being built with a new version of the AspectJ tools. It could be a bug that was introduced by the tools, but often it results from relying on behavior that was not guaranteed by the compiler. For example, the order of advice across two aspects is not guaranteed unless there is a precedence relationship between the aspects. If the program implicitly relies on a certain order that obtains in one compiler, it can fail when built with a different compiler.

Another trap is deploying into the same system, when the `aspectjrt.jar` has not been changed accordingly.

Finally, when updating to a version that has new language features, there is a temptation to change both the code and the tools at the same time. It's best to validate the old code with the new tools before updating the code to use new features. That distinguishes problems of new engineering from those of new semantics.