Apache Ant User Manual Guide

Version 1.6.0

Compiled by: Abdul Habra (www.tek271.com)
From: http://ant.apache.org/

12/2003

Blank Page

Contents At A Glance

Table Of Contents

1 Apache Ant User Manual Authors

- Stephane Bailliez (sbailliez@imediation.com)
- Nicola Ken Barozzi (nicolaken@apache.org)
- Jacques Bergeron (jacques.bergeron@dogico.com)
- Stefan Bodewig (stefan.bodewig@freenet.de)
- Patrick Chanezon (chanezon@netscape.com)
- James Duncan Davidson (duncan@x180.com)
- Tom Dimock (tad1@cornell.edu)
- Peter Donald (donaldp@apache.org)
- dIon Gillard (dion@apache.org)
- Erik Hatcher (ehatcher@apache.org)
- Diane Holt (holtdl@yahoo.com)
- Bill Kelly (bill.kelly@softwired-inc.com)
- Arnout J. Kuiper (ajkuiper@wxs.nl)
- Antoine Lévy-Lambert
- Conor MacNeill
- Jan Matèrne
- Stefano Mazzocchi (stefano@apache.org)
- Erik Meade (emeade@geekfarm.org)
- Sam Ruby (rubys@us.ibm.com)
- Nico Seessle (nico@seessle.de)
- Jon S. Stevens (jon@latchkey.com)
- Wolf Siberski
- Magesh Umasankar
- Roger Vaughn (rvaughn@seaconinc.com)
- Dave Walend (dwalend@cs.tufts.edu)
- Phillip Wells (philwells@rocketmail.com)
- Christoph Wilhelms
- Craeg Strong (cstrong@arielpartners.com)

Version: 1.6.0

$Id: credits.html,v 1.23.2.4 2003/12/16 14:48:11 antoine Exp $

2 Feedback and Troubleshooting

If things do not work, especially simple things like ant -version, then something is wrong with your configuration. Before filing bug reports and emailing all the ant mailing lists

1. Check your environment variables. Are ANT_HOME and JAVA_HOME correct? If they have quotes or trailing slashes, remove them.
2. Unset CLASSPATH; if that is wrong things go horribly wrong. Ant does not need the CLASSPATH variable defined to anything to work.
3. Make sure there are no versions of crimson.jar or other XML parsers in JRE/ext
4. Is your path correct? is Ant on it? What about JDK/bin? have you tested this? If you are using Jikes, is it on the path? A createProcess error (especially with ID=2 on windows) usually means executable not found on the path.
5. Which version of ant are you running? Other applications distribute a copy -it may be being picked up by accident.
6. If a task is failing to run is optional.jar in ANT_HOME/lib? Are there any libraries which it depends on missing?
7. If a task doesn't do what you expect, run ant -verbose or ant -debug to see what is happening

If you can't fix your problem, start with the Ant User Mailing List . These are other ant users who will help you learn to use ant. If they cannot fix it then someone may suggest filing a bug report, which will escalate the issue. Remember of course, that support, like all open source development tasks, is voluntary. If you haven't invested time in helping yourself by following the steps above, it is unlikely that anyone will invest the time in helping you.

Also, if you don't understand something, the Ant User Mailing List is the place to ask questions. Not the developer list, nor the individuals whose names appears in the source and documentation. If they answered all such emails, nobody would have any time to improve ant.

To provide feedback on this software, please subscribe to the Ant User Mailing List

If you want to contribute to Ant or stay current with the latest development, join the Ant Development Mailing List

Archives of both lists can be found at http://archive.covalent.net/. Many thanks to Covalent Technologies. A searchable archive can be found at http://marc.theaimsgroup.com. If you know of any additional archive sites, please report them to the lists.

3 Introduction

This is the manual for version 1.6.0 of Apache Ant. If your version of Ant (as verified with `ant -version`) is older or newer than this version then this is not the correct manual set. Please use the documentation appropriate to your current version. Also, if you are using a version older than the most recent release, we recommend an upgrade to fix bugs as well as provide new functionality.

Apache Ant is a Java-based build tool. In theory, it is kind of like make, without make's wrinkles.

3.1 Why?

Why another build tool when there is already make, gnumake, nmake, jam, and others? Because all those tools have limitations that Ant's original author couldn't live with when developing software across multiple platforms. Make-like tools are inherently shell-based: they evaluate a set of dependencies, then execute commands not unlike what you would issue on a shell. This means that you can easily extend these tools by using or writing any program for the OS that you are working on; however, this also means that you limit yourself to the OS, or at least the OS type, such as Unix, that you are working on.

Makefiles are inherently evil as well. Anybody who has worked on them for any time has run into the dreaded tab problem. "Is my command not executing because I have a space in front of my tab?!!" said the original author of Ant way too many times. Tools like Jam took care of this to a great degree, but still have yet another format to use and remember.

Ant is different. Instead of a model where it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML-based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular Task interface.

Granted, this removes some of the expressive power that is inherent in being able to construct a shell command such as `` `find . -name foo -exec rm {}` ``, but it gives you the ability to be cross-platform - to work anywhere and everywhere. And hey, if you really need to execute a shell command, Ant has an <exec> task that allows different commands to be executed based on the OS it is executing on.

4 Installing Ant

4.1 Getting Ant

4.1.1 Binary Edition

The latest stable version of Ant is available from the Ant web page http://ant.apache.org/. If you like living on the edge, you can download the latest version from http://cvs.apache.org/builds/ant/nightly/.

4.1.2 Source Edition

If you prefer the source edition, you can download the source for the latest Ant release from http://ant.apache.org/srcdownload.cgi. Again, if you prefer the edge, you can access the code as it is being developed via CVS. The Jakarta website has details on accessing CVS. Please checkout the ant module. See the section Building Ant on how to build Ant from the source code. You can also access the Ant CVS repository on-line.

4.2 System Requirements

Ant has been used successfully on many platforms, including Linux, commercial flavours of Unix such as Solaris and HP-UX, Windows 9x and NT, OS/2 Warp, Novell Netware 6 and MacOS X.
To build and use Ant, you must have a JAXP-compliant XML parser installed and available on your classpath.

The binary distribution of Ant includes the latest version of the Apache Xerces2 XML parser. Please see http://java.sun.com/xml/ for more information about JAXP. If you wish to use a different JAXP-compliant parser, you should remove xercesImpl.jar and xml-apis.jar from Ant's lib directory. You can then either put the jars from your preferred parser into Ant's lib directory or put the jars on the system classpath.

For the current version of Ant, you will also need a JDK installed on your system, version 1.2 or later.
Note: The Microsoft JVM/JDK is not supported.
Note #2: If a JDK is not present, only the JRE runtime, then many tasks will not work.

4.3 Installing Ant

The binary distribution of Ant consists of the following directory layout:

```
ant
 +--- bin  // contains launcher scripts
 |
 +--- lib  // contains Ant jars plus necessary dependencies
 |
 +--- docs // contains documentation
 |      +--- ant2    // a brief description of ant2 requirements
 |      |
 |      +--- images  // various logos for html documentation
 |      |
 |      +--- manual  // Ant documentation (a must read ;-)
 |
 +--- etc // contains xsl goodies to:
          //   - create an enhanced report from xml output of various tasks.
          //   - migrate your build files and get rid of 'deprecated' warning
          //   - ... and more ;-)
```

Only the bin and lib directories are required to run Ant. To install Ant, choose a directory and copy the distribution file there. This directory will be known as ANT_HOME.

Windows 95, Windows 98 & Windows ME Note:

- On these systems, the script used to launch Ant will have problems if ANT_HOME is a long filename (i.e. a filename which is not of the format known as "8.3"). This is due to limitations in the OS's handling of the "for" batch-file statement. It is recommended, therefore, that Ant be installed in a short, 8.3 path, such as C:\Ant.
- On these systems you will also need to configure more environment space to cater for the environment variables used in the Ant lauch script. To do this, you will need to add or update the following line in the config.sys file

```
shell=c:\command.com c:\ /p /e:32768
```

4.3.1 Setup

Before you can run ant there is some additional set up you will need to do:

- Add the bin directory to your path.
- Set the ANT_HOME environment variable to the directory where you installed Ant. On some operating systems the ant wrapper scripts can guess ANT_HOME (Unix dialects and Windows NT/2000) - but it is better to not rely on this behavior.
- Optionally, set the JAVA_HOME environment variable (see the Advanced section below). This should be set to the directory where your JDK is installed.

Note: Do not install Ant's ant.jar file into the lib/ext directory of the JDK/JRE. Ant is an application, whilst the extension directory is intended for JDK extensions. In particular there are security restrictions on the classes which may be loaded by an extension.

4.3.2 Optional Tasks

Ant supports a number of optional tasks. An optional task is a task which typically requires an external library to function. The optional tasks are packaged together with the core Ant tasks.
The external libraries required by each of the optional tasks is detailed in the Library Dependencies section. These external libraries may either be placed in Ant's lib directory, where they will be picked up automatically, or made available on the system CLASSPATH environment variable.

4.3.3 Windows and OS/2

Assume Ant is installed in `c:\ant\.` The following sets up the environment:

```
set ANT_HOME=c:\ant
set JAVA_HOME=c:\jdk1.2.2
set PATH=%PATH%;%ANT_HOME%\bin
```

4.3.4 Unix (bash)

Assume Ant is installed in /usr/local/ant. The following sets up the environment:

```
export ANT_HOME=/usr/local/ant
export JAVA_HOME=/usr/local/jdk-1.2.2
export PATH=${PATH}:${ANT_HOME}/bin
```

4.3.5 Unix (csh)

```
setenv ANT_HOME /usr/local/ant
setenv JAVA_HOME /usr/local/jdk-1.2.2
set path=( $path $ANT_HOME/bin )
```

4.3.6 Advanced

There are lots of variants that can be used to run Ant. What you need is at least the following:

- The classpath for Ant must contain ant.jar and any jars/classes needed for your chosen JAXP-compliant XML parser.

- When you need JDK functionality (such as for the javac task or the rmic task), then for JDK 1.1, the classes.zip file of the JDK must be added to the classpath; for JDK 1.2 or JDK 1.3, tools.jar must be added. The scripts supplied with Ant, in the bin directory, will add the required JDK classes automatically, if the JAVA_HOME environment variable is set.
- When you are executing platform-specific applications, such as the exec task or the cvs task, the property ant.home must be set to the directory containing where you installed Ant. Again this is set by the Ant scripts to the value of the ANT_HOME environment variable.

The supplied ant shell scripts all support an ANT_OPTS environment variable which can be used to supply extra options to ant. Some of the scripts also read in an extra script stored in the users home directory, which can be used to set such options. Look at the source for your platform's invocation script for details.

4.4 Building Ant

To build Ant from source, you can either install the Ant source distribution or checkout the ant module from CVS.
Once you have installed the source, change into the installation directory.
Set the JAVA_HOME environment variable to the directory where the JDK is installed. See Installing Ant for examples on how to do this for your operating system.

Make sure you have downloaded any auxiliary jars required to build tasks you are interested in. These should either be available on the CLASSPATH or added to the lib directory. See Library Dependencies for a list of jar requirements for various features. Note that this will make the auxiliary jars available for the building of Ant only. For running Ant you will still need to make the jars available as described under Installing Ant.

Your are now ready to build Ant:

build -Ddist.dir=<directory_to_contain_Ant_distribution> dist (Windows)

build.sh -Ddist.dir=<directory_to_contain_Ant_distribution> dist (Unix)

This will create a binary distribution of Ant in the directory you specified.
The above action does the following:

- If necessary it will bootstrap the Ant code. Bootstrapping involves the manual compilation of enough Ant code to be able to run Ant. The bootstrapped Ant is used for the remainder of the build steps.
- Invokes the bootstrapped Ant with the parameters passed to the build script. In this case, these parameters define an Ant property value and specify the "dist" target in Ant's own build.xml file.

On most occasions you will not need to explicitly bootstrap Ant since the build scripts do that for you. If however, the build file you are using makes use of features not yet compiled into the bootstrapped Ant, you will need to manually bootstrap. Run bootstrap.bat (Windows) or bootstrap.sh (UNIX) to build a new bootstrap version of Ant.

If you wish to install the build into the current ANT_HOME directory, you can use:

```
build install    (Windows)
build.sh install    (Unix)
```

You can avoid the lengthy Javadoc step, if desired, with:

```
build install-lite    (Windows)
build.sh install-lite    (Unix)
```

This will only install the bin and lib directories.
Both the install and install-lite targets will overwrite the current Ant version in ANT_HOME.

4.5 Library Dependencies

The following libraries are needed in your CLASSPATH or in the install directory's lib directory if you are using the indicated feature. Note that only one of the regexp libraries is needed for use with the mappers. You will also need to install the Ant optional jar containing the task definitions to make these tasks available. Please refer to the Installing Ant / Optional Tasks section above.

Jar Name	Needed For	Available At
An XSL transformer like Xalan or XSL:P	style task	**If you use JDK 1.4, an XSL transformer is already included, so you need not do anything special.** • XALAN : http://xml.apache.org/xalan-j/index.html • XSL:P : used to live at http://www.clc-marketing.com/xslp/, but the link doesn't work any longer and we are not aware of a replacement site.
jakarta-regexp-1.3.jar	regexp type with mappers	http://jakarta.apache.org/regexp/
jakarta-oro-2.0.7.jar	regexp type with mappers and the perforce tasks	http://jakarta.apache.org/oro/
junit.jar	junit tasks	http://www.junit.org/
xalan.jar	junitreport task	http://xml.apache.org/xalan-j/
stylebook.jar	stylebook task	CVS repository of http://xml.apache.org/
testlet.jar	**deprecated** test task	Build from the gzip compress tar archive in http://avalon.apache.org/historiccvs/testlet/
antlr.jar	antlr task	http://www.antlr.org/
bsf.jar	script task **Note**: Ant 1.6 and later require Apache BSF, not the IBM version. I.e. you need BSF 2.3.0-rc1 or later.	http://jakarta.apache.org/bsf/
netrexx.jar	netrexx task, Rexx with the script task	http://www2.hursley.ibm.com/netrexx/
js.jar	Javascript with script task If you use Apache BSF 2.3.0-rc1, you must use rhino 1.5R3 - later versions of BSF work with 1.5R4 as well.	http://www.mozilla.org/rhino/
jython.jar	Python with script task	http://jython.sourceforge.net/
jpython.jar	Python with script task **deprecated, jython is the prefered engine**	http://www.jpython.org/
jacl.jar and tcljava.jar	TCL with script task	http://www.scriptics.com/software/java/
BeanShell JAR(s)	BeanShell with script task. **Note**: Ant 1.6 and later require BeanShell version 1.3 or later	http://www.beanshell.org/
jruby.jar	Ruby with script task	http://jruby.sourceforge.net/
judo.jar	Judoscript with script task	http://www.judoscript.com/index.html
commons-logging.jar	CommonsLoggingListener	http://jakarta.apache.org/commons/logging/index.html
log4j.jar	Log4jListener	http://jakarta.apache.org/log4j/docs/index.html
commons-net.jar	ftp, rexec and telnet tasks	http://jakarta.apache.org/commons/net/index.html

bcel.jar	classfileset data type, JavaClassHelper used by the ClassConstants filter reader and optionally used by ejbjar for dependency determination	http://jakarta.apache.org/bcel/
mail.jar	Mail task with Mime encoding, and the MimeMail task	http://java.sun.com/products/javamail/
jsse.jar	Support for SMTP over TLS/SSL in the Mail task Already included in jdk 1.4	http://java.sun.com/products/jsse/
activation.jar	Mail task with Mime encoding, and the MimeMail task	http://java.sun.com/products/javabeans/glasgow/jaf.html
jdepend.jar	jdepend task	http://www.clarkware.com/software/JDepend.html
resolver.jar **1.1beta or later**	xmlcatalog datatype only if support for external catalog files is desired	http://xml.apache.org/commons/.
jsch.jar	sshexec and scp tasks	http://www.jcraft.com/jsch/index.html
JAI - Java Advanded Imaging	image task	http://java.sun.com/products/java-media/jai/
IContract	icontract task Warning : the icontract jar file contains also antlr classes. To make the antlr task work properly, remove antlr/ANTLRGrammarParseBehavior.class from the icontract jar file installed under $ANT_HOME/lib.	http://www.reliable-systems.com/tools/

4.6 Platform Issues

4.6.1 Unix

- You should use a GNU version of tar to untar the ant source tree, if you have downloaded this as a tar file.
- Ant does not preserve file permissions when a file is copied, moved or archived, because Java does not let it read or write the permissions. Use <chmod> to set permissions, and when creating a tar archive, use the mode attribute of <tarfileset> to set the permissions in the tar file, or <apply> the real tar program.
- Ant is not symbolic link aware in moves, deletes and when recursing down a tree of directories to build up a list of files. Unexpected things can happen.

4.6.2 Microsoft Windows

Windows 9x (win95, win98, win98SE and winME) has a batch file system which does not work fully with long file names, so we recommend that ant and the JDK are installed into directories without spaces, and with 8.3 filenames. The Perl and Python launcher scripts do not suffer from this limitation.

All versions of windows are usually case insensitive, although mounted file systems (Unix drives, Clearcase views) can be case sensitive underneath, confusing patternsets.
Ant can often not delete a directory which is open in an Explorer window. There is nothing we can do about this short of spawning a program to kill the shell before deleting directories.

4.6.3 Cygwin

Cygwin is not really an operating system; rather it is an application suite running under Windows and providing some UNIX like functionality. AFAIK, Sun did not create any specific Java Development Kit or Java Runtime

Environment for cygwin. See this link : http://www.inonit.com/cygwin/faq/ . Only Windows path names are supported by JDK and JRE tools under Windows or cygwin. Relative path names such as "src/org/apache/tools" are supported, but Java tools do not understand /cygdrive/c to mean c:\.

The utility cygpath (used industrially in the ant script to support cygwin) can convert cygwin path names to Windows. You can use the task in ant to convert cygwin paths to Windows path, for instance like that :

```
<property name="some.cygwin.path" value="/cygdrive/h/somepath"/>
<exec executable="cygpath" outputproperty="windows.pathname">
   <arg value="--windows"/>
   <arg value="${some.cygwin.path}"/>
</exec>
<echo message="${windows.pathname}"/>
```

4.6.4 Apple MacOS X

MacOS X is the first of the Apple platforms that Ant supports completely; it is treated like any other Unix.

4.6.5 Novell Netware

To give the same level of sophisticated control as Ant's startup scripts on other platforms, it was decided to make the main ant startup on NetWare be via a Perl Script, "runant.pl". This is found in the bin directory (for instance - bootstrap\bin or dist\bin).

One important item of note is that you need to set up the following to run ant:

- CLASSPATH - put ant.jar, xercesImpl.jar, xml-apis.jar and any other needed jars on the system classpath.
- ANT_OPTS - On NetWare, ANT_OPTS needs to include a parameter of the form, "-envCWD=ANT_HOME", with ANT_HOME being the fully expanded location of Ant, not an environment variable. This is due to the fact that the NetWare System Console has no notion of a current working directory.

It is suggested that you create up an ant.ncf that sets up these parameters, and calls perl ANT_HOME/dist/bin/runant.pl

The following is an example of such an NCF file(assuming ant is installed in 'sys:/apache-ant/'):

```
   envset CLASSPATH=SYS:/apache-ant/bootstrap/lib/ant.jar
   envset CLASSPATH=$CLASSPATH;SYS:/apache-ant/lib/xercesImpl.jar
   envset CLASSPATH=$CLASSPATH;SYS:/apache-ant/lib/xml-apis.jar
   envset CLASSPATH=$CLASSPATH;SYS:/apache-ant/lib/optional/junit.jar
   envset CLASSPATH=$CLASSPATH;SYS:/apache-ant/bootstrap/lib/optional.jar

   setenv ANT_OPTS=-envCWD=sys:/apache-ant
   envset ANT_OPTS=-envCWD=sys:/apache-ant
   setenv ANT_HOME=sys:/apache-ant/dist/lib
   envset ANT_HOME=sys:/apache-ant/dist/lib

   perl sys:/apache-ant/dist/bin/runant.pl
```

Ant works on JVM version 1.3 or higher. You may have some luck running it on JVM 1.2, but serious problems have been found running Ant on JVM 1.1.7B. These problems are caused by JVM bugs that will not be fixed.

JVM 1.3 is supported on Novell NetWare versions 5.1 and higher.

4.6.6 Other platforms

Support for other platforms is not guaranteed to be complete, as certain techniques to hide platform details from build files need to be written and tested on every particular platform. Contributions in this area are welcome.

5 Running Ant

5.1 Command Line

If you've installed Ant as described in the Installing Ant section, running Ant from the command-line is simple: just type ant.

When no arguments are specified, Ant looks for a build.xml file in the current directory and, if found, uses that file as the build file and runs the target specified in the default attribute of the <project> tag. To make Ant use a build file other than build.xml, use the command-line option -buildfile file, where file is the name of the build file you want to use.

If you use the -find [file] option, Ant will search for a build file first in the current directory, then in the parent directory, and so on, until either a build file is found or the root of the filesystem has been reached. By default, it will look for a build file called build.xml. To have it search for a build file other than build.xml, specify a file argument. Note: If you include any other flags or arguments on the command line after the -find flag, you must include the file argument for the -find flag, even if the name of the build file you want to find is build.xml.

You can also set properties on the command line. This can be done with the -Dproperty=value option, where property is the name of the property, and value is the value for that property. If you specify a property that is also set in the build file (see the property task), the value specified on the command line will override the value specified in the build file. Defining properties on the command line can also be used to pass in the value of environment variables - just pass -DMYVAR=%MYVAR% (Windows) or -DMYVAR=$MYVAR (Unix) to Ant. You can then access these variables inside your build file as ${MYVAR}. You can also access environment variables using the property task's environment attribute.

Options that affect the amount of logging output by Ant are: -quiet, which instructs Ant to print less information to the console; -verbose, which causes Ant to print additional information to the console; and -debug, which causes Ant to print considerably more additional information.

It is also possible to specify one or more targets that should be executed. When omitted, the target that is specified in the default attribute of the project tag is used.

The -projecthelp option prints out a list of the build file's targets. Targets that include a description attribute are listed as "Main targets", those without a description are listed as "Subtargets", then the "Default" target is listed.

5.1.1 Command-line Options Summary

```
ant [options] [target [target2 [target3] ...]]
Options:
  -help, -h              print this message
  -projecthelp, -p       print project help information
  -version               print the version information and exit
  -diagnostics           print information that might be helpful to
                         diagnose or report problems.
  -quiet, -q             be extra quiet
  -verbose, -v           be extra verbose
  -debug, -d             print debugging information
  -emacs, -e             produce logging information without adornments
  -lib <path>            specifies a path to search for jars and classes
  -logfile <file>        use given file for log
    -l     <file>                '' 
  -logger <classname>    the class which is to perform logging
  -listener <classname>  add an instance of class as a project listener
```

```
-noinput               do not allow interactive input
-buildfile <file>      use given buildfile
  -file    <file>              ''
  -f       <file>              ''
-D<property>=<value>   use value for given property
-keep-going, -k        execute all targets that do not depend
                       on failed target(s)
-propertyfile <name>   load all properties from file with -D
                       properties taking precedence
-inputhandler <class>  the class which will handle input requests
-find <file>           (s)earch for buildfile towards the root of
  -s  <file>           the filesystem and use it
```

For more information about -logger and -listener see Loggers & Listeners.
For more information about -inputhandler see InputHandler.

5.1.2 Library Directories

Prior to Ant 1.6, all jars in the ANT_HOME/lib would be added to the CLASSPATH used to run Ant. This was done in the scripts that started Ant. From Ant 1.6, two directories are scanned by default and more can be added as required. The default directories scanned are ANT_HOME/lib and a user specific directory, ${user.home}/.ant/lib. This arrangement allows the Ant installation to be shared by many users while still allowing each user to deploy additional jars. Such additional jars could be support jars for Ant's optional tasks or jars containing third-party tasks to be used in the build. It also allows the main Ant installation to be locked down which will please system adminstrators.

Additional directories to be searched may be added by using the -lib option. The -lib option specifies a search path. Any jars or classes in the directories of the path will be added to Ant's classloader. The order in which jars are added to the classpath is as follows

- -lib jars in the order specified by the -lib elements on the command line
- jars from ${user.home}/.ant/lib
- jars from ANT_HOME/lib

Note that the CLASSPATH environment variable is passed to Ant using a -lib option. Ant itself is started with a very minimalistic classpath.

The location of ${user.home}/.ant/lib is somewhat dependent on the JVM. On Unix systems ${user.home} maps to the user's home directory whilst on recent versions of Windows it will be somewhere such as C:\Documents and Settings\username\.ant\lib. You should consult your JVM documentation for more details.

Examples

`ant`
runs Ant using the build.xml file in the current directory, on the default target.

`ant -buildfile test.xml`
runs Ant using the test.xml file in the current directory, on the default target.

`ant -buildfile test.xml dist`
runs Ant using the test.xml file in the current directory, on the target called dist.

`ant -buildfile test.xml -Dbuild=build/classes dist`
runs Ant using the test.xml file in the current directory, on the target called dist, setting the build property to the value build/classes.

```
ant -lib /home/ant/extras
```

runs Ant picking up additional task and support jars from the /home/ant/extras location

5.1.3 Files

The Ant wrapper script for Unix will source (read and evaluate) the file ~/.antrc before it does anything. On Windows, the Ant wrapper batch-file invokes %HOME%\antrc_pre.bat at the start and %HOME%\antrc_post.bat at the end. You can use these files, for example, to set/unset environment variables that should only be visible during the execution of Ant. See the next section for examples.

5.1.4 Environment Variables

The wrapper scripts use the following environment variables (if set):

- JAVACMD - full path of the Java executable. Use this to invoke a different JVM than JAVA_HOME/bin/java(.exe).
- ANT_OPTS - command-line arguments that should be passed to the JVM. For example, you can define system properties or set the maximum Java heap size here.
- ANT_ARGS - Ant command-line arguments. For example, set ANT_ARGS to point to a different logger, include a listener, and to include the -find flag. Note: If you include -find in ANT_ARGS, you should include the name of the build file to find, even if the file is called build.xml.

5.1.5 Java System Properties

Some of Ants core classes ant tasks can be configured via system properties.

So here the result of a search through the codebase. Because system properties are available via Project instance, I searched for them with a

```
grep -r -n "getPropert" * > ..\grep.txt
```

command. After that I filtered out the often-used but not-so-important values (most of them read-only values): *path.separator, ant.home, basedir, user.dir, os.name, ant.file, line.separator, java.home, java.version, java.version, user.home, java.class.path*

And I filtered out the getPropertyHelper access.

property name	**valid values /default value**	**description**
ant.input.properties	filename (required)	Name of the file holding the values for the PropertyFileInputHandler.
ant.logger.defaults	filename (optional, default '/org/ apache/ tools/ ant/ listener/ defaults.properties')	Name of the file holding the color mappings for the AnsiColorLogger.
ant.netrexxc.*	several formats	Use specified values as defaults for netrexxc.
ant.PropertyHelper	ant-reference-name (optional)	Specify the PropertyHelper to use. The object must be of the type org.apache.tools.ant.PropertyHelper. If not defined an object of org.apache.tools.ant.PropertyHelper will be used as PropertyHelper.
ant.regexp.regexpimpl	classname	classname for a RegExp implementation; if not set Ant tries to find another (Jdk14, Oro...); RegExp-Mapper "Choice of regular expression implementation"
ant.reuse.loader	boolean	allow to reuse classloaders used in

		org.apache.tools.ant.util.ClasspathUtil
ant.XmlLogger.stylesheet.uri	filename (default 'log.xsl')	Name for the stylesheet to include in the logfile by XmlLogger.
build.compiler	name	Specify the default compiler to use. see javac, EJB Tasks (compiler attribute), IContract, javah
build.compiler.emacs	boolean (default false)	Enable emacs-compatible error messages. see javac "Jikes Notes"
build.compiler.fulldepend	boolean (default false)	Enable full dependency checking see javac "Jikes Notes"
build.compiler.jvc.extensions	boolean (default true)	enable Microsoft extensions of their java compiler see javac "Jvc Notes"
build.compiler.pedantic	boolean (default false)	Enable pedantic warnings. see javac "Jikes Notes"
build.compiler.warnings	Deprecated flag	see javac "Jikes Notes"
build.rmic	name	control the rmic compiler
build.sysclasspath	"only", something else	only: current threads get the actual class loader (AntClassLoader.setThreadContextLoader()). else: use core loader as default (ComponentHelper.initTasks()). Disable changing the classloader (oata.taskdefs.Classloader.execute() experimental task).
file.encoding	name of a supported character set (e.g. UTF-8, ISO-8859-1, US-ASCII)	use as default character set of email messages; use as default for source-, dest- and bundleencoding in translate see JavaDoc of java.nio.charset.Charset for more information about character sets (not used in Ant, but has nice docs).
jikes.class.path	path	The specified path is added to the classpath if jikes is used as compiler.
MailLogger.properties.file, MailLogger.*	filename (optional, defaults derived from Project instance)	Name of the file holding properties for sending emails by the MailLogger. Override properties set inside the buildfile or via command line.
org.apache.tools.ant.ProjectHelper	classname (optional, default 'org. apache. tools. ant. ProjectHelper')	specifies the classname to use as ProjectHelper. The class must extend org.apache.tools.ant.ProjectHelper.
p4.port, p4.client, p4.user	several formats	Specify defaults for port-, client- and user-setting of the perforce tasks.
websphere.home	path	Points to home directory of websphere. see EJB Tasks
XmlLogger.file	filename (default 'log.xml')	Name for the logfile for MailLogger.

5.1.6 Cygwin Users

The Unix launch script that come with Ant works correctly with Cygwin. You should not have any problems launching Ant form the Cygwin shell. It is important to note however, that once Ant is runing it is part of the JDK which operates as a native Windows application. The JDK is not a Cygwin executable, and it therefore has no knowledge of the Cygwin paths, etc. In particular when using the <exec> task, executable names such as "/bin/sh" will not work, even though these work from the Cygwin shell from which Ant was launched. You can use an executable name such as "sh" and rely on that command being available in the Windows path.

5.1.7 OS/2 Users

The OS/2 lanuch script was developed so as it can perform complex task. It has two parts: ant.cmd which calls Ant and antenv.cmd which sets environment for Ant. Most often you will just call ant.cmd using the same command line options as described above. The behaviour can be modified by a number of ways explained below.

Script ant.cmd first verifies whether the Ant environment is set correctly. The requirements are:

1. Environment variable JAVA_HOME is set.
2. Environment variable ANT_HOME is set.
3. environment variable CLASSPATH is set and contains at least one element from JAVA_HOME and at least one element from ANT_HOME.

If any of these conditions is violated, script antenv.cmd is called. This script first invokes configuration scripts if there exist: the system-wide configuration antconf.cmd from the %ETC% directory and then the user comfiguration antrc.cmd from the %HOME% directory. At this moment both JAVA_HOME and ANT_HOME must be defined because antenv.cmd now adds classes.zip or tools.jar (depending on version of JVM) and everything from %ANT_HOME%\lib except ant-*.jar to CLASSPATH. Finally ant.cmd calls per-directory configuration antrc.cmd. All settings made by ant.cmd are local and are undone when the script ends. The settings made by antenv.cmd are persistent during the lifetime of the shell (of course unless called automaticaly from ant.cmd). It is thus possible to call antenv.cmd manually and modify some settings before calling ant.cmd.

Scripts envset.cmd and runrc.cmd perform auxilliary tasks. All scripts have some documentation inside.

5.2 Running Ant via Java

If you have installed Ant in the do-it-yourself way, Ant can be started with two entry points:

```
java -Dant.home=c:\ant org.apache.tools.ant.Main [options] [target]
java -Dant.home=c:\ant org.apache.tools.ant.launch.Launcher [options] [target]
```

The first method runs Ant's traditional entry point. The second method uses the Ant Launcher introduced in Ant 1.6. The former method does not support the -lib option and all required classes are loaded from the CLASSPATH. You must ensure that all required jars are available. At a minimum the CLASSPATH should include:

- ant.jar and ant-launcher.jar
- jars/classes for your XML parser
- the JDK's required jar/zip files

The latter method supports the -lib option and will load jars from the specified ANT_HOME. You should start the latter with the most minimal classpath possible, generally just the ant-launcher.jar.

6 Using Ant

6.1 Writing a Simple Buildfile

Ant's buildfiles are written in XML. Each buildfile contains one project and at least one (default) target. Targets contain task elements. Each task element of the buildfile can have an id attribute and can later be referred to by the value supplied to this. The value has to be unique. (For additional information, see the Tasks section below.)

6.1.1 Projects

A *project* has three attributes:

Attribute	Description	Required
name	the name of the project.	No
default	the default target to use when no target is supplied.	Yes.
basedir	the base directory from which all path calculations are done. This attribute might be overridden by setting the "basedir" property beforehand. When this is done, it must be omitted in the project tag. If neither the attribute nor the property have been set, the parent directory of the buildfile will be used.	No

Optionally, a description for the project can be provided as a top-level <description> element (see the description type).

Each project defines one or more targets. A target is a set of tasks you want to be executed. When starting Ant, you can select which target(s) you want to have executed. When no target is given, the project's default is used.

6.1.2 Targets

A target can depend on other targets. You might have a target for compiling, for example, and a target for creating a distributable. You can only build a distributable when you have compiled first, so the distribute target depends on the compile target. Ant resolves these dependencies.

It should be noted, however, that Ant's depends attribute only specifies the order in which targets should be executed - it does not affect whether the target that specifies the dependency(s) gets executed if the dependent target(s) did not (need to) run.

Ant tries to execute the targets in the depends attribute in the order they appear (from left to right). Keep in mind that it is possible that a target can get executed earlier when an earlier target depends on it:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

Suppose we want to execute target D. From its depends attribute, you might think that first target C, then B and then A is executed. Wrong! C depends on B, and B depends on A, so first A is executed, then B, then C, and finally D.

A target gets executed only once, even when more than one target depends on it (see the previous example).

A target also has the ability to perform its execution if (or unless) a property has been set. This allows, for example, better control on the building process depending on the state of the system (java version, OS,

command-line property defines, etc.). To make a target sense this property, you should add the if (or unless) attribute with the name of the property that the target should react to. Note: Ant will only check whether the property has been set, the value doesn't matter. A property set to the empty string is still an existing property. For example:

```
<target name="build-module-A" if="module-A-present"/>
<target name="build-own-fake-module-A" unless="module-A-present"/>
```

In the first example, if the module-A-present property is set (to any value), the target will be run. In the second example, if the module-A-present property is set (again, to any value), the target will not be run.

If no if and no unless attribute is present, the target will always be executed.

The optional description attribute can be used to provide a one-line description of this target, which is printed by the -projecthelp command-line option. Targets without such a description are deemed internal and will not be listed, unless either the -verbose or -debug option is used.

It is a good practice to place your tstamp tasks in a so-called initialization target, on which all other targets depend. Make sure that target is always the first one in the depends list of the other targets. In this manual, most initialization targets have the name "init".

If the depends attribute and the if/unless attribute are set, the depends attribute is executed first.

A target has the following attributes:

Attribute	Description	Required
name	the name of the target.	Yes
depends	a comma-separated list of names of targets on which this target depends.	No
if	the name of the property that must be set in order for this target to execute.	No
unless	the name of the property that must not be set in order for this target to execute.	No
description	a short description of this target's function.	No

A target name can be any alphanumeric string valid in the encoding of the XML file. The empty string "" is in this set, as is comma "," and space " ". Please avoid using these, as they will not be supported in future Ant versions because of all the confusion they cause. IDE support of unusual target names, or any target name containing spaces, varies with the IDE.

Targets beginning with a hyphen such as "-restart" are valid, and can be used to name targets that should not be called directly from the command line.

6.1.3 Tasks

A task is a piece of code that can be executed.

A task can have multiple attributes (or arguments, if you prefer). The value of an attribute might contain references to a property. These references will be resolved before the task is executed.

Tasks have a common structure:

```
<name attribute1="value1" attribute2="value2" ... />
```

where name is the name of the task, attributeN is the attribute name, and valueN is the value for this attribute.

There is a set of built-in tasks, along with a number of optional tasks, but it is also very easy to write your own.

All tasks share a task name attribute. The value of this attribute will be used in the logging messages generated by Ant.

Tasks can be assigned an id attribute:

```
<taskname id="taskID" ... />
```

where taskname is the name of the task, and taskID is a unique identifier for this task. You can refer to the corresponding task object in scripts or other tasks via this name. For example, in scripts you could do:

```
<script ... >
  task1.setFoo("bar");
</script>
```

to set the foo attribute of this particular task instance. In another task (written in Java), you can access the instance via project.getReference("task1").

Note1: If "task1" has not been run yet, then it has not been configured (ie., no attributes have been set), and if it is going to be configured later, anything you've done to the instance may be overwritten.

Note2: Future versions of Ant will most likely not be backward-compatible with this behaviour, since there will likely be no task instances at all, only proxies.

6.1.4 Properties

A project can have a set of properties. These might be set in the buildfile by the property task, or might be set outside Ant. A property has a name and a value; the name is case-sensitive. Properties may be used in the value of task attributes. This is done by placing the property name between "${" and "}" in the attribute value. For example, if there is a "builddir" property with the value "build", then this could be used in an attribute like this: ${builddir}/classes. This is resolved at run-time as build/classes.

6.1.5 Built-in Properties

Ant provides access to all system properties as if they had been defined using a <property> task. For example, ${os.name} expands to the name of the operating system.

For a list of system properties see the Javadoc of System.getProperties.

In addition, Ant has some built-in properties:

```
basedir             the absolute path of the project's basedir (as set
                    with the basedir attribute of <project>).
ant.file            the absolute path of the buildfile.
ant.version         the version of Ant
ant.project.name    the name of the project that is currently executing;
                    it is set in the name attribute of <project>.
ant.java.version    the JVM version Ant detected; currently it can hold
                    the values "1.1", "1.2", "1.3" and "1.4".
```

6.1.6 Example Buildfile

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
```

```
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Notice that we are declaring properties outside any target. As of Ant 1.6 all tasks can be declared outside targets (earlier version only allowed <property>,<typedef> and <taskdef>). When you do this they are evaluated before any targets are executed. Some tasks will generate build failures if they are used outside of targets as they may cause infinite loops otherwise (<antcall> for example).

We have given some targets descriptions; this causes the projecthelp invocation option to list them as public targets with the descriptions; the other target is internal and not listed.

Finally, for this target to work the source in the src subdirectory should be stored in a directory tree which matches the package names. Check the <javac> task for details.

6.1.7 Token Filters

A project can have a set of tokens that might be automatically expanded if found when a file is copied, when the filtering-copy behavior is selected in the tasks that support this. These might be set in the buildfile by the filter task.

Since this can potentially be a very harmful behavior, the tokens in the files must be of the form @token@, where token is the token name that is set in the <filter> task. This token syntax matches the syntax of other build systems that perform such filtering and remains sufficiently orthogonal to most programming and scripting languages, as well as with documentation systems.

Note: If a token with the format @token@ is found in a file, but no filter is associated with that token, no changes take place; therefore, no escaping method is available - but as long as you choose appropriate names for your tokens, this should not cause problems.

Warning: If you copy binary files with filtering turned on, you can corrupt the files. This feature should be used with text files only.

6.1.8 Path-like Structures

You can specify PATH- and CLASSPATH-type references using both ":" and ";" as separator characters. Ant will convert the separator to the correct character of the current operating system.

Wherever path-like values need to be specified, a nested element can be used. This takes the general form of:

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/helper.jar"/>
</classpath>
```

The location attribute specifies a single file or directory relative to the project's base directory (or an absolute filename), while the path attribute accepts colon- or semicolon-separated lists of locations. The path attribute is intended to be used with predefined paths - in any other case, multiple elements with location attributes should be preferred.
As a shortcut, the <classpath> tag supports path and location attributes of its own, so:

```
<classpath>
  <pathelement path="${classpath}"/>
</classpath>
```

can be abbreviated to:

```
<classpath path="${classpath}"/>
```

In addition, DirSets, FileSets, and FileLists can be specified via nested <dirset>, <fileset>, and <filelist> elements, respectively. Note: The order in which the files building up a FileSet are added to the path-like structure is not defined.

```
<classpath>
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
  <dirset dir="${build.dir}">
    <include name="apps/**/classes"/>
    <exclude name="apps/**/*Test*"/>
  </dirset>
  <filelist refid="third-party_jars"/>
</classpath>
```

This builds a path that holds the value of ${classpath}, followed by all jar files in the lib directory, the classes directory, all directories named classes under the apps subdirectory of ${build.dir}, except those that have the text Test in their name, and the files specified in the referenced FileList.

If you want to use the same path-like structure for several tasks, you can define them with a <path> element at the same level as targets, and reference them via their id attribute - see References for an example.

A path-like structure can include a reference to another path-like structure via nested <path> elements:

```
<path id="base.path">
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
</path>

<path id="tests.path">
  <path refid="base.path"/>
  <pathelement location="testclasses"/>
</path>
```

The shortcuts previously mentioned for <classpath> are also valid for <path>.For example:

```
<path id="base.path">
  <pathelement path="${classpath}"/>
</path>
```

can be written as:

```
<path id="base.path" path="${classpath}"/>
```

6.1.9 Command-line Arguments

Several tasks take arguments that will be passed to another process on the command line. To make it easier to specify arguments that contain space characters, nested arg elements can be used.

Attribute	Description	Required
value	a single command-line argument; can contain space characters.	Exactly one of these.
file	The name of a file as a single command-line argument; will be replaced with the absolute filename of the file.	
path	A string that will be treated as a path-like string as a single command-line argument; you can use ; or : as path separators and Ant will convert it to the platform's local conventions.	
pathref	Reference to a path defined elsewhere. Ant will convert it to the platform's local conventions.	
line	a space-delimited list of command-line arguments.	

It is highly recommended to avoid the line version when possible. Ant will try to split the command line in a way similar to what a (Unix) shell would do, but may create something that is very different from what you expect under some circumstances.

Examples

```
<arg value="-l -a"/>
```

is a single command-line argument containing a space character.

```
<arg line="-l -a"/>
```

represents two separate command-line arguments.

```
<arg path="/dir;/dir2:\dir3"/>
```

is a single command-line argument with the value \dir;\dir2;\dir3 on DOS-based systems and /dir:/dir2:/dir3 on Unix-like systems.

6.1.10 References

The id attribute of the buildfile's elements can be used to refer to them. This can be useful if you are going to replicate the same snippet of XML over and over again - using a <classpath> structure more than once, for example.

The following example:

```
<project ... >
  <target ... >
    <rmic ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </javac>
  </target>
</project>
```

could be rewritten as:

```
<project ... >
  <path id="project.class.path">
    <pathelement location="lib/"/>
    <pathelement path="${java.class.path}/"/>
    <pathelement path="${additional.path}"/>
  </path>

  <target ... >
    <rmic ...>
      <classpath refid="project.class.path"/>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath refid="project.class.path"/>
    </javac>
  </target>
</project>
```

All tasks that use nested elements for PatternSets, FileSets, ZipFileSets or path-like structures accept references to these structures as well.

7 Concepts

7.1 build.sysclasspath

The value of the build.sysclasspath property control how the system classpath, ie. the classpath in effect when Ant is run, affects the behaviour of classpaths in Ant. The default behavior varies from Ant to Ant task.

The values and their meanings are:

only	Only the system classpath is used and classpaths specified in build files, etc are ignored. This situation could be considered as the person running the build file knows more about the environment than the person writing the build file
ignore	The system classpath is ignored. This situation is the reverse of the above. The person running the build trusts the build file writer to get the build file right
last	The classpath is concatenated to any specified classpaths at the end. This is a compromise, where the build file writer has priority.
first	Any specified classpaths are concatenated to the system classpath. This is the other form of compromise where the build runner has priority.

7.2 Common Attributes of all Tasks

All tasks share the following attributes:

Attribute	Description	Required
id	Unique identifier for this task instance, can be used to reference this task in scripts.	No
taskname	A different name for this task instance - will show up in the logging output.	No
description	Room for your comments	No

7.3 Core Types

7.3.1 Assertions

The assertion type enables or disables the Java1.4 assertion feature, on a whole java program, or components of a program. It can be used in <java> and <junit> to add extra validation to code.

Assertions are covered in the Java 1.4.2 documentation, and the Java Language Specification
The key points to note are that a java.lang.AssertionError error is thrown when an assertion fails, and that the facility is only available on Java1.4 and later. To enable assertions one must set source="1.4", "1.5" or later in <javac> when the source is being compiled, and that the code must contain assert statements to be tested. The result of such an action is code that neither compiles or runs on earlier versions of Java. For this reason Ant itself currently contains no assertions.

When assertions are enabled (or disabled) in a task through nested assertions elements, the classloader or command line is modified with the appopriate options. This means that the JVM executed must be a Java1.4 or later JVM, even if there are no assertions in the code. Attempting to enable assertions on earlier VMs will result in an "Unrecognized option" error and the JVM will not start.

Attributes

Attribute	Description	Required
enableSystemAssertions	Flag to turn system assertions on or off.	No, default is 'unspecified'

When the System assertions have neither been enabled or disabled, then the JVM is not given any assertion information - the default action of the current JVMs is to disable system assertions.

Note also that there is no apparent documentation for what parts of the system have built in assertions.

Nested elements

enable

Enable assertions in portions of code.

Attribute	Description	Required
class	The name of a class to enable assertions on.	No
package	The name of a package to turn assertions on. Use "..." for the anonymous package.	No

disable

Disable assertions in portions of code.

Attribute	Description	Required
class	The name of a class to disable assertions for.	No
package	The name of a package to turn assertions off on. Use "..." for the anonymous package.	No

Because assertions are disabled by default, it only makes sense to disable assertions where they have been enabled in a parent package.

Examples

Example: enable a single class

Enable assertions in a class called Test

```
<assertions >
  <enable class="Test" />
</assertions>
```

Example: enable a package

Enable assertions in a all packages below org.apache

```
<assertions >
  <enable package="org.apache" />
</assertions>
```

Example: System assertions

Example: set system assertions and all org.apache packages except for ant, and the class org.apache.tools.ant.Main.

```
    <java fork="true" failonerror="true"
      classname="${classname}"
      classpathref="assert.classpath">
      <assertions enableSystemAssertions="true" >
        <enable package="org.apache" />
        <disable package="org.apache.ant" />
        <enable class="org.apache.tools.ant.Main"/>
      </assertions>
    </java>
```

Example: disabled and anonymous package assertions

Disable system assertions; enable those in the anonymous package

```
<assertions enableSystemAssertions="false" >
  <enable package="..." />
</assertions>
```

Example: referenced assertions

This type is a datatype, so you can declare assertions and use them later

```
<assertions id="project.assertions" >
  <enable package="org.apache.test" />
</assertions>

<java fork="true" failonerror="true"
  classname="${classname}"
  classpathref="assert.classpath">
  <assertions refid="project.assertions"/>
</java>
```

7.3.2 Description

Allows for a description of the project to be specified that will be included in the output of the ant -projecthelp command.

Parameters

(none)

Examples

```
<description>
This buildfile is used to build the Foo subproject within
the large, complex Bar project.
</description>
```

7.3.3 Directory-based Tasks

Some tasks use directory trees for the actions they perform. For example, the javac task, which compiles a directory tree with .java files into .class files, is one of these directory-based tasks. Because some of these tasks do so much work with a directory tree, the task itself can act as an implicit FileSet.

Whether the fileset is implicit or not, it can often be very useful to work on a subset of the directory tree. This section describes how you can select a subset of such a directory tree when using one of these directory-based tasks.

Ant gives you two ways to create a subset of files in a fileset, both of which can be used at the same time:

- Only include files and directories that match any include patterns and do not match any exclude patterns in a given PatternSet.
- Select files based on selection criteria defined by a collection of selector nested elements.

7.3.3.1 Patternset

We said that Directory-based tasks can sometimes act as an implicit <fileset>, but in addition to that, a FileSet acts as an implicit <patternset>.
The inclusion and exclusion elements of the implicit PatternSet can be specified inside the directory-based task (or explicit fileset) via either:

- the attributes includes and excludes.
- nested elements <include> and <exclude>.
- external files specified with the attributes includesfile and excludesfile.
- external files specified with the nested elements <includesfile> and <excludesfile>.

When dealing with an external file, each line of the file is taken as a pattern that is added to the list of include or exclude patterns.
When both inclusion and exclusion are used, only files/directories that match at least one of the include patterns and don't match any of the exclude patterns are used. If no include pattern is given, all files are assumed to match the include pattern (with the possible exception of the default excludes).

Patterns

As described earlier, patterns are used for the inclusion and exclusion of files. These patterns look very much like the patterns used in DOS and UNIX:

'*' matches zero or more characters, '?' matches one character.

Examples:

*.java matches .java, x.java and FooBar.java, but not FooBar.xml (does not end with .java).

?.java matches x.java, A.java, but not .java or xyz.java (both don't have one character before .java).

Combinations of *'s and ?'s are allowed.

Matching is done per-directory. This means that first the first directory in the pattern is matched against the first directory in the path to match. Then the second directory is matched, and so on. For example, when we have the pattern /?abc/*/*.java and the path /xabc/foobar/test.java, the first ?abc is matched with xabc, then * is matched with foobar, and finally *.java is matched with test.java. They all match, so the path matches the pattern.

To make things a bit more flexible, we add one extra feature, which makes it possible to match multiple directory levels. This can be used to match a complete directory tree, or a file anywhere in the directory tree. To do this, ** must be used as the name of a directory. When ** is used as the name of a directory in the pattern, it matches zero or more directories. For example: /test/** matches all files/directories under /test/, such as /test/x.java, or /test/foo/bar/xyz.html, but not /xyz.xml.

There is one "shorthand" - if a pattern ends with / or \, then ** is appended. For example, mypackage/test/ is interpreted as if it were mypackage/test/**.

Example patterns:

**/CVS/*	Matches all files in CVS directories that can be located anywhere in the directory tree. Matches: CVS/Repository org/apache/CVS/Entries org/apache/jakarta/tools/ant/CVS/Entries But not: org/apache/CVS/foo/bar/Entries (foo/bar/ part does not match)
org/apache/jakarta/**	Matches all files in the org/apache/jakarta directory tree. Matches: org/apache/jakarta/tools/ant/docs/index.html org/apache/jakarta/test.xml But not: org/apache/xyz.java (jakarta/ part is missing).
org/apache/**/CVS/*	Matches all files in CVS directories that are located anywhere in the directory tree under org/apache. Matches: org/apache/CVS/Entries org/apache/jakarta/tools/ant/CVS/Entries But not: org/apache/CVS/foo/bar/Entries

	(foo/bar/ part does not match)
/test/	Matches all files that have a test element in their path, including test as a filename.

When these patterns are used in inclusion and exclusion, you have a powerful way to select just the files you want.

7.3.3.2 Selectors

The <fileset>, whether implicit or explicit in the directory-based task, also acts as an <and> selector container. This can be used to create arbitrarily complicated selection criteria for the files the task should work with. See the Selector documentation for more information.

7.3.3.3 Standard Tasks/Filesets

Many of the standard tasks in ant take one or more filesets which follow the rules given here. This list, a subset of those, is a list of standard ant tasks that can act as an implicit fileset:

- <checksum>
- <copydir> (deprecated)
- <delete>
- <dependset>
- <fixcrlf>
- <javac>
- <replace>
- <rmic>
- <style> (aka <xslt>)
- <tar>
- <zip>
- <ddcreator>
- <ejbjar>
- <ejbc>
- <cab>
- <icontract>
- <native2ascii>
- <netrexxc>
- <renameextensions>
- <depend>
- <ilasm>
- <csc>
- <vbc>
- <translate>
- <vajexport>
- <image>
- <jlink> (deprecated)
- <jspc>
- <wljspc>

Examples

```
<copy todir="${dist}">
  <fileset dir="${src}"
           includes="**/images/*"
           excludes="**/*.gif"
  />
</copy>
```

This copies all files in directories called images that are located in the directory tree defined by ${src} to the destination directory defined by ${dist}, but excludes all *.gif files from the copy.

```
<copy todir="${dist}">
  <fileset dir="${src}">
    <include name="**/images/*"/>
    <exclude name="**/*.gif"/>
  </fileset>
</copy>
```

The same as the example above, but expressed using nested elements.

```
<delete dir="${dist}">
    <include name="**/images/*"/>
    <exclude name="**/*.gif"/>
</delete>
```

Deleting the original set of files, the delete task can act as an implicit fileset.

7.3.3.4 Default Excludes

There are a set of definitions that are excluded by default from all directory-based tasks. They are:

```
**/*~
**/#*#
**/.#*
**/%*%
**/._*
**/CVS
**/CVS/**
**/.cvsignore
**/SCCS
**/SCCS/**
**/vssver.scc
**/.svn
**/.svn/**
**/.DS_Store
```

If you do not want these default excludes applied, you may disable them with the defaultexcludes="no" attribute.

This is the default list, note that you can modify the list of default excludes by using the defaultexcludes task.

7.3.4 DirSet

DirSets are groups of directories. These directories can be found in a directory tree starting in a base directory and are matched by patterns taken from a number of PatternSets. DirSets can appear inside tasks that support this feature or at the same level as target (i.e., as children of <project>).

PatternSets can be specified as nested <patternset> elements. In addition, DirSet holds an implicit PatternSet and supports the nested <include>, <includesfile>, <exclude> and <excludesfile> elements of <patternset> directly, as well as <patternset>'s attributes.

Attribute	Description	Required
dir	The root of the directory tree of this DirSet.	Yes
includes	A comma- or space-separated list of patterns of directories that must be included; all directories are included when omitted.	No
includesfile	The name of a file; each line of this file is taken to be an include pattern.	No

excludes	A comma- or space-separated list of patterns of directories that must be excluded; no directories are excluded when omitted.	No
excludesfile	The name of a file; each line of this file is taken to be an exclude pattern.	No
casesensitive	Specifies whether case-sensitivty should be applied (true|yes|on or false|no|off).	No; defaults to true.
followsymlinks	Shall symbolic links be followed? Defaults to true. See fileset's documentation.	No

Examples

```
<dirset dir="${build.dir}">
  <include name="apps/**/classes"/>
  <exclude name="apps/**/*Test*"/>
</dirset>
```

Groups all directories named classes found under the apps subdirectory of ${build.dir}, except those that have the text Test in their name.

```
<dirset dir="${build.dir}">
  <patternset id="non.test.classes">
    <include name="apps/**/classes"/>
    <exclude name="apps/**/*Test*"/>
  </patternset>
</dirset>
```

Groups the same directories as the above example, but also establishes a PatternSet that can be referenced in other <dirset> elements, rooted at a different directory.

```
<dirset dir="${debug_build.dir}">
  <patternset refid="non.test.classes"/>
</dirset>
```

Groups all directories in directory ${debug_build.dir}, using the same patterns as the above example.

7.3.5 FileList

FileLists are explicitly named lists of files. Whereas FileSets act as filters, returning only those files that exist in the file system and match specified patterns, FileLists are useful for specifying files that may or may not exist. Multiple files are specified as a list of files, relative to the specified directory, with no support for wildcard expansion (filenames with wildcards will be included in the list unchanged). FileLists can appear inside tasks that support this feature or at the same level as <target> (i.e., as children of <project>).

Attribute	Description	Required
dir	The base directory of this FileList.	Yes
files	The list of file names.	Yes

Examples

```
<filelist
    id="docfiles"
    dir="${doc.src}"
    files="foo.xml,bar.xml"/>
```

The files ${doc.src}/foo.xml and ${doc.src}/bar.xml. Note that these files may not (yet) actually exist.

```
<filelist
    id="docfiles"
    dir="${doc.src}"
    files="foo.xml
```

```
            bar.xml"/>
```

Same files as the example above.

```
<filelist refid="docfiles"/>
```

Same files as the example above.

7.3.6 FileSet

FileSets are groups of files. These files can be found in a directory tree starting in a base directory and are matched by patterns taken from a number of PatternSets and Selectors. FileSets can appear inside tasks that support this feature or at the same level as target - i.e., as children of project.

PatternSets can be specified as nested <patternset> elements. In addition, FileSet holds an implicit PatternSet and supports the nested <include>, <includesfile>, <exclude> and <excludesfile> elements of PatternSet directly, as well as PatternSet's attributes.

Selectors are available as nested elements.within the FileSet. If any of the selectors within the FileSet do not select the file, the file is not considered part of the FileSet. This makes FileSets equivalent to an <and> selector container.

Attribute	Description	Required
dir	the root of the directory tree of this FileSet.	Either dir or file must be specified
file	shortcut for specifying a single file fileset	
defaultexcludes	indicates whether default excludes should be used or not (yes | no); default excludes are used when omitted.	No
includes	comma- or space-separated list of patterns of files that must be included; all files are included when omitted.	No
includesfile	the name of a file; each line of this file is taken to be an include pattern.	No
excludes	comma- or space-separated list of patterns of files that must be excluded; no files (except default excludes) are excluded when omitted.	No
excludesfile	the name of a file; each line of this file is taken to be an exclude pattern.	No
casesensitive	Must the include and exclude patterns be treated in a case sensitive way? Defaults to true.	No
followsymlinks	Shall symbolic links be followed? Defaults to true. See the note below.	No

Note: All files/directories for which the canonical path is different from its path are considered symbolic links. On Unix systems this usually means the file really is a symbolic links but it may lead to false results on other platforms.

Examples

```
<fileset dir="${server.src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</fileset>
```

Groups all files in directory ${server.src} that are Java source files and don't have the text Test in their name.

```
<fileset dir="${server.src}" casesensitive="yes">
  <patternset id="non.test.sources">
    <include name="**/*.java"/>
    <exclude name="**/*Test*"/>
  </patternset>
```

```
</fileset>
```
Groups the same files as the above example, but also establishes a PatternSet that can be referenced in other <fileset> elements, rooted at a different directory.

```
<fileset dir="${client.src}" >
  <patternset refid="non.test.sources"/>
</fileset>
```
Groups all files in directory ${client.src}, using the same patterns as the above example.

```
<fileset dir="${server.src}" casesensitive="yes">
  <filename name="**/*.java"/>
  <filename name="**/*Test*" negate="true"/>
</fileset>
```
Groups the same files as the top example, but using the <filename> selector.

```
<fileset dir="${server.src}" casesensitive="yes">
  <filename name="**/*.java"/>
  <not>
    <filename name="**/*Test*"/>
  </not>
</fileset>
```
Groups the same files as the previous example using a combination of the <filename> selector and the <not> selector container.

7.3.7 Mapping File Names

Some tasks take source files and create target files. Depending on the task, it may be quite obvious which name a target file will have (using javac, you know there will be .class files for your .java files) - in other cases you may want to specify the target files, either to help Ant or to get an extra bit of functionality.

While source files are usually specified as filesets, you don't specify target files directly - instead, you tell Ant how to find the target file(s) for one source file. An instance of org.apache.tools.ant.util.FileNameMapper is responsible for this. It constructs target file names based on rules that can be parameterized with from and to attributes - the exact meaning of which is implementation-dependent.

These instances are defined in <mapper> elements with the following attributes:

Attribute	Description	Required
type	specifies one of the built-in implementations.	Exactly one of both
classname	specifies the implementation by class name.	
classpath	the classpath to use when looking up classname.	No
classpathref	the classpath to use, given as reference to a path defined elsewhere.	No
from	the from attribute for the given implementation.	Depends on implementation.
to	the to attribute for the given implementation.	Depends on implementation.

Note that Ant will not automatically convert / or \ characters in the to and from attributes to the correct directory separator of your current platform. If you need to specify this separator, use ${file.separator} instead.

Parameters specified as nested elements
The classpath can be specified via a nested <classpath>, as well - that is, a path-like structure.

The built-in mapper types are:
All built-in mappers are case-sensitive.

identity

The target file name is identical to the source file name. Both to and from will be ignored.

Examples:

```
<mapper type="identity"/>
```

Source file name	Target file name
A.java	A.java
foo/bar/B.java	foo/bar/B.java
C.properties	C.properties
Classes/dir/dir2/A.properties	Classes/dir/dir2/A.properties

flatten

The target file name is identical to the source file name, with all leading directory information stripped off. Both to and from will be ignored.

Examples:

```
<mapper type="flatten"/>
```

Source file name	Target file name
A.java	A.java
foo/bar/B.java	B.java
C.properties	C.properties
Classes/dir/dir2/A.properties	A.properties

merge

The target file name will always be the same, as defined by to - from will be ignored.

Examples:

```
<mapper type="merge" to="archive.tar"/>
```

Source file name	Target file name
A.java	archive.tar
foo/bar/B.java	archive.tar
C.properties	archive.tar
Classes/dir/dir2/A.properties	archive.tar

glob

Both to and from define patterns that may contain at most one *. For each source file that matches the from pattern, a target file name will be constructed from the to pattern by substituting the * in the to pattern with the text that matches the * in the from pattern. Source file names that don't match the from pattern will be ignored.

Examples:

```
<mapper type="glob" from="*.java" to="*.java.bak"/>
```

Source file name	Target file name
A.java	A.java.bak
foo/bar/B.java	foo/bar/B.java.bak
C.properties	ignored
Classes/dir/dir2/A.properties	ignored

```
<mapper type="glob" from="C*ies" to="Q*y"/>
```

Source file name	Target file name
A.java	ignored

foo/bar/B.java	ignored
C.properties	Q.property
Classes/dir/dir2/A.properties	Qlasses/dir/dir2/A.property

regexp

Both to and from define regular expressions. If the source file name matches the from pattern, the target file name will be constructed from the to pattern, using \0 to \9 as back-references for the full match (\0) or the matches of the subexpressions in parentheses. Source files not matching the from pattern will be ignored.

Note that you need to escape a dollar-sign ($) with another dollar-sign in Ant.

The regexp mapper needs a supporting library and an implementation of org.apache.tools.ant.util.regexp.RegexpMatcher that hides the specifics of the library. Ant comes with implementations for the java.util.regex package of JDK 1.4, jakarta-regexp and jakarta-ORO. If you compile from sources and plan to use one of them, make sure the libraries are in your CLASSPATH. For information about using gnu.regexp or gnu.rex with Ant, see this article.

This means, you need optional.jar from the Ant release you are using and one of the supported regular expression libraries. Make sure, both will be loaded from the same classpath, that is either put them into your CLASSPATH, ANT_HOME/lib directory or a nested <classpath> element of the mapper - you cannot have optional.jar in ANT_HOME/lib and the library in a nested <classpath>.

Ant will choose the regular-expression library based on the following algorithm:

- If the system property ant.regexp.matcherimpl has been set, it is taken as the name of the class implementing org.apache.tools.ant.util.regexp.RegexpMatcher that should be used.
- If it has not been set, first try the JDK 1.4 classes, then jakarta-ORO and finally try jakarta-regexp.

Examples:

```
<mapper type="regexp" from="^(.*)\.java$$" to="\1.java.bak"/>
```

Source file name	Target file name
A.java	A.java.bak
foo/bar/B.java	foo/bar/B.java.bak
C.properties	ignored
Classes/dir/dir2/A.properties	ignored

```
<mapper type="regexp" from="^(.*)/([^/]+)/([^/]*)$$" to="\1/\2/\2-\3"/>
```

Source file name	Target file name
A.java	ignored
foo/bar/B.java	foo/bar/bar-B.java
C.properties	ignored
Classes/dir/dir2/A.properties	Classes/dir/dir2/dir2-A.properties

```
<mapper type="regexp" from="^(.*)\.(.*)$$" to="\2.\1"/>
```

Source file name	Target file name
A.java	java.A
foo/bar/B.java	java.foo/bar/B
C.properties	properties.C
Classes/dir/dir2/A.properties	properties.Classes/dir/dir2/A

package

Sharing the same syntax as the glob mapper, the package mapper replaces directory separators found in the matched source pattern with dots in the target pattern placeholder. This mapper is particularly useful in combination with <uptodate> and <junit> output.

Example:

```
<mapper type="package"
        from="*Test.java" to="TEST-*Test.xml"/>
```

Source file name	Target file name
org/apache/tools/ant/util/PackageMapperTest.java	TEST-org.apache.tools.ant.util.PackageMapperTest.xml
org/apache/tools/ant/util/Helper.java	ignored

unpackage (since ant 1.6)

This mapper is the inverse of the package mapper. It replaces the dots in a package name with directory separators. This is useful for matching XML formatter results against their JUnit test test cases. The mapper shares the sample syntax as the glob mapper.

Example:

```
<mapper type="unpackage"
        from="TEST-*Test.xml" to="${test.src.dir}/*Test.java">
```

Source file name	Target file name
TEST-org.acme.AcmeTest.xml	${test.src.dir}/org/acme/AcmeTest.java

7.3.8 FilterChains and FilterReaders

Look at Unix pipes - they offer you so much flexibility - say you wanted to copy just those lines that contained the string blee from the first 10 lines of a file 'foo' to a file 'bar' - you would do something like

```
cat foo|head -n10|grep blee > bar
```

Ant was not flexible enough. There was no way for the <copy> task to do something similar. If you wanted the <copy> task to get the first 10 lines, you would have had to create special attributes:

```
<copy file="foo" tofile="bar" head="10" contains="blee"/>
```

The obvious problem thus surfaced: Ant tasks would not be able to accomodate such data transformation attributes as they would be endless. The task would also not know in which order these attributes were to be interpreted. That is, must the task execute the contains attribute first and then the head attribute or vice-versa? What Ant tasks needed was a mechanism to allow pluggable filter (data tranformer) chains. Ant would provide a few filters for which there have been repeated requests. Users with special filtering needs would be able to easily write their own and plug them in.

The solution was to refactor data transformation oriented tasks to support FilterChains. A FilterChain is a group of ordered FilterReaders. Users can define their own FilterReaders by just extending the java.io.FilterReader class. Such custom FilterReaders can be easily plugged in as nested elements of <filterchain> by using <filterreader> elements.

Example:

```
<copy file="${src.file}" tofile="${dest.file}">
  <filterchain>
    <filterreader classname="your.extension.of.java.io.FilterReader">
      <param name="foo" value="bar"/>
    </filterreader>
```

```
    <filterreader classname="another.extension.of.java.io.FilterReader">
      <classpath>
        <pathelement path="${classpath}"/>
      </classpath>
      <param name="blah" value="blee"/>
      <param type="abra" value="cadabra"/>
    </filterreader>
  </filterchain>
</copy>
```

Ant provides some built-in filter readers. These filter readers can also be declared using a syntax similar to the above syntax. However, they can be declared using some simpler syntax also.

Example:

```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <headfilter lines="15"/>
  </filterchain>
</loadfile>
```

is equivalent to:

```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.HeadFilter">
      <param name="lines" value="15"/>
    </filterreader>
  </filterchain>
</loadfile>
```

The following built-in tasks support nested <filterchain> elements.
Concat,
Copy,
LoadFile,
LoadProperties,
Move

A FilterChain is formed by defining zero or more of the following nested elements.
FilterReader
ClassConstants
EscapeUnicode
ExpandProperties
HeadFilter
LineContains
LineContainsRegExp
PrefixLines
ReplaceTokens
StripJavaComments
StripLineBreaks
StripLineComments
TabsToSpaces
TailFilter
DeleteCharacters
ConcatFilter
TokenFilter

7.3.8.1 FilterReader

The filterreader element is the generic way to define a filter. User defined filter elements are defined in the build file using this. Please note that built in filter readers can also be defined using this syntax. A FilterReader element must be supplied with a class name as an attribute value. The class resolved by this name must extend java.io.FilterReader. If the custom filter reader needs to be parameterized, it must implement org.apache.tools.type.Parameterizable.

Attribute	Description	Required
classname	The class name of the filter reader.	Yes

7.3.8.2 Nested Elements:

<filterreader> supports <classpath> and <param> as nested elements. Each <param> element may take in the following attributes - name, type and value.
The following FilterReaders are supplied with the default distribution.

7.3.8.3 ClassConstants

This filters basic constants defined in a Java Class, and outputs them in lines composed of the format name=value

Example:

This loads the basic constants defined in a Java class as Ant properties.

```
<loadproperties srcfile="foo.class">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.ClassConstants"/>
  </filterchain>
</loadproperties>
```

Convenience method:

```
<loadproperties srcfile="foo.class">
  <filterchain>
    <classconstants/>
  </filterchain>
</loadproperties>
```

7.3.8.4 EscapeUnicode

This filter converts its input by changing all non US-ASCII characters into their equivalent unicode escape backslash u plus 4 digits.

since Ant 1.6

Example:

This loads the basic constants defined in a Java class as Ant properties.

```
<loadproperties srcfile="non_ascii_property.properties">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.EscapeUnicode"/>
  </filterchain>
</loadproperties>
```

Convenience method:

```
<loadproperties srcfile="non_ascii_property.properties">
```

```
  <filterchain>
    <escapeunicode/>
  </filterchain>
</loadproperties>
```

7.3.8.5 ExpandProperties

If the data contains data that represents Ant properties (of the form ${...}), that is substituted with the property's actual value.

Example:

This results in the property modifiedmessage holding the value "All these moments will be lost in time, like teardrops in the rain"

```
<echo
  message="All these moments will be lost in time, like teardrops in the
${weather}"
  file="loadfile1.tmp"
  />
<property name="weather" value="rain" />
<loadfile property="modifiedmessage" srcFile="loadfile1.tmp">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.ExpandProperties"/>
  </filterchain>
</loadfile>
```

Convenience method:

```
<echo
  message="All these moments will be lost in time, like teardrops in the
${weather}"
  file="loadfile1.tmp"
  />
<property name="weather" value="rain" />
<loadfile property="modifiedmessage" srcFile="loadfile1.tmp">
  <filterchain>
    <expandproperties/>
  </filterchain>
</loadfile>
```

7.3.8.6 HeadFilter

This filter reads the first few lines from the data supplied to it.

Parameter Name	Parameter Value	Required
lines	Number of lines to be read. Defaults to "10" A negative value means that all lines are passed (useful with skip)	No
skip	Number of lines to be skipped (from the beginning). Defaults to "0"	No

Example:

This stores the first 15 lines of the supplied data in the property `${src.file.head}`

```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.HeadFilter">
      <param name="lines" value="15"/>
    </filterreader>
  </filterchain>
</loadfile>
```

Convenience method:
```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <headfilter lines="15"/>
  </filterchain>
</loadfile>
```

This stores the first 15 lines, skipping the first 2 lines, of the supplied data in the porperty ${src.file.head}.
(Means: lines 3-17)
```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <headfilter lines="15" skip="2"/>
  </filterchain>
</loadfile>
```

See the testcases for more examples (src\etc\testcases\filters\head-tail.xml in the source distribution).

7.3.8.7 LineContains

This filter includes only those lines that contain all the user-specified strings.

Parameter Type	Parameter Value	Required
contains	Substring to be searched for.	Yes

Example:

This will include only those lines that contain foo and bar.
```
<filterreader classname="org.apache.tools.ant.filters.LineContains">
  <param type="contains" value="foo"/>
  <param type="contains" value="bar"/>
</filterreader>
```

Convenience method:
```
<linecontains>
  <contains value="foo">
  <contains value="bar">
</linecontains>
```

7.3.8.8 LineContainsRegExp

Filter which includes only those lines that contain the user-specified regular expression matching strings.

Parameter Type	Parameter Value	Required
regexp	Pattern of the substring to be searched for.	Yes

Example:

This will fetch all those lines that contain the pattern foo
```
<filterreader classname="org.apache.tools.ant.filters.LineContainsRegExp">
  <param type="regexp" value="foo*"/>
</filterreader>
```

Convenience method:
```
<linecontainsregexp>
  <regexp pattern="foo*">
</linecontainsregexp>
```

7.3.8.9 PrefixLines

Attaches a prefix to every line.

Parameter Name	Parameter Value	Required
prefix	Prefix to be attached to lines.	Yes

Example:

This will attach the prefix Foo to all lines.

```
<filterreader classname="org.apache.tools.ant.filters.PrefixLines">
  <param name="prefix" value="Foo"/>
</filterreader>
```

Convenience method:

```
<prefixlines prefix="Foo"/>
```

7.3.8.10 ReplaceTokens

This filter reader replaces all strings that are sandwiched between begintoken and endtoken with user defined values.

Parameter Type	Parameter Name	Parameter Value	Required
tokenchar	begintoken	Character marking the beginning of a token. Defaults to @	No
tokenchar	endtoken	Character marking the end of a token. Defaults to @	No
token	User defined String.	User defined search String	Yes

Example:

This replaces occurences of the string @DATE@ in the data with today's date and stores it in the property ${src.file.replaced}

```
<tstamp/>
<loadfile srcfile="${src.file}" property="${src.file.replaced}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.ReplaceTokens">
      <param type="token" name="DATE" value="${TODAY}"/>
    </filterreader>
  </filterchain>
</loadfile>
```

Convenience method:

```
<tstamp/>
<loadfile srcfile="${src.file}" property="${src.file.replaced}">
  <filterchain>
    <replacetokens>
      <token key="DATE" value="${TODAY}"/>
    </replacetokens>
  </filterchain>
</loadfile>
```

7.3.8.11 StripJavaComments

This filter reader strips away comments from the data, using Java syntax guidelines. This filter does not take in any parameters.

Example:

```
<loadfile srcfile="${java.src.file}" property="${java.src.file.nocomments}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.StripJavaComments"/>
  </filterchain>
</loadfile>
```

Convenience method:

```
<loadfile srcfile="${java.src.file}" property="${java.src.file.nocomments}">
  <filterchain>
```

```
    <stripjavacomments/>
  </filterchain>
</loadfile>
```

7.3.8.12 StripLineBreaks

This filter reader strips away specific characters from the data supplied to it.

Parameter Name	Parameter Value	Required
linebreaks	Characters that are to be stripped out. Defaults to "\r\n"	No

Examples:

This strips the '\r' and '\n' characters.

```
<loadfile srcfile="${src.file}" property="${src.file.contents}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.StripLineBreaks"/>
  </filterchain>
</loadfile>
```

Convenience method:

```
<loadfile srcfile="${src.file}" property="${src.file.contents}">
  <filterchain>
    <striplinebreaks/>
  </filterchain>
</loadfile>
```

This treats the '(' and ')' characters as line break characters and strips them.

```
<loadfile srcfile="${src.file}" property="${src.file.contents}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.StripLineBreaks">
      <param name="linebreaks" value="()"/>
    </filterreader>
  </filterchain>
</loadfile>
```

7.3.8.13 StripLineComments

This filter removes all those lines that begin with strings that represent comments as specified by the user.

Parameter Type	Parameter Value	Required
comment	Strings that identify a line as a comment when they appear at the start of the line.	Yes

Examples:

This removes all lines that begin with #, --, REM, rem and //

```
<filterreader classname="org.apache.tools.ant.filters.StripLineComments">
  <param type="comment" value="#"/>
  <param type="comment" value="--"/>
  <param type="comment" value="REM "/>
  <param type="comment" value="rem "/>
  <param type="comment" value="//"/>
</filterreader>
```

Convenience method:

```
<striplinecomments>
  <comment value="#"/>
  <comment value="--"/>
  <comment value="REM "/>
  <comment value="rem "/>
  <comment value="//"/>
</striplinecomments>
```

7.3.8.14 TabsToSpaces

This filter replaces tabs with spaces

Parameter Name	Parameter Value	Required
lines	tablength Defaults to "8"	No

Examples:

This replaces tabs in ${src.file} with spaces.

```
<loadfile srcfile="${src.file}" property="${src.file.notab}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.TabsToSpaces"/>
  </filterchain>
</loadfile>
```

Convenience method:

```
<loadfile srcfile="${src.file}" property="${src.file.notab}">
  <filterchain>
    <tabstospaces/>
  </filterchain>
</loadfile>
```

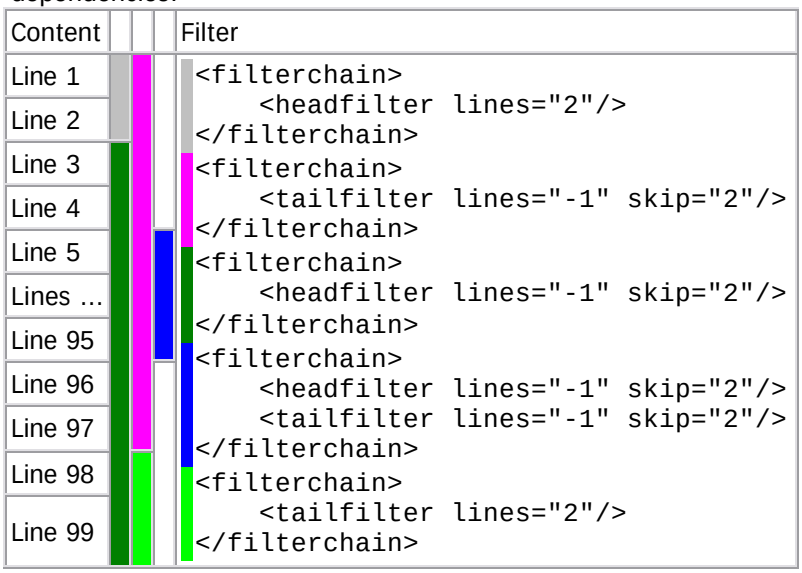
7.3.8.15 TailFilter

This filter reads the last few lines from the data supplied to it.

Parameter Name	Parameter Value	Required
lines	Number of lines to be read. Defaults to "10" A negative value means that all lines are passed (useful with skip)	No
skip	Number of lines to be skipped (from the end). Defaults to "0"	No

Background:

With HeadFilter and TailFilter you can extract each part of a text file you want. This graphic shows the dependencies:



Examples:

This stores the last 15 lines of the supplied data in the property ${src.file.tail}

```
<loadfile srcfile="${src.file}" property="${src.file.tail}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.TailFilter">
```

```
      <param name="lines" value="15"/>
    </filterreader>
  </filterchain>
</loadfile>
```

Convenience method:

```
<loadfile srcfile="${src.file}" property="${src.file.tail}">
  <filterchain>
    <tailfilter lines="15"/>
  </filterchain>
</loadfile>
```

This stores the last 5 lines of the first 15 lines of the supplied data in the property ${src.file.mid}

```
<loadfile srcfile="${src.file}" property="${src.file.mid}">
  <filterchain>
    <filterreader classname="org.apache.tools.ant.filters.HeadFilter">
      <param name="lines" value="15"/>
    </filterreader>
    <filterreader classname="org.apache.tools.ant.filters.TailFilter">
      <param name="lines" value="5"/>
    </filterreader>
  </filterchain>
</loadfile>
```

Convenience method:

```
<loadfile srcfile="${src.file}" property="${src.file.mid}">
  <filterchain>
    <headfilter lines="15"/>
    <tailfilter lines="5"/>
  </filterchain>
</loadfile>
```

This stores the last 10 lines, skipping the last 2 lines, of the supplied data in the porperty ${src.file.head}. (Means: if supplied data contains 60 lines, lines 49-58 are extracted)

```
<loadfile srcfile="${src.file}" property="${src.file.head}">
  <filterchain>
    <tailfilter lines="10" skip="2"/>
  </filterchain>
</loadfile>
```

7.3.8.16 DeleteCharacters

This filter deletes specified characters.

since Ant 1.6

This filter is only available in the convenience form.

Parameter Name	Parameter Value	Required
chars	The characters to delete. This attribute is backslash enabled.	Yes

Examples:

Delete tabs and returns from the data.

```
<deletecharacters chars="\t\r"/>
```

7.3.8.17 ConcatFilter

This filter prepends or appends the content file to the filtered files.

since Ant 1.6

Parameter Name	Parameter Value	Required

prepend	The name of the file which content should be prepended to the file.	No
append	The name of the file which content should be appended to the file.	No

Examples:

Do nothing:

```
<filterchain>
    <concatfilter/>
</filterchain>
```

Adds a license text before each java source:

```
<filterchain>
    <concatfilter prepend="apache-license-java.txt"/>
</filterchain>
```

7.3.8.18 TokenFilter

This filter tokenizes the inputstream into strings and passes these strings to filters of strings. Unlike the other filterreaders, this does not support params, only convenience methods are implemented. The tokenizer and the string filters are defined by nested elements.

since Ant 1.6

Only one tokenizer element may be used, the LineTokenizer is the default if none are specified. A tokenizer splits the input into token strings and trailing delimiter strings.

There may be zero or more string filters. A string filter processes a token and either returns a string or a null. It the string is not null it is passed to the next filter. This proceeds until all the filters are called. If a string is returned after all the filters, the string is outputs with its associated token delimitier (if one is present). The trailing delimiter may be overridden by the delimOutput attribute.

blackslash interpretation A number of attributes (including delimOutput) interpret backslash escapes. The following are understood: \n, \r, \f, \t and \\.

Attribute	Description	Required
delimOutput	This overrides the tokendelimiter returned by the tokenizer if it is not empty. This attribute is backslash enabled.	No

The following tokenizers are provided by the default distribution.
LineTokenizer
FileTokenizer
StringTokenizer

The following string filters are provided by the default distribution.
ReplaceString
ContainsString
ReplaceRegex
ContainsRegex
Trim
IgnoreBlank
DeleteCharacters

The following string filters are provided by the optional distribution.
ScriptFilter

Some of the filters may be used directly within a filter chain. In this case a tokenfilter is created implicitly. An extra attribute "byline" is added to the filter to specify whether to use a linetokenizer (byline="true") or a filetokenizer (byline="false"). The default is "true".

7.3.8.19 LineTokenizer

This tokenizer splits the input into lines. The tokenizer delimits lines by "\r", "\n" or "\r\n". This is the default tokenizer.

Attribute	Description	Required
includeDelims	Include the line endings in the token. Default is false.	No

Examples:

Convert input current line endings to unix style line endings.

```
<tokenfilter delimoutput="\n"/>
```

Remove blank lines.

```
<tokenfilter>
    <ignoreblank/>
</tokenfilter>
```

7.3.8.20 FileTokenizer

This tokenizer treats all the input as a token. So be careful not to use this on very large input.

Examples:

Replace the first occurance of package with //package.

```
<tokenfilter>
      <filetokenizer/>
      <replaceregex pattern="([\n\r]+[ \t]*|^[ \t]*)package"
                    flags="s"
                    replace="\1//package"/>
</tokenfilter>
```

7.3.8.21 StringTokenizer

This tokenizer is based on java.util.StringTokenizer. It splits up the input into strings separated by white space, or by a specified list of delimiting characters. If the stream starts with delimiter characters, the first token will be the empty string (unless the delimsaretokens attribute is used).

Attribute	Description	Required
delims	The delimiter characters. White space is used if this is not set. (White space is defined in this case by java.lang.Character.isWhitespace()).	No
delimsaretokens	If this is true, each delimiter character is returned as a token. Default is false.	No
suppressdelims	If this is true, delimiters are not returned. Default is false.	No
includeDelims	Include the delimiters in the token. Default is false.	No

Examples:

Surround each non space token with a "[]".

```
<tokenfilter>
    <stringtokenizer/>
    <replaceregex pattern="(.+)" replace="[\1]"/>
</tokenfilter>
```

7.3.8.22 ReplaceString

This is a simple filter to replace strings. This filter may be used directly within a filterchain.

Attribute	Description	Required
from	The string that must be replaced.	Yes
to	The new value for the replaced string. When omitted an empty string is used.	No

Examples:

Replace "sun" with "moon".

```
<tokenfilter>
    <replacestring from="sun" to="moon"/>
</tokenfilter>
```

7.3.8.23 ContainsString

This is a simple filter to filter tokens that contains a specified string.

Attribute	Description	Required
contains	The string that the token must contain.	Yes

Examples:

Include only lines that contain "foo";

```
<tokenfilter>
    <containsstring contains="foo"/>
</tokenfilter>
```

7.3.8.24 ReplaceRegex

This string filter replaces regular expressions. See ReplaceRegexp for an explanation on regular expressions. This filter may be used directly within a filterchain.

Attribute	Description	Required
pattern	The regular expression pattern to match in the token.	Yes
replace	The substitution pattern to replace the matched regular expression. When omitted an empty string is used.	No
flags	See ReplaceRegexp for an explanation of regex flags.	No

Examples:

Replace all occurances of "hello" with "world", ignoring case.

```
<tokenfilter>
    <replaceregex pattern="hello" replace="world" flags="gi"/>
</tokenfilter>
```

7.3.8.25 ContainsRegex

This filters strings that match regular expressions. The filter may optionally replace the matched regular expression. See ReplaceRegexp for an explanation on regular expressions. This filter may be used directly within a filterchain.

Attribute	Description	Required
pattern	The regular expression pattern to match in the token.	Yes
replace	The substitution pattern to replace the matched regular expression. When omitted the orignal token is returned.	No
flags	See ReplaceRegexp for an explanation of regex flags.	No

Examples:

Filter lines that contain "hello" or "world", ignoring case.

```
<tokenfilter>
    <containsregex pattern="(hello|world)" flags="i"/>
</tokenfilter>
```

This example replaces lines like "SUITE(TestSuite, bits);" with "void register_bits();" and removes other lines.

```
<tokenfilter>
    <containsregex
        pattern="^ *SUITE\(.*,\s*(.*)\s*\).*"
        replace="void register_\1();"/>
</tokenfilter>
```

7.3.8.26 Trim

This filter trims whitespace from the start and end of tokens. This filter may be used directly within a filterchain.

7.3.8.27 IgnoreBlank

This filter removes empty tokens. This filter may be used directly within a filterchain.

7.3.8.28 DeleteCharacters

This filter deletes specified characters from tokens.

Attribute	Description	Required
chars	The characters to delete. This attribute is backslash enabled.	Yes

Examples:

Delete tabs from lines, trim the lines and removes empty lines.

```
<tokenfilter>
    <deletecharacters chars="\t"/>
    <trim/>
    <ignoreblank/>
</tokenfilter>
```

7.3.8.29 ScriptFilter

This is an optional filter that executes a script in a Apache BSF supported language.

See the Script task for an explanation of scripts and dependencies.

The script is provided with an object self that has getToken() and setToken(String) methods. The getToken() method returns the current token. The setToken(String) method replaces the current token.

This filter may be used directly within a filterchain.

Attribute	Description	Required
language	The programming language the script is written in. Must be a supported Apache BSF language	Yes
src	The location of the script as a file, if not inline	No

Examples:

Convert to uppercase:

```
<tokenfilter>
    <scriptfilter language="javascript">
        self.setToken(self.getToken().toUpperCase());
    </scriptfilter>
</tokenfilter>
```

Remove lines containing the string "bad" while copying text files:

```
<copy todir="dist">
  <fileset dir="src" includes="**/*.txt"/>
  <scriptfilter language="beanshell">
      if (self.getToken().indexOf("bad") != -1) {
          self.setToken(null);
      }
  </scriptfilter>
</copy>
```

7.3.8.30 Custom tokenizers and string filters

Custom string filters and tokenizers may be plugged in by extending the interfaces org.apache.tools.ant.filters.TokenFilter.Filter and org.apache.tools.ant.util.Tokenizer respectly. They are defined the build file using <typedef/>. For example a string filter that capitalizes words may be declared as:

```
package my.customant;
import org.apache.tools.ant.filters.TokenFilter;

public class Capitalize
    implements TokenFilter.Filter
{
    public String filter(String token) {
        if (token.length() == 0)
            return token;
        return token.substring(0, 1).toUpperCase() +
                token.substring(1);
   }
}
```

This may be used as follows:

```
  <typedef type="capitalize" classname="my.customant.Capitalize"
           classpath="my.customant.path"/>
  <copy file="input" tofile="output">
    <filterchain>
      <tokenfilter>
        <stringtokenizer/>
        <capitalize/>
      </tokenfilter>
    </filterchain>
  </copy>
```

7.3.9 FilterSet

FilterSets are groups of filters. Filters can be defined as token-value pairs or be read in from a file. FilterSets can appear inside tasks that support this feature or at the same level as <target> - i.e., as children of <project>.

FilterSets support the id and refid attributes. You can define a FilterSet with an id attribute and then refer to that definition from another FilterSet with a refid attribute. It is also possible to nest filtersets into filtersets to get a set union of the contained filters.

In addition, FilterSets can specify begintoken and/or endtoken attributes to define what to match.
Filtersets are used for doing replacements in tasks such as <copy>, etc.

Note: When a filterset is used in an operation, the files are processed in text mode and the filters applied line by line. This means that the copy operations will typically corrupt binary files. When applying filters you should ensure that the set of files being filtered are all text files.

7.3.9.1 Filterset

Attribute	Description	Default	Required
begintoken	The string marking the beginning of a token (eg., @DATE@).	@	No
endtoken	The string marking the end of a token (eg., @DATE@).	@	No

7.3.9.2 Filter

Attribute	Description	Required
token	The token to replace (eg., @DATE@)	Yes
value	The value to replace it with (eg., Thursday, April 26, 2001).	Yes

7.3.9.3 Filtersfile

Attribute	Description	Required
file	A properties file of name-value pairs from which to load the tokens.	Yes

Examples

You are copying the version.txt file to the dist directory from the build directory but wish to replace the token @DATE@ with today's date.

```
<copy file="${build.dir}/version.txt" toFile="${dist.dir}/version.txt">
  <filterset>
    <filter token="DATE" value="${TODAY}"/>
  </filterset>
</copy>
```

You are copying the version.txt file to the dist directory from the build directory but wish to replace the token %DATE* with today's date.

```
<copy file="${build.dir}/version.txt" toFile="${dist.dir}/version.txt">
  <filterset begintoken="%" endtoken="*">
    <filter token="DATE" value="${TODAY}"/>
  </filterset>
</copy>
```

Copy all the docs but change all dates and appropriate notices as stored in a file.

```
<copy toDir="${dist.dir}/docs">
  <fileset dir="${build.dir}/docs">
    <include name="**/*.html">
  </fileset>
  <filterset begintoken="%" endtoken="*">
    <filtersfile file="${user.dir}/dist.properties"/>
  </filterset>
</copy>
```

Define a FilterSet and reference it later.

```
<filterset id="myFilterSet" begintoken="%" endtoken="*">
  <filter token="DATE" value="${TODAY}"/>
</filterset>

<copy file="${build.dir}/version.txt" toFile="${dist.dir}/version.txt">
  <filterset refid="myFilterSet"/>
</copy>
```

7.3.10 PatternSet

Patterns can be grouped to sets and later be referenced by their id attribute. They are defined via a patternset element, which can appear nested into a FileSet or a directory-based task that constitutes an implicit FileSet. In addition, patternsets can be defined as a stand alone element at the same level as target — i.e., as children of project as well as as children of target.
Patterns can be specified by nested <include>, or <exclude> elements or the following attributes.

Attribute	Description
includes	comma- or space-separated list of patterns of files that must be included. All files are included when omitted.
includesfile	the name of a file; each line of this file is taken to be an include pattern. You can specify more than one include file by using a nested includesfile elements.
excludes	comma- or space-separated list of patterns of files that must be excluded; no files (except default excludes) are excluded when omitted.
excludesfile	the name of a file; each line of this file is taken to be an exclude pattern. You can specify more than one exclude file by using a nested excludesfile elements.

Parameters specified as nested elements

include* and *exclude

Each such element defines a single pattern for files to include or exclude.

Attribute	Description	Required
name	the pattern to in/exclude.	Yes
if	Only use this pattern if the named property is set.	No
unless	Only use this pattern if the named property is not set.	No

includesfile* and *excludesfile

If you want to list the files to include or exclude external to your build file, you should use the includesfile/excludesfile attributes or elements. Using the attribute, you can only specify a single file of each type, while the nested elements can be specified more than once - the nested elements also support if/unless attributes you can use to test the existance of a property.

Attribute	Description	Required
name	the name of the file holding the patterns to in/exclude.	Yes
if	Only read this file if the named property is set.	No
unless	Only read this file if the named property is not set.	No

patternset

Patternsets may be nested within one another, adding the nested patterns to the parent patternset.

Examples

```
<patternset id="non.test.sources">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</patternset>
```

Builds a set of patterns that matches all .java files that do not contain the text Test in their name. This set can be referred to via <patternset refid="non.test.sources"/>, by tasks that support this feature, or by FileSets.

Note that while the includes and excludes attributes accept multiple elements separated by commas or spaces, the nested <include> and <exclude> elements expect their name attribute to hold a single pattern.

The nested elements allow you to use if and unless arguments to specify that the element should only be used if a property is set, or that it should be used only if a property is not set.

For example

```
<patternset id="sources">
  <include name="std/**/*.java"/>
  <include name="prof/**/*.java" if="professional"/>
```

```
  <exclude name="**/*Test*"/>
</patternset>
```

will only include the files in the sub-directory prof if the property professional is set to some value.

The two sets
```
<patternset includesfile="some-file"/>
```

and
```
<patternset>
  <includesfile name="some-file"/>
<patternset/>
```

are identical. The include patterns will be read from the file some-file, one pattern per line.
```
<patternset>
  <includesfile name="some-file"/>
  <includesfile name="${some-other-file}"
                if="some-other-file"
  />
<patternset/>
```

will also read include patterns from the file the property some-other-file points to, if a property of that name has been defined.

7.3.11 Permissions

Permissions represents a set of security permissions granted or revoked to a specific part code executed in the JVM where ant is running in. The actual Permissions are specified via a set of nested permission items either <grant>ed or <revoke>d.

In the base situation a base set of permissions granted. Extra permissions can be granted. A granted permission can be overruled by revoking a permission. The security manager installed by the permissions will throw an SecurityException if the code subject to these permissions try to use an permission that has not been granted or that has been revoked.

Nested elements

grant

Indicates a specific permission is always granted. Its attributes indicate which permissions are granted.

Attribute	Description	Required
class	The fully qualified name of the Permission class.	Yes
name	The name of the Permission. The actual contents depends on the Permission class.	No
actions	The actions allowed. The actual contents depend on the Permission class and name.	No

Implied permissions are granted.
Please note that some Permission classes may actually need a name and / or actions in order to function properly. The name and actions are parsed by the actual Permission class.

revoke

Indicates a specific permission is revoked.

Attribute	Description	Required
class	The fully qualified name of the Permission class.	Yes
name	The name of the Permission. The actual contents depends on the Permission class.	No
actions	The actions allowed. The actual contents depend on the Permission class and name.	No

Implied permissions are not resolved and therefore also not revoked.

The name can handle the * wildcard at the end of the name, in which case all permissions of the specified class of which the name starts with the specified name (excluding the *) are revoked. Note that the - wildcard often supported by the granted properties is not supported. If the name is left empty all names match, and are revoked. If the actions are left empty all actions match, and are revoked.

7.3.11.1 Base set

A permissions set implictly contains the following permissions:

```
<grant class="java.net.SocketPermission" name="localhost:1024-" actions="listen">
<grant class="java.util.PropertyPermission" name="java.version" actions="read">
<grant class="java.util.PropertyPermission" name="java.vendor" actions="read">
<grant class="java.util.PropertyPermission" name="java.vendor.url" actions="read">
<grant class="java.util.PropertyPermission" name="java.class.version" actions="read">
<grant class="java.util.PropertyPermission" name="os.name" actions="read">
<grant class="java.util.PropertyPermission" name="os.version" actions="read">
<grant class="java.util.PropertyPermission" name="os.arch" actions="read">
<grant class="java.util.PropertyPermission" name="file.encoding" actions="read">
<grant class="java.util.PropertyPermission" name="file.separator" actions="read">
<grant class="java.util.PropertyPermission" name="path.separator" actions="read">
<grant class="java.util.PropertyPermission" name="line.separator" actions="read">
<grant class="java.util.PropertyPermission" name="java.specification.version" actions="read">
<grant class="java.util.PropertyPermission" name="java.specification.vendor" actions="read">
<grant class="java.util.PropertyPermission" name="java.specification.name" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.specification.version" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.specification.vendor" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.specification.name" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.version" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.vendor" actions="read">
<grant class="java.util.PropertyPermission" name="java.vm.name" actions="read">
```

These permissions can be revoked via <revoke> elements if necessary.

Examples

```
<permissions>
  <grant class="java.security.AllPermission"/>
  <revoke class="java.util.PropertyPermission"/>
</permissions>
```

Grants all permissions to the code except for those handling Properties.

```
<permissions>
  <grant class="java.net.SocketPermission" name="foo.bar.com" action="connect"/>
  <grant class="java.util.PropertyPermission" name="user.home"
action="read,write"/>
</permissions>
```

Grants the base set of permissions with the addition of a SocketPermission to connect to foo.bar.com and the permission to read and write the user.home system property.

7.3.12 PropertySet

Since Ant 1.6

Groups a set of properties to be used by reference in a task that supports this.

Attribute	Description	Required
dynamic	Whether to reevaluate the set everytime the set is used. Default is "true".	No

7.3.12.1 Parameters specified as nested elements

propertyref

Selects properties from the current project to be included in the set.

Attribute	Description	Required
name	Select the property with the given name.	Exactly one of these.
prefix	Select the properties whose name starts with the given string.	
regexp	Select the properties that match the given regular expression. Similar to regexp type mappers, this equires a supported regular expression library.	
builtin	Selects a builtin set of properties. Valid values for this attribute are all for all Ant properties, system for the system properties and commandline for all properties specified on the command line when invoking Ant (plus a number of special internal Ant properties).	

propertyset

A propertyset can be used as the set union of more propertysets.

For example:

```
<propertyset id="properties-starting-with-foo">
  <propertyref prefix="foo"/>
</propertyset>
<propertyset id="properties-starting-with-bar">
  <propertyref prefix="bar"/>
</propertyset>
<propertyset id="my-set">
  <propertyset refid="properties-starting-with-foo"/>
  <propertyset refid="properties-starting-with-bar"/>
</propertyset>
```

collects all properties whose name starts with either "foo" or "bar" in the set named "my-set".

mapper

A mapper - at maximum one mapper can be specified. The mapper is used to change the names of the property keys, for example:

```
<propertyset id="properties-starting-with-foo">
  <propertyref prefix="foo"/>
  <mapper type="glob" from="foo*" to="bar*"/>
</propertyset>
```

collects all properties whose name starts with "foo", but changes the names to start with "bar" instead.

7.3.13 Selectors

Selectors are a mechanism whereby the files that make up a fileset can be selected based on criteria other than filename as provided by the <include> and <exclude> tags.

7.3.13.1 How to use a Selector

A selector is an element of FileSet, and appears within it. It can also be defined outside of any target by using the <selector> tag and then using it as a reference.

Different selectors have different attributes. Some selectors can contain other selectors, and these are called Selector Containers. There is also a category of selectors that allow user-defined extensions, called Custom Selectors. The ones built in to Ant are called Core Selectors.

7.3.13.2 Core Selectors

Core selectors are the ones that come standard with Ant. They can be used within a fileset and can be contained within Selector Containers.

The core selectors are:

- <contains> - Select files that contain a particular text string
- <date> - Select files that have been modified either before or after a particular date and time
- <depend> - Select files that have been modified more recently than equivalent files elsewhere
- <depth> - Select files that appear so many directories down in a directory tree
- <different> - Select files that are different from those elsewhere
- <filename> - Select files whose name matches a particular pattern. Equivalent to the include and exclude elements of a patternset.
- <present> - Select files that either do or do not exist in some other location
- <containsregexp> - Select files that match a regular expression
- <size> - Select files that are larger or smaller than a particular number of bytes.
- <type> - Select files that are either regular files or directories.
- <modified> - Select files if the return value of the configured algorithm is different from that stored in a cache.

Contains Selector

The <contains> tag in a FileSet limits the files defined by that fileset to only those which contain the string specified by the text attribute.

Attribute	Description	Required
text	Specifies the text that every file must contain	Yes
casesensitive	Whether to pay attention to case when looking for the string in the text attribute. Default is true.	No
ignorewhitespace	Whether to eliminate whitespace before checking for the string in the text attribute. Default is false.	No

Here is an example of how to use the Contains Selector:

```
<fileset dir="${doc.path}" includes="**/*.html">
    <contains text="script" casesensitive="no"/>
</fileset>
```

Selects all the HTML files that contain the string script.

Date Selector

The <date> tag in a FileSet will put a limit on the files specified by the include tag, so that tags whose last modified date does not meet the date limits specified by the selector will not end up being selected.

Attribute	Description	Required
datetime	Specifies the date and time to test for using a string of the format MM/DD/YYYY HH:MM AM_or_PM.	At least one of the two.
millis	The number of milliseconds since 1970 that should be tested for. It is usually much easier to use the datetime attribute.	
when	Indicates how to interpret the date, whether the files to be selected are those whose last modified times should be before, after, or equal to the specified value.	No

	Acceptable values for this attribute are: before - select files whose last modified date is before the indicated date after - select files whose last modified date is after the indicated date equal - select files whose last modified date is this exact date The default is equal.	

Here is an example of how to use the Date Selector:

```
<fileset dir="${jar.path}" includes="**/*.jar">
    <date datetime="01/01/2001 12:00 AM" when="before"/>
</fileset>
```

Selects all JAR files which were last modified before midnight January 1, 2001.

Depend Selector

The <depend> tag selects files whose last modified date is later than another, equivalent file in another location.

The <depend> tag supports the use of a contained <mapper> element to define the location of the file to be compared against. If no <mapper> element is specified, the identity type mapper is used.

The <depend> selector is case-sensitive.

Attribute	Description	Required
targetdir	The base directory to look for the files to compare against. The precise location depends on a combination of this attribute and the <mapper> element, if any.	Yes
granularity	The number of milliseconds leeway to give before deciding a file is out of date. This is needed because not every file system supports tracking the last modified time to the millisecond level. Default is 0 milliseconds, or 2 seconds on DOS systems.	No

Here is an example of how to use the Depend Selector:

```
<fileset dir="${ant.1.5}/src/main" includes="**/*.java">
    <depend targetdir="${ant.1.4.1}/src/main"/>
</fileset>
```

Selects all the Java source files which were modified in the 1.5 release.

Depth Selector

The <depth> tag selects files based on how many directy levels deep they are in relation to the base directory of the fileset.

Attribute	Description	Required
min	The minimum number of directory levels below the base directory that a file must be in order to be selected. Default is no limit.	At least one of the two.
max	The maximum number of directory levels below the base directory that a file can be and still be selected. Default is no limit.	

Here is an example of how to use the Depth Selector:

```
<fileset dir="${doc.path}" includes="**/*">
    <depth max="1"/>
</fileset>
```

Selects all files in the base directory and one directory below that.

Different Selector

The <different> tag selects files who are deemed to be 'different' from another, equivalent file in another location. The rules for determining difference between two files are as follows:

1. If there is no 'other' file, it's different.
2. Files with different lengths are different.
3. If ignoreFileTimes is turned off, then differing file timestamps will cause files to be regarded as different.
4. Finally a byte-for-byte check is run against the two files

This is a useful selector to work with programs and tasks that don't handle dependency checking properly; Even if a predecessor task always creates its output files, followup tasks can be driven off copies made with a different selector, so their dependencies are driven on the absolute state of the files, not just a timestamp. For example: anything fetched from a web site, or the output of some program. To reduce the amount of checking, when using this task inside a <copy> task, set the preservelastmodified to propagate the timestamp from source file to destintaion file.

The <different> tag supports the use of a contained <mapper> element to define the location of the file to be compared against. If no <mapper> element is specified, the identity type mapper is used.

Attribute	Description	Required
targetdir	The base directory to look for the files to compare against. The precise location depends on a combination of this attribute and the <mapper> element, if any.	Yes
ignoreFileTimes	Whether to use file times in the comparison or not. Default is true (time differences are ignored).	No
granularity	The number of milliseconds leeway to give before deciding a file is out of date. This is needed because not every file system supports tracking the last modified time to the millisecond level. Default is 0 milliseconds, or 2 seconds on DOS systems.	No

Here is an example of how to use the Different Selector:

```
<fileset dir="${ant.1.5}/src/main" includes="**/*.java">
    <different targetdir="${ant.1.4.1}/src/main"
        ignoreFileTimes="true"/>
</fileset>
```

Compares all the Java source files between the 1.4.1 and the 1.5 release and selects those who are different, disregarding file times.

Filename Selector

The <filename> tag acts like the <include> and <exclude> tags within a fileset. By using a selector instead, however, one can combine it with all the other selectors using whatever selector container is desired.

The <filename> selector is case-sensitive.

Attribute	Description	Required
name	The name of files to select. The name parameter can contain the standard Ant wildcard characters.	Yes
casesensitive	Whether to pay attention to case when looking at file names. Default is "true".	No
negate	Whether to reverse the effects of this filename selection, therefore emulating an exclude rather than include tag. Default is "false".	No

Here is an example of how to use the Filename Selector:

```
<fileset dir="${doc.path}" includes="**/*">
    <filename name="**/*.css"/>
</fileset>
```

Selects all the cascading style sheet files.

Present Selector

The <present> tag selects files that have an equivalent file in another directory tree.

The <present> tag supports the use of a contained <mapper> element to define the location of the file to be tested against. If no <mapper> element is specified, the identity type mapper is used.

The <present> selector is case-sensitive.

Attribute	Description	Required
targetdir	The base directory to look for the files to compare against. The precise location depends on a combination of this attribute and the <mapper> element, if any.	Yes
present	Whether we are requiring that a file is present in the src directory tree only, or in both the src and the target directory tree. Valid values are: srconly - select files only if they are in the src directory tree but not in the target directory tree both - select files only if they are present both in the src and target directory trees Default is both. Setting this attribute to "srconly" is equivalent to wrapping the selector in the <not> selector container.	No

Here is an example of how to use the Present Selector:

```
<fileset dir="${ant.1.5}/src/main" includes="**/*.java">
    <present present="srconly" targetdir="${ant.1.4.1}/src/main"/>
</fileset>
```

Selects all the Java source files which are new in the 1.5 release.

Regular Expression Selector

The <containsregexp> tag in a FileSet limits the files defined by that fileset to only those which contain a match to the regular expression specified by the expression attribute.

Attribute	Description	Required
expression	Specifies the regular expression that must match true in every file	Yes

Here is an example of how to use the regular expression Selector:

```
<fileset dir="${doc.path}" includes="*.txt">
    <containsregexp expression="[4-6]\.[0-9]"/>
</fileset>
```

Selects all the text files that match the regular expression (have a 4,5 or 6 followed by a period and a number from 0 to 9).

Size Selector

The <size> tag in a FileSet will put a limit on the files specified by the include tag, so that tags which do not meet the size limits specified by the selector will not end up being selected.

Attribute	Description	Required
value	The size of the file which should be tested for.	Yes
units	The units that the value attribute is expressed in. When using the standard single letter SI designations, such as "k","M", or "G", multiples of 1000 are used. If you want to use power of 2 units, use the IEC standard: "Ki" for 1024, "Mi" for 1048576, and so on. The default is no units, which means the value attribute expresses the exact number of bytes.	No
when	Indicates how to interpret the size, whether the files to be selected should be larger, smaller, or equal to that value. Acceptable values for this attribute are: less - select files less than the indicated size more - select files greater than the indicated size equal - select files this exact size	No

	The default is less.	

Here is an example of how to use the Size Selector:

```
<fileset dir="${jar.path}">
  <patternset>
    <include name="**/*.jar"/>
  </patternset>
  <size value="4" units="Ki" when="more"/>
</fileset>
```

Selects all JAR files that are larger than 4096 bytes.

Type Selector

The <type> tag selects files of a certain type: directory or regular.

Attribute	Description	Required
type	The type of file which should be tested for. Acceptable values are: file - regular files dir - directories	Yes

Here is an example of how to use the Type Selector to select only directories in ${src}

```
<fileset dir="${src}">
  <type type="dir"/>
</fileset>
```

The Type Selector is often used in conjunction with other selectors. For example, to select files that also exist in a template directory, but avoid selecting empty directories, use:

```
<fileset dir="${src}">
    <and>
        <present targetdir="template"/>
        <type type="file"/>
    </and>
</fileset>
```

Modified Selector

The <modified> selector computes a value for a file, compares that to the value stored in a cache and select the file, if these two values differ.

Because this selector is highly configurable the order in which the selection is done is:

1. get the absolute path for the file
2. get the cached value from the configured cache (absolute path as key)
3. get the new value from the configured algorithm
4. compare these two values with the configured comparator
5. update the cache if needed and requested
6. do the selection according to the comparison result

The comparison, computing of the hashvalue and the store is done by implementation of special interfaces. Therefore they may provide additional parameters.

Attribute	Description	Required
algorithm	The type of algorithm should be used. Acceptable values are (further information see later): hashvalue - HashvalueAlgorithm digest - DigestAlgorithm	No, defaults to digest

cache	The type of cache should be used. Acceptable values are (further information see later): propertyfile - PropertyfileCache	No, defaults to propertyfile
comparator	The type of comparator should be used. Acceptable values are (further information see later): equal - EqualComparator rule - java.text.RuleBasedCollator	No, defaults to equal
update	Should the cache be updated when values differ? (boolean)	No, defaults to true
seldirs	Should directories be selected? (boolean)	No, defaults to true

These attributes can be set with nested <param/> tags. With <param/> tags you can set other values too - as long as they are named according to the following rules:

- **algorithm** : same as attribute algorithm
- **cache** : same as attribute cache
- **comparator** : same as attribute cache
- **update** : same as attribute comparator
- **seldirs** : same as attribute seldirs
- **algorithm.*** : Value is transfered to the algorithm via its setXX-methods
- **cache.*** : Value is transfered to the cache via its setXX-methods
- **comparator.*** : Value is transfered to the comparator via its setXX-methods

Algorithm´s	
Name	Description
hashvalue	Reads the content of a file into a java.lang.String and use thats hashValue(). No additional configuration required.
digest	Uses java.security.MessageDigest. This Algorithm supports the following attributes: algorithm.algorithm (optional): Name of the Digest algorithm (e.g. 'MD5' or 'SHA', default = MD5) algorithm.provider (optional): Name of the Digest provider (default = null)
Cache´s	
propertyfile	Use the java.util.Properties class and its possibility to load and store to file. This Cache implementation supports the following attributes: cache.cachefile (optional): Name of the properties-file (default = cache.properties)
Comparator´s	
equal	Very simple object comparison.
rule	Uses java.text.RuleBasedCollator for Object comparison.

Here are some examples of how to use the Modified Selector:

```
<copy todir="dest">
    <filelist dir="src">
        <modified/>
    </filelist>
</copy>
```

This will copy all files from src to dest which content has changed. Using an updating PropertyfileCache with cache.properties and MD5-DigestAlgorithm.

```
<copy todir="dest">
    <filelist dir="src">
        <modified update="true"
                  seldirs="true"
                  cache="propertyfile"
                  algorithm="digest"
                  comparator="equal">
```

```
                <param name="cache.cachefile"     value="cache.properties"/>
                <param name="algorithm.algorithm" value="MD5"/>
            </modified>
        </filelist>
    </copy>
```

This is the same example rewritten as CoreSelector with setting the all the values (same as defaults are).

```
    <copy todir="dest">
        <filelist dir="src">
            <custom
class="org.apache.tools.ant.types.selectors.modifiedselector.ModifiedSelector">
                <param name="update"     value="true"/>
                <param name="seldirs"    value="true"/>
                <param name="cache"      value="propertyfile"/>
                <param name="algorithm"  value="digest"/>
                <param name="comparator" value="equal"/>
                <param name="cache.cachefile"     value="cache.properties"/>
                <param name="algorithm.algorithm" value="MD5"/>
            </custom>
        </filelist>
    </copy>
```

And this is the same rewritten as CustomSelector.

```
  <target name="generate-and-upload-site">
      <echo> generate the site using forrest </echo>
      <antcall target="site"/>

      <echo> upload the changed file </echo>
      <ftp server="${ftp.server}" userid="${ftp.user}" password="${ftp.pwd}">
          <fileset dir="htdocs/manual">
              <modified/>
          </fileset>
      </ftp>
  </target>
```

A useful scenario for this selector inside a build environment for homepage generation (e.g. with Apache Forrest). Here all changed files are uploaded to the server. The CacheSelector saves therefore much upload time.

7.3.13.3 Selector Containers

To create more complex selections, a variety of selectors that contain other selectors are available for your use. They combine the selections of their child selectors in various ways.

The selector containers are:

- <and> - select a file only if all the contained selectors select it.
- <majority> - select a file if a majority of its selectors select it.
- <none> - select a file only if none of the contained selectors select it.
- <not> - can contain only one selector, and reverses what it selects and doesn't select.
- <or> - selects a file if any one of the contained selectors selects it.
- <selector> - contains only one selector and forwards all requests to it without alteration, provided that any "if" or "unless" conditions are met. This is the selector to use if you want to define a reference. It is usable as an element of <project>. It is also the one to use if you want selection of files to be dependent on Ant property settings.

All selector containers can contain any other selector, including other containers, as an element. Using containers, the selector tags can be arbitrarily deep. Here is a complete list of allowable selector elements within a container:

- <and>
- <contains>
- <custom>
- <date>
- <depend>
- <depth>
- <filename>
- <majority>
- <none>
- <not>
- <or>
- <present>
- <selector>
- <size>

7.3.13.4 And Selector

The <and> tag selects files that are selected by all of the elements it contains. It returns as soon as it finds a selector that does not select the file, so it is not guaranteed to check every selector.

Here is an example of how to use the And Selector:

```
<fileset dir="${dist}" includes="**/*.jar">
    <and>
        <size value="4" units="Ki" when="more"/>
        <date datetime="01/01/2001 12:00 AM" when="before"/>
    </and>
</fileset>
```

Selects all the JAR file larger than 4096 bytes which haven't been update since the last millenium.

7.3.13.5 Majority Selector

The <majority> tag selects files provided that a majority of the contained elements also select it. Ties are dealt with as specified by the allowtie attribute.

Attribute	Description	Required
allowtie	Whether files should be selected if there are an even number of selectors selecting them as are not selecting them. Default is true.	No

Here is an example of how to use the Majority Selector:

```
<fileset dir="${docs}" includes="**/*.html">
    <majority>
        <contains text="project" casesensitive="false"/>
        <contains text="taskdef" casesensitive="false"/>
        <contains text="IntrospectionHelper" casesensitive="true"/>
    </majority>
</fileset>
```

Selects all the HTML files which contain at least two of the three phrases "project", "taskdef", and "IntrospectionHelper" (this last phrase must match case exactly).

7.3.13.6 None Selector

The <none> tag selects files that are not selected by any of the elements it contains. It returns as soon as it finds a selector that selects the file, so it is not guaranteed to check every selector.
Here is an example of how to use the None Selector:

```
<fileset dir="${src}" includes="**/*.java">
    <none>
        <present targetdir="${dest}"/>
        <present targetdir="${dest}">
            <mapper type="glob" from="*.java" to="*.class"/>
        </present>
    </none>
</fileset>
```

Selects only Java files which do not have equivalent java or class files in the dest directory.

7.3.13.7 Not Selector

The <not> tag reverses the meaning of the single selector it contains.

Here is an example of how to use the Not Selector:

```
<fileset dir="${src}" includes="**/*.java">
    <not>
        <contains text="test"/>
    </not>
</fileset>
```

Selects all the files in the src directory that do not contain the string "test".

7.3.13.8 Or Selector

The <or> tag selects files that are selected by any one of the elements it contains. It returns as soon as it finds a selector that selects the file, so it is not guaranteed to check every selector.
Here is an example of how to use the Or Selector:

```
<fileset dir="${basedir}">
    <or>
        <depth max="0"/>
        <filename name="*.png"/>
        <filename name="*.gif"/>
        <filename name="*.jpg"/>
    </or>
</fileset>
```

Selects all the files in the top directory along with all the image files below it.

7.3.13.9 Selector Reference

The <selector> tag is used to create selectors that can be reused through references. It is the only selector which can be used outside of any target, as an element of the <project> tag. It can contain only one other selector, but of course that selector can be a container.

The <selector> tag can also be used to select files conditionally based on whether an Ant property exists or not. This functionality is realized using the "if" and "unless" attributes in exactly the same way they are used on targets or on the <include> and <exclude> tags within a <patternset>.

Attribute	Description	Required
if	Allow files to be selected only if the named property is set.	No

unless	Allow files to be selected only if the named property is not set.	No

Here is an example of how to use the Selector Reference:

```
<project default="all" basedir="./ant">

    <selector id="completed">
        <none>
            <depend targetdir="build/classes">
                <mapper type="glob" from="*.java" to="*.class"/>
            </depend>
            <depend targetdir="docs/manual/api">
                <mapper type="glob" from="*.java" to="*.html"/>
            </depend>
        </none>
    </selector>

    <target>
        <zip>
            <fileset dir="src/main" includes="**/*.java">
                <selector refid="completed"/>
            </fileset>
        </zip>
    </target>

</project>
```

Zips up all the Java files which have an up-to-date equivalent class file and javadoc file associated with them.

And an example of selecting files conditionally, based on whether properties are set:

```
<fileset dir="${working.copy}">
    <or>
        <selector if="include.tests">
            <filename name="**/*Test.class">
        </selector>
        <selector if="include.source">
            <and>
                <filename name="**/*.java">
                <not>
                    <selector unless="include.tests">
                        <filename name="**/*Test.java">
                    </selector>
                </not>
            </and>
        </selector>
    </or>
</fileset>
```

A fileset that conditionally contains Java source files and Test source and class files.

7.3.13.10 Custom Selectors

You can write your own selectors and use them within the selector containers by specifying them within the <custom> tag.

First, you have to write your selector class in Java. The only requirement it must meet in order to be a selector is that it implements the org.apache.tools.ant.types.selectors.FileSelector interface, which contains a single method. See Programming Selectors in Ant for more information.

Once that is written, you include it in your build file by using the <custom> tag.

Attribute	Description	Required
classname	The name of your class that implements org.apache.tools.ant.types.selectors.FileSelector.	Yes
classpath	The classpath to use in order to load the custom selector class. If neither this classpath nor the classpathref are specified, the class will be loaded from the classpath that Ant uses.	No
classpathref	A reference to a classpath previously defined. If neither this reference nor the classpath above are specified, the class will be loaded from the classpath that Ant uses.	No

Here is how you use <custom> to use your class as a selector:

```
<fileset dir="${mydir}" includes="**/*">
    <custom classname="com.mydomain.MySelector">
        <param name="myattribute" value="myvalue"/>
    </custom>
</fileset>
```

A number of core selectors can also be used as custom selectors by specifying their attributes using <param> elements. These are

- Contains Selector with classname org.apache.tools.ant.types.selectors.ContainsSelector
- Date Selector with classname org.apache.tools.ant.types.selectors.DateSelector
- Depth Selector with classname org.apache.tools.ant.types.selectors.DepthSelector
- Filename Selector with classname org.apache.tools.ant.types.selectors.FilenameSelector
- Size Selector with classname org.apache.tools.ant.types.selectors.SizeSelector

Here is the example from the Depth Selector section rewritten to use the selector through <custom>.

```
<fileset dir="${doc.path}" includes="**/*">
    <custom classname="org.apache.tools.ant.types.selectors.DepthSelector">
        <param name="max" value="1"/>
    </custom>
</fileset>
```

Selects all files in the base directory and one directory below that.
For more details concerning writing your own selectors, consult Programming Selectors in Ant.

7.3.14 XMLCatalog

An XMLCatalog is a catalog of public resources such as DTDs or entities that are referenced in an XML document. Catalogs are typically used to make web references to resources point to a locally cached copy of the resource.

This allows the XML Parser, XSLT Processor or other consumer of XML documents to efficiently allow a local substitution for a resource available on the web.

Note: This task uses, but does not depend on external libraries not included in the Ant distribution. See Library Dependencies for more information.

This data type provides a catalog of resource locations based on the OASIS "Open Catalog" standard. The catalog entries are used both for Entity resolution and URI resolution, in accordance with the org.xml.sax.EntityResolver and javax.xml.transform.URIResolver interfaces as defined in the Java API for XML Processing (JAXP) Specification.

For example, in a web.xml file, the DTD is referenced as:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
```

```
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

The XML processor, without XMLCatalog support, would need to retrieve the DTD from the URL specified whenever validation of the document was required.

This can be very time consuming during the build process, especially where network throughput is limited. Alternatively, you can do the following:

1. Copy web-app_2_2.dtd onto your local disk somewhere (either in the filesystem or even embedded inside a jar or zip file on the classpath).
2. Create an <xmlcatalog> with a <dtd> element whose location attribute points to the file.
3. Success! The XML processor will now use the local copy instead of calling out to the internet.

XMLCatalogs can appear inside tasks that support this feature or at the same level as target - i.e., as children of project for reuse across different tasks, e.g. XML Validation and XSLT Transformation. The XML Validate task uses XMLCatalogs for entity resolution. The XSLT Transformation task uses XMLCatalogs for both entity and URI resolution.

XMLCatalogs are specified as either a reference to another XMLCatalog, defined previously in a build file, or as a list of dtd or entity locations. In addition, external catalog files may be specified in a nested catalogpath , but they will be ignored unless the resolver library from xml-commons is available in the system classpath. **Due to backwards incompatible changes in the resolver code after the release of resolver 1.0, Ant will not support resolver.jar in version 1.0 - we expect a resolver release 1.1 to happen before Ant 1.6 gets released.** A separate classpath for entity resolution may be specified inline via nested classpath elements; otherwise the system classpath is used for this as well.

XMLCatalogs can also be nested inside other XMLCatalogs. For example, a "superset" XMLCatalog could be made by including several nested XMLCatalogs that referred to other, previously defined XMLCatalogs.

Resource locations can be specified either in-line or in external catalog file(s), or both. In order to use an external catalog file, the xml-commons resolver library ("resolver.jar") must be in your path. External catalog files may be either plain text format or XML format. If the xml-commons resolver library is not found in the classpath, external catalog files, specified in catalogpath, will be ignored and a warning will be logged. In this case, however, processing of inline entries will proceed normally.

Currently, only <dtd> and <entity> elements may be specified inline; these roughly correspond to OASIS catalog entry types PUBLIC and URI respectively. By contrast, external catalog files may use any of the entry types defined in the +OASIS specification.

7.3.14.1 Entity/DTD/URI Resolution Algorithm

When an entity, DTD, or URI is looked up by the XML processor, the XMLCatalog searches its list of entries to see if any match. That is, it attempts to match the publicId attribute of each entry with the PublicID or URI of the entity to be resolved. Assuming a matching entry is found, XMLCatalog then executes the following steps:

1. Filesystem lookup
The location is first looked up in the filesystem. If the location is a relative path, the ant project basedir attribute is used as the base directory. If the location specifies an absolute path, it is used as is. Once we have an absolute path in hand, we check to see if a valid and readable file exists at that path. If so, we are done. If not, we proceed to the next step.

2. Classpath lookup
The location is next looked up in the classpath. Recall that jar files are merely fancy zip files. For classpath lookup, the location is used as is (no base is prepended). We use a Classloader to attempt to load the resource

from the classpath. For example, if hello.jar is in the classpath and it contains foo/bar/blat.dtd it will resolve an entity whose location is foo/bar/blat.dtd. Of course, it will not resolve an entity whose location is blat.dtd.

3a. Apache xml-commons resolver lookup
What happens next depends on whether the resolver library from xml-commons is available on the classpath. If so, we defer all further attempts at resolving to it. The resolver library supports extremely sophisticated functionality like URL rewriting and so on, which can be accessed by making the appropriate entries in external catalog files (XMLCatalog does not yet provide inline support for all of the entries defined in the OASIS standard).

3. URL-space lookup
Finally, we attempt to make a URL out of the location. At first this may seem like this would defeat the purpose of XMLCatalogs -- why go back out to the internet? But in fact, this can be used to (in a sense) implement HTTP redirects, substituting one URL for another. The mapped-to URL might also be served by a local web server. If the URL resolves to a valid and readable resource, we are done. Otherwise, we give up. In this case, the XML processor will perform its normal resolution algorithm. Depending on the processor configuration, further resolution failures may or may not result in fatal (i.e. build-ending) errors.

7.3.14.2 XMLCatalog attributes

Attribute	Description	Required
id	a unique name for an XMLCatalog, used for referencing the XMLCatalog's contents from another XMLCatalog	No
refid	the id of another XMLCatalog whose contents you would like to be used for this XMLCatalog	No

7.3.14.3 XMLCatalog nested elements

dtd/entity
The dtd and entity elements used to specify XMLCatalogs are identical in their structure

Attribute	Description	Required
publicId	The public identifier used when defining a dtd or entity, e.g. "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"	Yes
location	The location of the local replacement to be used for the public identifier specified. This may be specified as a file name, resource name found on the classpath, or a URL. Relative paths will be resolved according to the base, which by default is the Ant project basedir.	Yes

classpath
The classpath to use for entity resolution. The nested <classpath> is a path-like structure.

catalogpath
The nested catalogpath element is a path-like structure listing catalog files to search. All files in this path are assumed to be OASIS catalog files, in either plain text format or XML format. Entries specifying nonexistent files will be ignored. If the resolver library from xml-commons is not available in the classpath, all catalogpaths will be ignored and a warning will be logged.

Examples
Set up an XMLCatalog with a single dtd referenced locally in a user's home directory:

```
    <xmlcatalog>
        <dtd
            publicId="-//OASIS//DTD DocBook XML V4.1.2//EN"
            location="/home/dion/downloads/docbook/docbookx.dtd"/>
    </xmlcatalog>
```

Set up an XMLCatalog with a multiple dtds to be found either in the filesystem (relative to the Ant project basedir) or in the classpath:

```
    <xmlcatalog id="commonDTDs">
        <dtd
            publicId="-//OASIS//DTD DocBook XML V4.1.2//EN"
            location="docbook/docbookx.dtd"/>
        <dtd
            publicId="-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
            location="web-app_2_2.dtd"/>
    </xmlcatalog>
```

Set up an XMLCatalog with a combination of DTDs and entities as well as a nested XMLCatalog and external catalog files in both formats:

```
    <xmlcatalog id="allcatalogs">
        <dtd
            publicId="-//ArielPartners//DTD XML Article V1.0//EN"
            location="com/arielpartners/knowledgebase/dtd/article.dtd"/>
        <entity
            publicId="LargeLogo"
            location="com/arielpartners/images/ariel-logo-large.gif"/>
        <xmlcatalog refid="commonDTDs"/>
            <catalogpath>
                <pathelement location="/etc/sgml/catalog"/>
                <fileset
                    dir="/anetwork/drive"
                    includes="**/catalog"/>
                <fileset
                    dir="/my/catalogs"
                    includes="**/catalog.xml"/>
            </catalogpath>
        </xmlcatalog>
    </xmlcatalog>
```

To reference the above XMLCatalog in an xslt task:

```
    <xslt basedir="${source.doc}"
           destdir="${dest.xdocs}"
           extension=".xml"
           style="${source.xsl.converter.docbook}"
           includes="**/*.xml"
           force="true">
        <xmlcatalog refid="allcatalogs"/>
    </xslt>
```

7.3.15 ZipFileSet

A <zipfileset> is a special form of a <fileset> which can behave in 2 different ways :

- When the src attribute is used, the zipfileset is populated with zip entries found in the file src.
- When the dir attribute is used, the zipfileset is populated with filesystem files found under dir.

<zipfileset> supports all attributes of <fileset> in addition to those listed below.

Since Ant 1.6, a zipfileset can be defined with the id attribute and referred to with the refid attribute.

Parameters

Attribute	Description	Required

prefix	all files in the fileset are prefixed with that path in the archive.	No
fullpath	the file described by the fileset is placed at that exact location in the archive.	No
src	may be used in place of the dir attribute to specify a zip file whose contents will be extracted and included in the archive.	No
filemode	A 3 digit octal string, specify the user, group and other modes in the standard Unix fashion. Only applies to plain files. Default is 644. since Ant 1.5.2.	No
dirmode	A 3 digit octal string, specify the user, group and other modes in the standard Unix fashion. Only applies to directories. Default is 755. since Ant 1.5.2.	No

The fullpath attribute can only be set for filesets that represent a single file. The prefix and fullpath attributes cannot both be set on the same fileset.
When using the src attribute, include and exclude patterns may be used to specify a subset of the zip file for inclusion in the archive as with the dir attribute.

Examples

```
<zip destfile="${dist}/manual.zip">
  <zipfileset dir="htdocs/manual" prefix="docs/user-guide"/>
  <zipfileset dir="." includes="ChangeLog27.txt" fullpath="docs/ChangeLog.txt"/>
  <zipfileset src="examples.zip" includes="**/*.html" prefix="docs/examples"/>
</zip>
```

zips all files in the htdocs/manual directory into the docs/user-guide directory in the archive, adds the file ChangeLog27.txt in the current directory as docs/ChangeLog.txt, and includes all the html files in examples.zip under docs/examples. The archive might end up containing the files:

```
docs/user-guide/html/index.html
docs/ChangeLog.txt
docs/examples/index.html
```

7.4 Optional Types

7.4.1 ClassFileSet

A classfileset is a specialised type of fileset which, given a set of "root" classes, will include all of the class files upon which the root classes depend. This is typically used to create a jar with all of the required classes for a particular application.

classfilesets are typically used by reference. They are declared with an "id" value and this is then used as a reference where a normal fileset is expected.

This type requires the jakarta-BCEL library.

Attributes

The class fileset support the following attributes in addition to those supported by the standard fileset:

Attribute	Description	Required
rootclass	A single root class name	No

Nested Elements

Root

When more than one root class is required, multiple nested <root> elements may be used

Attribute	Description	Required

classname	The fully qualified name of the root class	Yes

RootFileSet

A root fileset is used to add a set of root classes from a fileset. In this case the entries in the fileset are expected to be Java class files. The name of the Java class is determined by the relative location of the classfile in the fileset. So, the file org/apache/tools/ant/Project.class corresponds to the Java class org.apache.tools.ant.Project.

Examples

```
<classfileset id="reqdClasses" dir="${classes.dir}">
  <root classname="org.apache.tools.ant.Project" />
</classfileset>
```

This example creates a fileset containing all the class files upon which the org.apache.tools.ant.Project class depends. This fileset could then be used to create a jar.

```
<jar destfile="minimal.jar">
  <fileset refid="reqdClasses"/>
</jar>
<classfileset id="reqdClasses" dir="${classes.dir}">
  <rootfileset dir="${classes.dir}" includes="org/apache/tools/ant/Project*.class"/>
</classfileset>
```

This example constructs the classfileset using all the class with names starting with Project in the org.apache.tools.ant package

7.4.2 Extension

Utility type that represents either an available "Optional Package" (formerly known as "Standard Extension") as described in the manifest of a JAR file, or the requirement for such an optional package.

Note that this type works with extensions as defined by the "Optional Package" specification. For more information about optional packages, see the document Optional Package Versioning in the documentation bundle for your Java2 Standard Edition package, in file guide/extensions/versioning.html or online at http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html.

Attributes

The extension type supports the following attributes:

Attribute	Description	Required
extensionName	The name of extension	yes
specificationVersion	The version of extension specification (Must be in dewey decimal aka dotted decimal notation. 3.2.4)	no
specificationVendor	The specification vendor	no
implementationVersion	The version of extension implementation (Must be in dewey decimal aka dotted decimal notation. 3.2.4)	no
implementationVendor	The implementation vendor	no
implementationVendorId	The implementation vendor ID	no
implementationURL	The url from which to retrieve extension.	no

Examples

```
<extension id="e1"
    extensionName="MyExtensions"
    specificationVersion="1.0"
    specificationVendor="Peter Donald"
    implementationVendorID="vv"
```

```
    implementationVendor="Apache"
    implementationVersion="2.0"
    implementationURL="http://somewhere.com/myExt.jar"/>
```
Fully specifiec extension object.

```
<extension id="e1"
    extensionName="MyExtensions"
    specificationVersion="1.0"
    specificationVendor="Peter Donald"/>
```
Extension object that just species the specification details.

7.4.3 ExtensionSet

Utility type that represents a set of Extensions.

Note that this type works with extensions as defined by the "Optional Package" specification. For more information about optional packages, see the document Optional Package Versioning in the documentation bundle for your Java2 Standard Edition package, in file guide/extensions/versioning.html or online at http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html.

Nested Elements

extension

Extension object to add to set.

fileset

FileSets all files contained contained within set that are jars and implement an extension are added to extension set.

LibFileSet

All files contained contained within set that are jars and implement an extension are added to extension set. However the extension information may be modified by attributes of libfileset

Examples

```
<extension id="e1"
    extensionName="MyExtensions"
    specificationVersion="1.0"
    specificationVendor="Peter Donald"
    implementationVendorID="vv"
    implementationVendor="Apache"
    implementationVersion="2.0"
    implementationURL="http://somewhere.com/myExt.jar"/>

<libfileset id="lfs"
               includeUrl="true"
               includeImpl="false"
               dir="tools/lib">
  <include name="*.jar"/>
</libfileset>

<extensionSet id="exts">
  <libfileset dir="lib">
    <include name="*.jar"/>
  </libfileset>
  <libfileset refid="lfs"/>
  <extension refid="e1"/>
```

```
</extensionSet>
```

7.5 XML Namespace Support

Ant 1.6 introduces support for XML namespaces.

7.5.1 History

All releases of Ant prior to Ant 1.6 do not support XML namespaces. No support basically implies two things here:

- Element names correspond to the "qname" of the tags, which is usually the same as the local name. But if the build file writer uses colons in names of defined tasks/types, those become part of the element name. Turning on namespace support gives colon-separated prefixes in tag names a special meaning, and thus build files using colons in user-defined tasks and types will break.
- Attributes with the names 'xmlns' and 'xmlns:<prefix>' are not treated specially, which means that custom tasks and types have actually been able to use such attributes as parameter names. Again, such tasks/types are going to break when namespace support is enabled on the parser.

Use of colons in element names has been discouraged in the past IIRC, and using any attribute starting with "xml" is actually strongly discouraged by the XML spec to reserve such names for future use.

7.5.2 Motivation

In build files using a lot of custom and third-party tasks, it is easy to get into name conflicts. When individual types are defined, the build file writer can do some name-spacing manually (for example, using "tomcat-deploy" instead of just "deploy"). But when defining whole libraries of types using the <typedef> 'resource' attribute, the build file writer has no chance to override or even prefix the names supplied by the library.

7.5.3 Assigning Namespaces

Adding a 'prefix' attribute to <typedef> might have been enough, but XML already has a well-known method for name-spacing. Thus, instead of adding a 'prefix' attribute, the <typedef> and <taskdef> tasks get a 'uri' attribute, which stores the URI of the XML namespace with which the type should be associated:

```
<typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
<my:task xmlns:my="http://example.org/tasks">
   ...
</my:task>
```

As the above example demonstrates, the namespace URI needs to be specified at least twice: one time as the value of the 'uri' attribute, and another time to actually map the namespace to occurrences of elements from that namespace, by using the 'xmlns' attribute. This mapping can happen at any level in the build file:

```
<project name="test" xmlns:my="http://example.org/tasks">
  <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
  <my:task>
    ...
  </my:task>
</project>
```

Use of a namespace prefix is of course optional. Therefore the example could also look like this:

```
<project name="test">
  <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
  <task xmlns="http://example.org/tasks">
    ...
  </task>
</project>
```

Here, the namespace is set as the default namespace for the <task> element and all its descendants.

7.5.4 Default namespace

The default namespace used by Ant is "antlib:org.apache.tools.ant".

```
<typedef resource="org/example/tasks.properties"
uri="antlib:org.apache.tools.ant"/>
<task>
      ....
</task>
```

7.5.5 Namespaces and Nested Elements

Almost always in Ant 1.6, elements nested inside a namespaced element have the same namespace as their parent. So if 'task' in the example above allowed a nested 'config' element, the build file snippet would look like this:

```
 <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
 <my:task xmlns:my="http://example.org/tasks">
   <my:config a="foo" b="bar"/>
   ...
 </my:task>
```

If the element allows or requires a lot of nested elements, the prefix needs to be used for every nested element. Making the namespace the default can reduce the verbosity of the script:

```
 <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
          <task xmlns="http://example.org/tasks">
          <config a="foo" b="bar"/>
   ...
          </task>
```

7.5.6 Namespaces and Attributes

Attributes are only used to configure the element they belong to if:

- they have no namespace (note that the default namespace does *not* apply to attributes)
- they are in the same namespace as the element they belong to

Other attributes are simply ignored.
This means that both:

```
 <my:task xmlns:my="http://example.org/tasks">
   <my:config a="foo" b="bar"/>
   ...
 </my:task>
```

and

```
 <my:task xmlns:my="http://example.org/tasks">
   <my:config my:a="foo" my:b="bar"/>
   ...
 </my:task>
```

result in the parameters "a" and "b" being used as parameters to configure the nested "config" element.

It also means that you can use attributes from other namespaces to markup the build file with extra meta data, such as RDF and XML-Schema (whether that's a good thing or not). The same is not true for elements from unknown namespaces, which result in a error.

7.5.7 Mixing Elements from Different Namespaces

Now comes the difficult part: elements from different namespaces can be woven together under certain circumstances. This has a lot to do with the Ant 1.6 add type introspection rules: Ant types and tasks are now free to accept arbritrary named types as nested elements, as long as the concrete type implements the interface expected by the task/type. The most obvioius example for this is the <condition> task, which supports various nested conditions, all of which extend the interface Condition. To integrate a custom condition in Ant, you can now simply <typedef> the condition, and then use it anywhere develop.html#nestedtypwhere conditions are allowed (assuming the containing element has a generic add(Condition) or addConfigured(Configured)method):

```
 <typedef resource="org/example/conditions.properties"
uri="http://example.org/conditions"/>
 <condition property="prop" xmlns="http://example.org/conditions">
   <and>
     <available file="bla.txt"/>
     <my:condition a="foo"/>
   </and>
 </condition>
```

In Ant 1.6, this feature cannot be used as much as we'd all like to: a lot of code has not yet been adapted to the new introspection rules, and elements like the builtin Ant conditions and selectors are not really types in 1.6. This is expected to change in Ant 1.7.

7.5.8 Namespaces and Antlib

The new AntLib feature is also very much integrated with the namespace support in Ant 1.6. Basically, you can "import" Antlibs simply by using a special scheme for the namespace URI: the antlib scheme, which expects the package name in which a special antlib.xml file is located.

7.6 Antlib

7.6.1 Description

An antlib file is an xml file with a root element of "antlib". Antlib's elements are ant definition tasks - like Typedef and Taskdef, or any ant task that extends org.apache.tools.ant.taskdefs.AntlibDefinition.

A group of tasks and types may be defined together in an antlib file. For example the file sample.xml contains the following:

```
<?xml version="1.0"?>
<antlib>
   <typedef name="if" classname="org.acme.ant.If"/>
   <typedef name="scriptpathmapper"
            classname="org.acme.ant.ScriptPathMapper"
            onerror="ignore"/>
</antlib>
```

It defines two types or tasks, if and scriptpathmapper. This antlib file may be used in a build script as follows:

```
<typedef file="sample.xml"/>
```

The other attributes of <typedef> may be used as well. For example, assuming that the sample.xml is in a jar file sample.jar also containing the classes, the following build fragment will define the if and scriptpathmapper tasks/types and place them in the namespace uri samples:/acme.org.

```
<typedef resource="org/acme/ant/sample.xml"
         uri="samples:/acme.org"/>
```

The definitions may then be used as follows:

```
<sample:if valuetrue="${props}" xmlns:sample="samples:/acme.org">
   <sample:scriptpathmapper language="beanshell">
      some bean shell
   </sample:scriptpathmapper>
</sample:if>
```

7.6.2 Antlib namespace

The name space URIs with the pattern antlib:java package are given special treatment.

When ant encounters a element with a namespace URI with this pattern, it will check to see if there is a resource of the name antlib.xml in the package directory in the default classpath.

For example, assuming that the file antcontrib.jar has been placed in the directory ${ant.home}/lib and it contains the resource net/sf/antcontrib/antlib.xml which has all antcontrib's definitions defined, the following build file will automatically load the antcontrib definitions at location HERE:

```
<project default="deletetest" xmlns:antcontrib="antlib:net.sf.antcontrib">
   <macrodef name="showdir">
      <attribute name="dir"/>
      <sequential>
         <antcontrib:shellscript shell="bash">  <!-- HERE -->
            ls -Rl @{dir}
         </antcontrib:shellscript>
      </sequential>
   </macrodef>

   <target name="deletetest">
      <delete dir="a" quiet="yes"/>
      <mkdir dir="a/b"/>
      <touch file="a/a.txt"/>
      <touch file="a/b/b.txt"/>
      <delete>
         <fileset dir="a"/>
      </delete>
      <showdir dir="a"/>
   </target>
</project>
```

The requirement that the resource is in the default classpath may be removed in future versions of Ant.

7.6.3 Current namespace

Definitions defined in antlibs may be used in antlibs. However the namespace that definitions are placed in are dependent on the <typedef> that uses the antlib. To deal with this problem, the definitions are placed in the namepace URI ant:current for the duration of the antlib execution. For example the following antlib defines the task <if>, the type <isallowed> and a macro <ifallowed> that makes use of the task and type:

```
<antlib xmlns:current="ant:current">
   <taskdef name="if" classname="org.acme.ant.If"/>
   <typedef name="isallowed" classname="org.acme.ant.Isallowed"/>
   <macrodef name="ifallowed">
      <attribute name="action"/>
      <element name="do"/>
      <sequential>
```

```
         <current:if>
            <current:isallowed test="@{action}"/>
            <current:then>
               <current:do/>
            </current:then>
         </current:if>
      </sequential>
   </macrodef>
</antlib>
```

7.6.4 Other examples and comments

Antlibs may make use of other antlibs.

As the names defined in the antlib are in the namespace uri as specified by the calling <typedef> or by automatic element resolution, one may reuse names from core ant types and tasks, provided the caller uses a namespace uri. For example, the following antlib may be used to define defaults for various tasks:

```
<antlib xmlns:antcontrib="antlib:net.sf.antcontrib">
   <presetdef name="javac">
      <javac deprecation="${deprecation}"
             debug="${debug}"/>
   </presetdef>
   <presetdef name="delete">
      <delete quiet="yes"/>
   </presetdef>
   <presetdef name="shellscript">
      <antcontrib:shellscript shell="bash"/>
   </presetdef>
</antlib>
```

This may be used as follows:

```
<project xmlns:local="localpresets">
   <typedef file="localpresets.xml" uri="localpresets"/>
   <local:shellscript>
      echo "hello world"
   </local:shellscript>
</project>
```

7.7 Custom Components

7.7.1 Overview

Custom components are conditions, selectors, filters and other objects that are defined outside ant core.
In Ant 1.6 custom conditions, selectors and filters has been overhauled.

It is now possible to define custom conditions, selectors and filters that behave like Ant Core components. This is achieved by allowing datatypes defined in build scripts to be used as custom components if the class of the datatype is compatible, or has been adapted by an adapter class.

The old methods of defining custom components are still supported.

7.7.2 Definition and use

A custom component is a normal Java class that implements a particular interface or extends a particular class, or has been adapted to the interface or class.

It is exactly like writing a custom task. One defines attributes and nested elements by writing setter methods and add methods.

After the class has been written, it is added to the ant system by using <typedef>.

7.7.3 Custom Conditions

Custom conditions are datatypes that implement org.apache.tools.ant.taskdefs.condition.Condition. For example a custom condition that returns true if a string is all upper case could be written as:

```
package com.mydomain;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.taskdefs.condition.Condition;

public class AllUpperCaseCondition implements Condition {
    private String value;

    // The setter for the "value" attribute
    public void setValue(String value) {
        this.value = value;
    }

    // This method evaluates the condition
    public boolean eval() {
        if (value == null) {
            throw new BuildException("value attribute is not set");
        }
        return value.toUpperCase().equals(value);
   }
}
```

Adding the condition to the system is achieved as follows:

```
<typedef
    name="alluppercase"
    classname="com.mydomain.AllUpperCaseCondition"
    classpath="${mydomain.classes}"/>
```

This condition can now be used wherever a Core Ant condition is used.

```
<condition property="allupper">
   <alluppercase value="THIS IS ALL UPPER CASE"/>
</condition>
```

7.7.4 Custom Selectors

Custom selectors are datatypes that implement org.apache.tools.ant.types.selectors.FileSelector.

There is only one method required. public boolean isSelected(File basedir, String filename, File file). It returns true or false depending on whether the given file should be selected or not.

An example of a custom selection that selects filenames ending in ".java" would be:

```
package com.mydomain;
import java.io.File;
import org.apache.tools.ant.types.selectors.FileSelector;
```

```
public class JavaSelector implements FileSelector {
    public boolean isSelected(File b, String filename, File f) {
       return filename.toLowerCase().endsWith(".java");
    }
}
```

Adding the selector to the system is achieved as follows:

```
<typedef
    name="javaselector"
    classname="com.mydomain.JavaSelector"
    classpath="${mydomain.classes}"/>
```

This selector can now be used wherever a Core Ant selector is used, for example:

```
<copy todir="to">
   <fileset dir="src">
      <javaselector/>
   </fileset>
</copy>
```

One may use org.apache.tools.ant.types.selectors.BaseSelector, a convenience class that provides reasonable default behaviour. It has some predefined behaviours you can take advantage of. Any time you encounter a problem when setting attributes or adding tags, you can call setError(String errmsg) and the class will know that there is a problem. Then, at the top of your isSelected() method call validate() and a BuildException will be thrown with the contents of your error message. The validate() method also gives you a last chance to check your settings for consistency because it calls verifySettings(). Override this method and call setError() within it if you detect any problems in how your selector is set up.

To write custom selector containers one should extend org.apache.tools.ant.types.selectors.BaseSelectorContainer. Implement the public boolean isSelected(File baseDir, String filename, File file) method to do the right thing. Chances are you'll want to iterate over the selectors under you, so use selectorElements() to get an iterator that will do that.

For example to create a selector container that will select files if a certain number of contained selectors select, one could write a selector as follows:

```
public class MatchNumberSelectors extends BaseSelectorContainer {
    private int number = -1;
    public void setNumber(int number) {
        this.number = number;
    }
    public void verifySettings() {
        if (number < 0) {
           throw new BuildException("Number attribute should be set");
        }
    }
    public boolean isSelected(File baseDir, String filename, File file) {
        validate();
        int numberSelected = 0;
        for (Enumeration e = selectorElements(); e.hasNextElement();) {
            FileSelector s = (FileSelector) e.nextElement();
            if (s.isSelected(baseDir, filename, file)) {
                numberSelected++;
            }
```

```
        }
        return numberSelected == number;
    }
}
```

To define and use this selector one could do:

```
<typedef name="numberselected"
         classname="com.mydomain.MatchNumberSelectors"/>
...
<fileset dir="${src.path}">
   <numberselected number="2">
      <contains text="script" casesensitive="no"/>
      <size value="4" units="Ki" when="more"/>
      <javaselector/>
   </numberselected>
</fileset>
```

The custom selector

The custom selector was the pre ant 1.6 way of defining custom selectors. This method is still supported for backward compatibility.

You can write your own selectors and use them within the selector containers by specifying them within the <custom> tag.

To create a new Custom Selector, you have to create a class that implements org.apache.tools.ant.types.selectors.ExtendFileSelector. The easiest way to do that is through the convenience base class org.apache.tools.ant.types.selectors.BaseExtendSelector, which provides all of the methods for supporting <param> tags. First, override the isSelected() method, and optionally the verifySettings() method. If your custom selector requires parameters to be set, you can also override the setParameters() method and interpret the parameters that are passed in any way you like. Several of the core selectors demonstrate how to do that because they can also be used as custom selectors.

Once that is written, you include it in your build file by using the <custom> tag.

Attribute	Description	Required
classname	The name of your class that implements org.apache.tools.ant.types.selectors.FileSelector.	Yes
classpath	The classpath to use in order to load the custom selector class. If neither this classpath nor the classpathref are specified, the class will be loaded from the classpath that Ant uses.	No
classpathref	A reference to a classpath previously defined. If neither this reference nor the classpath above are specified, the class will be loaded from the classpath that Ant uses.	No

Here is how you use <custom> to use your class as a selector:

```
<fileset dir="${mydir}" includes="**/*">
    <custom classname="com.mydomain.MySelector">
        <param name="myattribute" value="myvalue"/>
    </custom>
</fileset>
```

The core selectors that can also be used as custom selectors are

- Contains Selector with classname org.apache.tools.ant.types.selectors.ContainsSelector

- Date Selector with classname org.apache.tools.ant.types.selectors.DateSelector
- Depth Selector with classname org.apache.tools.ant.types.selectors.DepthSelector
- Filename Selector with classname org.apache.tools.ant.types.selectors.FilenameSelector
- Size Selector with classname org.apache.tools.ant.types.selectors.SizeSelector

Here is the example from the Depth Selector section rewritten to use the selector through <custom>.

```
<fileset dir="${doc.path}" includes="**/*">
    <custom classname="org.apache.tools.ant.types.selectors.DepthSelector">
        <param name="max" value="1"/>
    </custom>
</fileset>
```

Selects all files in the base directory and one directory below that.

7.7.5 Custom Filter Readers

Custom filter readers selectors are datatypes that implement org.apache.tools.ant.types.filters.ChainableReader.

There is only one method required. Reader chain(Reader reader). This returns a reader that filters input from the specified reader.

For example a filterreader that removes every second character could be:

```
public class RemoveOddCharacters implements ChainableReader {
   public Reader chain(Reader reader) {
      return new BaseFilterReader(reader) {
          int count = 0;
          public int read() throws IOException {
              while (true) {
                int c = in.read();
                if (c == -1) {
                    return c;
                }
                count++;
                if ((count % 2) == 1) {
                    return c;
                }
              }
          }
      }
   }
}
```

For line oriented filters it may be easier to extend ChainableFilterReader an inner class of org.apache.tools.ant.filters.TokenFilter.

For example a filter that appends the line number could be

```
public class AddLineNumber extends ChainableReaderFilter {
   private void lineNumber = 0;
   public String filter(String string) {
      lineNumber++;
      return "" + lineNumber + "\t" + string;
   }
}
```

8 Listeners & Loggers

8.1 Overview

Ant has two related features to allow the build process to be monitored: listeners and loggers.

8.1.1 Listeners

A listener is alerted of the following events:

- build started
- build finished
- target started
- target finished
- task started
- task finished
- message logged

8.1.2 Loggers

Loggers extend the capabilities of listeners and add the following features:

- Receives a handle to the standard output and error print streams and therefore can log information to the console or the -logfile specified file.
- Logging level (-quiet, -verbose, -debug) aware
- Emacs-mode aware

8.2 Built-in Listeners/Loggers

Classname	Description	Type
org.apache.tools.ant.DefaultLogger	The logger used implicitly unless overridden with the -logger command-line switch.	BuildLogger
org.apache.tools.ant.NoBannerLogger	This logger omits output of empty target output.	BuildLogger
org.apache.tools.ant.listener.MailLogger	Extends DefaultLogger such that output is still generated the same, and when the build is finished an e-mail can be sent.	BuildLogger
org.apache.tools.ant.listener.AnsiColorLogger	Colorifies the build output.	BuildLogger
org.apache.tools.ant.listener.Log4jListener	Passes events to Log4j for highly customizable logging.	BuildListener
org.apache.tools.ant.XmlLogger	Writes the build information to an XML file.	BuildLogger

8.2.1 DefaultLogger

Simply run Ant normally, or:

```
ant -logger org.apache.tools.ant.DefaultLogger
```

8.2.2 NoBannerLogger

Removes output of empty target output.

```
ant -logger org.apache.tools.ant.NoBannerLogger
```

8.2.3 MailLogger

The MailLogger captures all output logged through DefaultLogger (standard Ant output) and will send success and failure messages to unique e-mail lists, with control for turning off success or failure messages individually.

Properties controlling the operation of MailLogger:

Property	Description	Required
MailLogger.mailhost	Mail server to use	No, default "localhost"
MailLogger.port	SMTP Port for the Mail server	No, default "25"
MailLogger.user	user name for SMTP auth	Yes, if SMTP auth is required on your SMTP server, the email message will be then sent using Mime and requires JavaMail
MailLogger.password	password for SMTP auth	Yes, if SMTP auth is required on your SMTP server the email message will be then sent using Mime and requires JavaMail
MailLogger.ssl	on or true if ssl is needed This feature requires JavaMail	no
MailLogger.from	Mail "from" address	Yes, if mail needs to be sent
MailLogger.replyto	Mail "replyto" address(es), comma-separated	No
MailLogger.failure.notify	Send build failure e-mails?	No, default "true"
MailLogger.success.notify	Send build success e-mails?	No, default "true"
MailLogger.failure.to	Address(es) to send failure messages to, comma-separated	Yes, if failure mail is to be sent
MailLogger.success.to	Address(es) to send success messages to, comma-separated	Yes, if success mail is to be sent
MailLogger.failure.subject	Subject of failed build	No, default "Build Failure"
MailLogger.success.subject	Subject of successful build	No, default "Build Success"
MailLogger.properties.file	Filename of properties file that will override other values.	No

```
ant -logger org.apache.tools.ant.listener.MailLogger
```

8.2.4 AnsiColorLogger

The AnsiColorLogger adds color to the standard Ant output by prefixing and suffixing ANSI color code escape sequences to it. It is just an extension of DefaultLogger and hence provides all features that DefaultLogger does.

AnsiColorLogger differentiates the output by assigning different colors depending upon the type of the message.

If used with the -logfile option, the output file will contain all the necessary escape codes to display the text in colorized mode when displayed in the console using applications like cat, more, etc.

This is designed to work on terminals that support ANSI color codes. It works on XTerm, ETerm, Win9x Console (with ANSI.SYS loaded.), etc.

NOTE: It doesn't work on WinNT even when a COMMAND.COM console loaded with ANSI.SYS is used.

If the user wishes to override the default colors with custom ones, a file containing zero or more of the custom color key-value pairs must be created. The recognized keys and their default values are shown below:

```
AnsiColorLogger.ERROR_COLOR=2;31
AnsiColorLogger.WARNING_COLOR=2;35
AnsiColorLogger.INFO_COLOR=2;36
AnsiColorLogger.VERBOSE_COLOR=2;32
```

```
AnsiColorLogger.DEBUG_COLOR=2;34
```

Each key takes as value a color combination defined as **Attribute;Foreground;Background**. In the above example, background value has not been used.

This file must be specfied as the value of a system variable named ant.logger.defaults and passed as an argument using the -D option to the java command that invokes the Ant application. An easy way to achieve this is to add -Dant.logger.defaults= /path/to/your/file to the ANT_OPTS environment variable. Ant's launching script recognizes this flag and will pass it to the java command appropriately.

Format:

```
AnsiColorLogger.*=Attribute;Foreground;Background

Attribute is one of the following:
0 -> Reset All Attributes (return to normal mode)
1 -> Bright (Usually turns on BOLD)
2 -> Dim
3 -> Underline
5 -> link
7 -> Reverse
8 -> Hidden

Foreground is one of the following:
30 -> Black
31 -> Red
32 -> Green
33 -> Yellow
34 -> Blue
35 -> Magenta
36 -> Cyan
37 -> White

Background is one of the following:
40 -> Black
41 -> Red
42 -> Green
43 -> Yellow
44 -> Blue
45 -> Magenta
46 -> Cyan
47 -> White
ant -logger org.apache.tools.ant.listener.AnsiColorLogger
```

8.2.5 Log4jListener

Passes build events to Log4j, using the full classname's of the generator of each build event as the category:

- build started / build finished - org.apache.tools.ant.Project
- target started / target finished - org.apache.tools.ant.Target
- task started / task finished - the fully qualified classname of the task
- message logged - the classname of one of the above, so if a task logs a message, its classname is the category used, and so on.

All start events are logged as INFO. Finish events are either logged as INFO or ERROR depending on whether the build failed during that stage. Message events are logged according to their Ant logging level, mapping directly to a corresponding Log4j level.

ant -listener org.apache.tools.ant.listener.Log4jListener

8.2.6 XmlLogger

Writes all build information out to an XML file named log.xml, or the value of the XmlLogger.file property if present, when used as a listener. When used as a logger, it writes all output to either the console or to the value of -logfile. Whether used as a listener or logger, the output is not generated until the build is complete, as it buffers the information in order to provide timing information for task, targets, and the project.

By default the XML file creates a reference to an XSLT file "log.xsl" in the current directory; look in ANT_HOME/etc for one of these. You can set the property ant.XmlLogger.stylesheet.uri to provide a uri to a style sheet. this can be a relative or absolute file path, or an http URL. If you set the property to the empty string, "", no XSLT transform is declared at all.

```
ant -listener org.apache.tools.ant.XmlLogger
ant -logger org.apache.tools.ant.XmlLogger -verbose -logfile build_log.xml
```

8.3 Writing your own

See the Build Events section for developers.
Notes:

- A listener or logger should not write to standard output or error - Ant captures these internally and may cause an infinite loop.

9 Ant in Anger (Using Apache Ant in a Production Development System)

By Steve Loughran
Last updated 2002-11-09

9.1 Introduction

Apache Ant can be an invaluable tool in a team development process -or it can be yet another source of problems in that ongoing crises we call development . This document contains some strategies and tactics for making the most of Ant. It is moderately frivolous in places, and lacks almost any actual examples of Ant xml. The lack of examples is entirely deliberate -it keeps document maintenance costs down. Most of the concepts covered don't need the detail about XML representations, as it is processes we are concerned about, not syntax. Finally, please be aware that the comments here are only suggestions which need to be customised to meet your own needs, not strict rules about what should and should not be done.

Firstly, here are some assumptions about the projects which this document covers:

- Pretty much pure Java, maybe with some legacy cruft on the edges.
- Team efforts, usually with the petulant prima-donnas all us Java programmers become once we realise how much in demand we are.
- A fairly distributed development team -spread across locations and maybe time zones.
- Separate sub projects -from separate beans in a big enterprise application to separate enterprise applications which need to be vaguely aware of each other.
- Significant mismatch between expectations and time available to deliver. 'Last Week' is the ideal delivery date handed down from above, late next century the date coming up from below.
- Everyone is struggling to keep up with platform and tool evolution.
- Extensive use of external libraries, both open and closed source.

What that all means is that there is no time to spend getting things right, you don't have that tight control on how the rest of the team works and the development process is often more one of chaos minimisation than anything else. The role of Ant in such projects is to ensure that the build, test and deploy processes run smoothly, leaving you with all the other problems.

9.2 Core Practices

9.2.1 Clarify what you want Ant to do

Ant is not a silver bullet. It is just another rusty bullet in the armory of development tools available at your disposal. Its primary purpose is to accelerate the construction and deployment of Java projects. You could certainly extend Ant to do anything Java makes possible: it is easy to imagine writing an image processing task to help in web site deployment by shrinking and recompressing jpeg files, for example. But that would be pushing the boundary of what Ant is really intended to do -so should be considered with care.

Ant is also a great adjunct to an IDE; a way of doing all the housekeeping of deployment and for clean, automated builds. But a good modern IDE is a productivity tool in its own right -one you should consider keeping using. Ant just lets you give the teams somewhat more freedom in IDE choice -"you can use whatever you want in development, but Ant for the deployment builds" Now that many modern open source and commercial IDEs include Ant support (including jEdit, Forte, Eclipse and IDEA), developers can use a great IDE, with Ant providing a rigorous and portable build process integrated into the tool.

9.2.2 Define standard targets

When you have multiple sub projects, define a standard set of targets. Projects with a split between interface and implementation jar files could consider **impl** and **intf** targets -with separate **debug-impl** and **debug-intf** targets for the debug version. And of course, the ubiquitous **clean** target.

With standard target names, it is easy to build encompassing Ant build files which just hand off the work to the classes below using the <ant> task. For example. the clean target could be handed down to the intf and impl subdirectories from a parent directory

```
<target name="clean" depends="clean-intf, clean-impl">
</target>
<target name="clean-intf" >
    <ant dir="intf" target="clean" />
</target>
<target name="clean-impl">
    <ant dir="impl" target="clean" />
</target>
```

If you give targets a description tag, then calling ant -projecthelp will list all tasks with their description as 'main targets', and all tasks without a description as subtargets. Describing all your entry points is therefore very useful, even before a project becomes big and complicated.

9.2.3 Extend Ant through new tasks

If Ant does not do what you want, you can use the exec and java tasks or inline scripting to extend it. In a project with many build.xml files, you soon find that having a single central place for implementing the functionality keeps maintenance overhead down. Implementing task extensions through Java code seems extra effort at first, but gives extra benefits:-

- Cross platform support can be added later without changing any build.xml files
- The code can be submitted to the Ant project itself, for other people to use and maintain
- It keeps the build files simpler

In a way, it is it this decoupling of functionality, "the tasks", from the declaration of use, "the build file", that has helped Ant succeed. If you have to get something complex done in Make or an IDE, you have a hairy makefile that everyone is scared of, or an IDE configuration that is invariably very brittle. But an Ant task is reusable and shareable among all Ant users. Many of the core and optional tasks in Ant today, tasks you do or will come to depend on, were written by people trying to solve their own pressing problems.

9.2.4 Embrace Automated Testing

(Alternatively "recriminate early, recriminate often")

Ant lets you call JUnit tasks, which unit test the code your team has written. Automated testing may seem like extra work at first, but JUnit makes writing unit tests so easy that you have almost no reason not to. Invest the time in learning how to use JUnit, write the test cases, and integrate them in a 'test' target from Ant so that your daily or hourly team build can have the tests applied automatically. One of the free to download chapters of Java Development with Ant shows you how to use JUnit from inside Ant.

Once you add a way to fetch code from the SCM system, either as an Ant task, in some shell script or batch file or via some continuous integration tool. the integration test code can be a pure Ant task run on any box dedicated to the task. This is ideal for verifying that the build and unit tests work on different targets from the usual development machines. For example, a Win95/Java1.1 combination could be used even though no developer would willingly use that configuration given the choice.

System tests are harder to automate than unit tests, but if you can write java code to stress large portions of the system -even if the code can not run as JUnit tasks- then the java task can be used to invoke them. It is best to specify that you want a new JVM for these tests, so that a significant crash does not break the full build. The Junit extensions such as HttpUnit for web pages, and Cactus for J2EE and servlet testing help to expand the testing framework. To test properly you will still need to invest a lot of effort in getting these to work with

your project, and deriving great unit, system and regression tests -but your customers will love you for shipping software that works.

9.2.5 Learn to Use and love the add-ons to Ant

The Ant distribution is not the limit of the Ant universe, it is only the beginning. Look at the External Tools and Tasks page for an up to date list. Here are some of them:

- Checkstyle
  This tool audits your code and generates HTML reports of wherever any style rule gets broken. Nobody can hide from the code police now! tip: start using this early, so the corrections are less.
- Ant-contrib
  This sourceforge project contains helper tasks that are kept separate from core Ant for ideological purity; the foreach and trycatch tasks in particular. These give you iteration and extra error handling. Also on the site is the <cc> task suite, that compile and link native code on a variety of platforms.
- XDoclet XDoclet adds attributed oriented programming to Java. By adding javadoc tags to your code you can have XDoclet automatically generate web.xml descriptors, taglib descriptors, EJB interfaces, JMX interface classes, Castor XML/SQL bindings, and many more. The key here is that all those fiddly little XML files you need to create, and those interfaces EJB and JMX requires to to implement, all can be autogenerated from your Java code with a few helper attributes. This reduces errors and means you can change your code and have the rest of the app take its cue from the source. Never do EJB, JMX or webapps without it!

9.3 Cross Platform Ant

Ant is the best foundation for cross platform Java development and testing to date. But if you are not paying attention, it is possible to produce build files which only work on one platform -or indeed, one single workstation.

The common barriers to cross-platform Ant are the use of command line tools (exec tasks) which are not portable, path issues, and hard coding in the location of things.

9.3.1 Command Line apps: Exec/ Apply

The trouble with external invocation is that not all functions are found cross platform, and those that are often have different names -DOS descendants often expect .exe or .bat at the end of files. That can be bad if you explicitly include the extension in the naming of the command (don't!), good when it lets you keep the unix and DOS versions of an executable in the same bin directory of the project without name clashes arising.

Both the command line invocation tasks let you specify which platform you want the code to run on, so you could write different tasks for each platform you are targeting. Alternatively, the platform differences could be handled inside some external code which Ant calls. This can be some compiled down java in a new task, or an external script file.

9.3.2 Cross platform paths

Unix paths use forward slashes between directories and a colon to split entries. Thus `"/bin/java/lib/xerces.jar:/bin/java/lib/ant.jar"` is a path in unix. In Windows the path must use semicolon separators, colons being used to specify disk drives, and backslash separators `"c:\bin\java\lib\xerces.jar;c:\bin\java\lib\ant.jar"`.

This difference between platforms (indeed, the whole java classpath paradigm) can cause hours of fun.

Ant reduces path problems; but does not eliminate them entirely. You need to put in some effort too. The rules for handling path names are that 'DOS-like pathnames are handled', 'Unix like paths are handled'. Disk drives -'C:'- are handled on DOS-based boxes, but placing them in the build.xml file ruins all chances of portability. Relative file paths are much more portable. Semicolons work as path separators -a fact which is useful if your

Ant invocation wrapper includes a list of jars as a defined property in the command line. In the build files you may find it better to build a classpath by listing individual files (using location= attributes), or by including a fileset of *.jar in the classpath definition.

There is also the PathConvert task which can put a fully resolved path into a property. Why do that? Because then you can use that path in other ways -such as pass it as a parameter to some application you are calling, or use the replace task to patch it into a localised shell script or batch file.

Note that DOS descended file systems are case insensitive (apart from the obscure aberration of the WinNT posix subsystem run against NTFS), and that Windows pretends that all file extensions with four or more letters are also three letter extensions (try DELETE *.jav in your java directories to see a disastrous example of this).

Ant's policy on case sensitivity is whatever the underlying file system implements, and its handling of file extensions is that *.jav does not find any .java files. The Java compiler is of course case sensitive -you can not have a class 'ExampleThree' implemented in "examplethree.java".

Some tasks only work on one platform - Chmod being a classic example. These tasks usually result in just a warning message on an unsupported platform -the rest of the target's tasks will still be called. Other tasks degrade their functionality on platforms or Java versions. In particular, any task which adjusts the timestamp of files can not do so properly on Java 1.1. Tasks which can do that - Get, Touch and Unjar/Unwar/Unzip for example, degrade their functionality on Java1.1, usually resorting to the current timestamp instead.

Finally, Perl makes a good place to wrap up Java invocations cross platform, rather than batch files. It is included in most Unix distributions, and is a simple download for Win32 platforms from ActiveState. A Perl file with .pl extension, with the usual Unix path to perl on the line 1 comment and marked as executable can be run on Windows, OS/2 and Unix and hence called from Ant without issues. The perl code can be left to resolve its own platform issues. Dont forget to set the line endings of the file to the appropriate platform when you redistribute Perl code; <fixCRLF> can do that for you.

9.4 Team Development Processes

Even if each team member is allowed their choice of IDE/editor, or even OS, you need to set a baseline of functionality on each box. In particular, the JDKs and jars need to be in perfect sync. Ideally pick the latest stable Java/JDK version available on all developer/target systems and stick with it for a while. Consider assigning one person to be the contact point for all tools coming in -particularly open source tools when a new build is available on a nightly basis. Unless needed, these tools should only really be updated monthly, or when a formal release is made.

Another good tactic is to use a unified directory tree, and add on extra tools inside that tree. All references can be made relative to the tree. If team members are expected to add a directory in the project to their path, then command line tools can be included there -including those invoked by Ant exec tasks. Put everything under source code control and you have a one stop shop for getting a build/execute environment purely from CVS or your equivalent.

9.5 Deploying with Ant

One big difference between Ant and older tools such as Make is that the processes for deploying Java to remote sites are reasonably well evolved in Ant. That is because we all have to do it these days, so many people have put in the effort to make the tasks easier.

Ant can Jar, Tar or Zip files for deployment, while the War task extends the jar task for better servlet deployment. Jlink is a jar generation file which lets you merge multiple sub jars. This is ideal for a build process in which separate jars are generated by sub projects, yet the final output is a merged jar. Cab can be used on Win32 boxes to build a cab file which is useful if you still have to target IE deployment.

The ftp task lets you move stuff up to a server. Beware of putting the ftp password in the build file -a property file with tight access control is slightly better. The FixCRLF task is often a useful interim step if you need to ensure that files have Unix file extensions before upload. A WebDav task has long been discussed, which would provide a more secure upload to web servers, but it is still in the todo list. Rumour has it that there is such a task in the jakarta-slide libraries. With MacOS X, Linux and Windows XP all supporting WebDAV file systems, you may even be able to use <copy> to deploy though a firewall.

EJB deployment is aided by the ejb tasks, while the <serverdeploy> suite can deploy to multiple servers. The popularity of Ant has encouraged vendors to produce their own deployment tasks which they redistribute with their servers. For example, the Tomcat4.1 installation includes tasks to deploy, undeploy and reload web applications.

Finally, there are of course the fallbacks of just copying files to a destination using Copy and Copydir , or just sending them to a person or process using Mail or the attachment aware MimeMail. In one project our team even used Ant to build CD images through a build followed by a long set of Copy tasks, which worked surprisingly well, certainly easier than when we mailed them to the free email service on myrealbox.com, then pulled them down from the far end's web browser, which we were running over WinNT remote desktop connection, that being tunneled through SSH.

9.6 Directory Structures

How you structure your directory tree is very dependent upon the project. Here are some directory layout patterns which can be used as starting points. All the jakarta projects follow a roughly similar style, which makes it easy to navigate around one form one project to another, and easy to clean up when desired.

9.6.1 Simple Project

The project contains sub directories

bin	common binaries, scripts -put this on the path.
build	This is the tree for building; Ant creates it and can empty it in the 'clean' project.
dist	Distribution outputs go in here; the directory is created in Ant and clean empties it out
doc	Hand crafted documentation
lib	Imported Java libraries go in to this directory
src	source goes in under this tree in a heirarchy which matches the package names. The dependency rules of <javac> requires this.

The bin, lib, doc and src directories should be under source code control. Slight variations include an extra tree of content to be included in the distribution jars -inf files, images, etc. These can go under source too, with a metadata directory for web.xml and similar manifests, and a web folder for web content -JSP, html, images and so on. Keeping the content in this folder (or sub heirarchy) together makes it easier to test links before deployment. The actual production of a deployment image -such as a war file- can be left to the appropriate Ant task: there is no need to completely model your source tree upon the deployment heirarchy.

Javadoc output can be directed to a doc/ folder beneath build/, or to doc/javadoc.

9.6.2 Interface and Implementation split

If the interface is split from the implementation code then this can be supported with minor changes just by having a separate build path for the interface directory -or better still just in the jar construction: one jar for interface and one jar for implementation.

9.6.3 Loosely Coupled Sub Projects

In the loosely coupled approach multiple projects can have their own copy of the tree, with their own source code access rights. One difference to consider is only having one instance of the bin and lib directories across all projects. This is sometimes good -it helps keep copies of xerces.jar in sync, and sometimes bad -it can update foundational jar files before unit testing is complete.

To still have a single build across the sub projects, use parent build.xml files which call down into the sub projects.

This style works well if different teams have different code access/commitment rights. The risk is that by giving extra leeway to the sub projects, you can end up with incompatible source, libraries, build processes and just increase your workload and integration grief all round.

The only way to retain control over a fairly loosely integrated collection of projects is to have a fully automated build and test process which verifies that everything is still compatible. Sam Ruby runs one for all the apache java libraries and emails everyone when something breaks; your own project may be able to make use of Cruise Control for an automated, continuous, background build process.

9.6.4 Integrated sub projects

Tightly coupled projects have all the source in the same tree; different projects own different subdirectories. Build files can be moved down to those subdirectores (say src/com/iseran/core and src/com/iseran/extras), or kept at the top -with independent build files named core.xml and extras.xml

This project style works well if everyone trusts each other and the sub projects are not too huge or complex. The risk is that a split to a more loosely coupled design will become a requirement as the projects progress -but by the time this is realised schedule pressure and intertwined build files make executing the split well nigh impossible. If that happens then just keep with it until there is the time to refactor the project directory structures.

9.7 Ant Update Policies

Once you start using Ant, you should have a policy on when and how the team updates their copies. A simple policy is "every official release after whatever high stress milestone has pushed all unimportant tasks (like sleep and seeing daylight) on the back burner". This insulates you from the changes and occasional instabilities that Ant goes through during development. Its main disadvantage is that it isolates you from the new tasks and features that Ant is constantly adding.

Often an update will require changes to the build.xml files. Most changes are intended to be backwards compatible, but sometimes an incompatible change turns out to be necessary. That is why doing the update in the lull after a big milestone is important. It is also why including ant.jar and related files in the CVS tree helps ensure that old versions of your software can be still be built.

The most aggressive strategy is to get a weekly or daily snapshot of the ant source, build it up and use it. This forces you to tweak the build.xml files more regulary, as new tasks and attributes can take while to stabilise. You really have to want the new features, enjoy gratuitous extra work or take pleasure in upsetting your colleagues to take this approach.

Once you start extending Ant with new tasks, it suddenly becomes much more tempting to pull down regular builds. The most recent Ant builds are invariably the the best platform for writing your extensions, as you can take advantage of the regular enhancements to the foundational classes. It also prevents you from wasting time working on something which has already been done. A newly submitted task to do something complex such as talk to EJB engines, SOAP servers or just convert a text file to uppercase may be almost exactly what you need -so take it, enhance it and offer up the enhancements to the rest of the world. This is certainly better than

starting work on your 'text case converter' task on Ant 0.8 in isolation, announcing its existence six months latter and discovering that instead of adulation all you get are helpful pointers to the existing implementation. The final benefit of being involved with the process is that it makes it easier for your tasks to be added with the Ant CVS tree, bringing forward the date when Ant has taken on all the changes you needed to make to get your project to work. If that happens you can revert to an official Ant release, and get on with all the other crises.

You should also get on the dev mailing list , as it is where the other developers post their work, problems and experience. The volume can be quite high: 40+ messages a day, so consider routing it to an email address you don't use for much else. And don't make everyone on the team subscribe; it can be too much of a distraction.

9.8 Installing with Ant.

Because Ant can read environment variables, copy, unzip and delete files and make java and OS calls, it can be used for simple installation tasks. For example, an installer for tomcat could extract the environment variable TOMCAT_HOME, stop tomcat running, and copy a war file to TOMCAT_HOME/webapps. It could even start tomcat again, but the build wouldn't complete until tomcat exited, which is probably not what was wanted.

The advantage of using Ant is firstly that the same install targets can be used from your local build files (via an ant invocation of the install.xml file), and secondly that a basic install target is quite easy to write. The disadvantages of this approach are that the destination must have an up to date version of Ant correctly pre-installed, and Ant doesn't allow you to handle failures well -and a good installer is all about handling when things go wrong, from files being in use to jar versions being different. This means that Ant is not suited for shrink wrapped software, but it does work for deployment and installation to your local servers.

One major build project I was involved in had an Ant install build file for the bluestone application server, which would shutdown all four instances of the app server on a single machine, copy the new version of the war file (with datestamp and buildstamp) to an archive directory, clean up the current deployed version of the war and then install the new version. Because bluestone restarted JVMs on demand, this script was all you needed for web service deployment. On the systems behind the firewall, we upped the ante in the deployment process by using the ftp task to copy out the war and build files, then the telnet task to remotely invoke the build file. The result was we had automated recompile and redeploy to local servers from inside our IDE (Jedit) or the command line, which was simply invaluable. Imagine pressing a button on your IDE toolbar to build, unit test, deploy and then functional test your webapp.

One extra trick I added later was a junit test case to run through the install check list. With tests to verify access permissions on network drives, approximate clock synchronisation between servers, DNS functionality, ability to spawn executables and all the other trouble spots, the install script could automatically do a system health test during install time and report problems. [The same tests could also be invoked from a JMX MBean, but that's another story].

So, Ant is not a substitute for a real installer tool, except in the special case of servers you control, but in that context it does let you integrate remote installation with your build.

9.9 Tips and Tricks

9.9.1 get

The <get> task can fetch any URL, so be used to trigger remote server side code during the build process, from remote server restarts to sending SMS/pager messages to the developer cellphones.

9.9.2 i18n

Internationalisation is always trouble. Ant helps here with the native2ascii task which can escape out all non ascii characters into unicode. You can use this to write java files which include strings (and indeed comments) in your own non-ASCII language and then use native2ascii to convert to ascii prior to feeding through javac. The rest of i18n and l12n is left to you...

9.9.3 Use Property Files

Use external property files to keep per-user settings out the build files -especially passwords. Property files can also be used to dynamically set a number of properties based on the value of a single property, simply by dyamically generating the property filename from the source property. They can also be used as a source of constants across multiple build files.

9.9.4 Faster compiles with Jikes

The jikes compiler is usually much faster than javac, does dependency checking and has better error messages (usually). Get it. Then set build.compiler to "jikes" for it to be used in your build files. Doing this explicitly in your build files is a bit dubious as it requires the whole team (and sub projects) to be using jikes too -something you can only control in small, closed source projects. But if you set ANT_OPTS = -Dbuild.compiler=jikes in your environment, then all your builds on your system will use Jikes automatically, while others can choose their own compiler, or let ant choose whichever is appropriate for the current version of Java.

9.9.5 #include targets to simplify multi build.xml projects

You can import XML files into a build file using the XML parser itself. This lets a multi-project development program share code through reference, rather than cut and paste re-use. It also lets one build up a file of standard tasks which can be reused over time. Because the import mechanism is at a level below which Ant is aware, treat it as equivalent to the #include mechanism of the 'legacy' languages C and C++.

There are two inclusion mechanisms, an ugly one for all parsers and a clean one. The ugly method is the only one that was available on Ant1.5 and earlier:-

```
    <!DOCTYPE project [
      <!ENTITY propertiesAndPaths SYSTEM "propertiesAndPaths.xml">
      <!ENTITY taskdefs SYSTEM "taskdefs.xml">
    ]>

        &propertiesAndPaths;
        &taskdefs;
```

The cleaner method in Ant1.6 is the <import> task that imports whole build files into other projects. The entity inclusion example could almost be replaced by two import statements:-

```
 <import file="propertiesAndPaths.xml">
 <import file="taskdefs.xml">
```

We say almost as top level declarations (properties and taskdefs) do not get inserted into the XML file exactly where the import statement goes, but added to the end of the file. This is because the import process takes place after the main build file is parsed, during execution, whereas XML entity expansion is handled during the parsing process.

The <import> task does powerful things, such as let you override targets, and use ant properties to name the location of the file to import. Consult the documentation for the specifics of these features.

Before you go overboard with using XML inclusion, note that the <ant> task lets you call any target in any other build file -with all your property settings propagating down to that target. So you can actually have a suite of utility targets -"deploy-to-stack-a", "email-to-team", "cleanup-installation" which can be called from any of your main build files, perhaps with subtly changed parameters. Indeed, after a couple of projects you may be able to create a re-usable core build file which contains the core targets of a basic Java development project -compile, debug, deploy- which project specific build files call with their own settings. If you can achive this then you are definately making your way up the software maturity ladder. With a bit of work you may progress from

being a SEI CMM Level 0 organisation "Individual Heroics are not enough" to SEI CMM Level 1, "Projects only succeed due to individual heroics"

NB, <ant> copies all your properties unless the inheritall attribute is set to false. Before that attribute existed you had to carefully name all property definitions in all build files to prevent unintentional overwriting of the invoked property by that of the caller, now you just have to remember to set inheritall="false" on all uses of the <ant> task.

9.9.6 Implement complex Ant builds through XSL

XSLT can be used to dynamically generate build.xml files from a source xml file, with the <xslt> task controlling the transform. This is the current recommended strategy for creating complex build files dynamically. However, its use is still apparently quite rare -which means you will be on the bleeding edge of technology.

9.9.7 Change the invocation scripts

By writing your own invocation script -using the DOS, Unix or Perl script as a starting point- you can modify Ant's settings and behavior for an individual project. For example, you can use an alternate variable to ANT_HOME as the base, extend the classpath differently, or dynamically create a new command line property 'project.interfaces' from all .jar files in an interfaces directory.

Having a custom invocation script which runs off a CVS controlled library tree under PROJECT_HOME also lets you control Ant versions across the team -developers can have other copies of Ant if they want, but the CVS tree always contains the jar set used to build your project.

You can also write wrapper scripts which invoke the existing Ant scripts. This is an easy way to extend them. The wrapper scripts can add extra definitions and name explicit targets, redefine ANT_HOME and generally make development easier. Note that "ant" in Windows is really "ant.bat", so should be invoked from another batch file with a "CALL ant" statement -otherwise it never returns to your wrapper.

9.9.8 Write all code so that it can be called from Ant

This seems a bit strange and idealistic, but what it means is that you should write all your java code as if it may be called as a library at some point in future. So do not place calls to System.exit() deep in the code -if you want to exit a few functions in, raise an exception instead and have main() deal with it.

Moving one step further, consider proving an Ant Task interface to the code as a secondary, primary or even sole interface to the functionality. Ant actually makes a great bootloader for Java apps as it handles classpath setup, and you can re-use all the built in tasks for preamble and postamble work. Some projects, such as XDoclet only run under Ant, because that is the right place to be.

9.9.9 Use the replace task to programmatic modify text files in your project.

Imagine your project has some source files -BAT files, ASPX pages (!), anything which needs to be statically customised at compile time for particular installations, such driven from some properties of the project such as JVM options, or the URL to direct errors too. The replace task can be used to modify files, substituting text and creating versions customised for that build or destination. Of course, per-destination customisation should be delayed until installation, but if you are using Ant for the remote installation that suddenly becomes feasible.

9.9.10 Use the mailing lists

There are two mailing lists related to Ant, user and developer. Ant user is where all questions related to using Ant should go. Installation, syntax, code samples, etc -post your questions there or search the archives for whether the query has been posted and answered before. Ant-developer is where Ant development takes place -so it is not the place to post things like "I get a compilation error when I build my project" or "how do I make a zip file". If you are actually extending Ant, on the other hand, it is the ideal place to ask questions about how to

add new tasks, make changes to existing ones -and to post the results of your work, if you want them incorporated into the Ant source tree.

9.10 Putting it all together

What does an Ant build process look like in this world? Assuming a single directory structure for simplicity, the build file should contain a number of top level targets

- build - do an (incremental) build
- test - run the junit tests
- clean - clean out the output directories
- deploy - ship the jars, wars, whatever to the execution system
- publish - output the source and binaries to any distribution site
- fetch - get the latest source from the cvs tree
- docs/javadocs - do the documenation
- all - clean, fetch, build, test, docs, deploy
- main - the default build process (usually build or build & test)

Sub projects 'web', 'bean-1', 'bean-2' can be given their own build files -web.xml, bean-1.xml, bean-2.xml- with the same entry points. Extra toplevel tasks related to databases, web site images and the like should be considered if they are part of the process.

Debug/release switching can be handled with separate initialisation targets called before the compile tasks which define the appropriate properties. Antcall is the trick here, as it allows you to have two paths of property initialisation in a build file.

Internal targets should be used to structure the process

- init - initialise properties, extra-tasks, read in per-user property files.
- init-release - initialise release properties
- compile - do the actual compilation
- link/jar - make the jars or equivalent
- staging - any pre-deployment process in which the output is dropped off then tested before being moved to the production site.

The switching between debug and release can be done by making init-release conditional on a property, such as release.build being set :-

```
<target name="init-release" if="release.build">
    <property name="build.debuglevel" value="lines,source"/>
</target>
```

You then have dependent targets, such as "compile", depend on this conditional target; there the 'default' properties are set, and then the property is actually used. Because Ant properties are immutable, if the release target was executed its settings will override the default values:

```
<target name="compile" depends="init,init-release">
    <property name="build.debuglevel" value="lines,vars,source"/>
    <echo>debug level=${build.debuglevel}</echo>
    <javac destdir="${build.classes.dir}"
       debug="true"
       debuglevel="${build.debuglevel}"
       includeAntRuntime="false"
       srcdir="src">
      <classpath refid="compile.classpath"/>
    </javac>
</target>
```

As a result, we now have a build where the release mode only includes the filename and line debug information (useful for bug reports), while the development system included variables too.

It is useful to define a project name property which can be echoed in the init task. This lets you work out which Ant file is breaking in a multi file build.

What goes in to the internal Ant tasks depends on your own projects. One very important tactic is 'keep path redefinition down through references' - you can reuse paths by giving them an ID and then referring to them via the 'refid' attribute you should only need to define a shared classpath once in the file; filesets can be reused similarly.

Once you have set up the directory structures, and defined the Ant tasks it is time to start coding. An early priority must be to set up the automated test process, as that not only helps ensures that the code works, it verifies that the build process is working.

And that's it. The build file shouldn't need changing as new source files get added, only when you want to change the deliverables or part of the build process. At some point you may want to massively restructure the entire build process, restructuring projects and the like, but even then the build file you have should act as a foundation for a split build file process -just pull out the common properties into a properties file all build files read in, keep the target names unified and keep going with the project. Restructuring the source code control system is often much harder work.

9.11 The Limits of Ant

Before you start adopting Ant as the sole mechanism for the build process, you need to be aware of what it doesn't do.

9.11.1 It's not a scripting language

Ant lets you declare what you want done, with a bit of testing of the platform and class libraries first to enable some platform specific builds to take place. It does not let you specify how to handle things going wrong (a listener class can do that), or support complex conditional statements.

If your build needs to handle exceptions then look at the sound listener as a simple example of how to write your own listener class. Complex conditional statements can be handled by having something else do the tests and then build the appropriate Ant task. XSLT can be used for this.

9.11.2 It's not Make

Some of the features of make, specifically inference rules and dependency checking are not included in Ant. That's because they are 'different' ways of doing a build. Make requires you to state dependencies and the build steps, Ant wants you to state tasks and the order between them, the tasks themselves can do depedency checking or not. A full java build using Jikes is so fast that dependency checking is relatively moot, while many of the other tasks (but not all), compare the timestamp of the source file with that of the destination file before acting.

9.11.3 It's not meant to be a nice language for humans

XML isn't a nice representation of information for humans. It's a reasonable representation for programs, and text editors and source code management systems can all handle it nicely. But a complex Ant file can get ugly because XML is a bit ugly, and a complex build is, well, complicated. Use XML comments so that the file you wrote last month still makes sense when you get back to it, and use Antidote to edit the files if you prefer it.

9.11.4 Big projects still get complicated fast

Large software projects create their own complexity, with inter-dependent libraries, long test cycles, hard deployment processes and a multitude of people each working on their own bit of the solution. That's even before the deadlines loom close, the integration problems become insurmountable, weekends become indistinguishable from weekdays in terms of workload and half the team stops talking to the other half. Ant may simplify the build and test process, and can eliminate the full time 'makefile engineer' role, but that doesn't mean that someone can stop 'owning the build'. Being in charge of the build has to mean more than they type 'ant all' on their system, it means they need to set the standards of what build tools to use, what the common targets, what property names and files should be and generally oversee the sub projects build processes. On a small project, you don't need to do that -but remember: small projects become big projects when you aren't looking. If you start off with a little bit of process, then you can scale it if needed. If you start with none, by the time you need it it will be too late.

9.11.5 You still need all the other foundational bits of a software project

If you don't have an source code management system, you are going to end up hosed. If you don't have everything under SCM, including web pages, dependent jars, installation files, you are still going to end up hosed, it's just a question of when it's going to happen. CVS is effectively free and works well with Ant, but Sourcesafe, Perforce, Clearcase and StarTeam also have Ant tasks. These tasks let you have auto-incrementing build counters, and automated file update processes.

You also need some kind of change control process, to resist uncontrolled feature creep. Bugzilla is a simple and low cost tool for this, using Ant and a continuous test process enables a rapid evolution of code to adapt to those changes which are inevitable.

9.12 Endpiece

Software development is meant to be fun. Being in the maelstrom of a tight project with the stress of integration and trying to code everything up for an insane deadline can be fun -it is certainly exhilirating. Adding a bit of automation to the process may make things less chaotic, and bit less entertaining, but it is a start to putting you in control of your development process. You can still have fun, you should just have less to worry about, a shorter build/test/deploy cycle and more time to spend on feature creep or important things like skiing. So get out there and have fun!

9.13 Further Reading

- Continuous Integration; Martin Fowler.
  A paper on using Ant within a software project running a continuous integration/testing proces.
- Refactoring; Martin Fowler, ISBN: 0201485672
  Covers JUnit as well as tactics for making some headway with the mess of code you will soon have.
- Java Development with Ant; Erik Hatcher and Steve Loughran.
- When Web Services Go Bad; Steve Loughran.
  One of the projects this paper is based on.

9.14 About the Author

Steve Loughran is a research scientist at a corporate R&D lab, currently on a sabbatical building production web services against implausible deadlines for the fun of it. He is also a committer on Apache Ant and Apache Axis, and co-author of Java Development with Ant. He thinks that if you liked this document you'll love that book because it doesn't just explain Ant, it goes into processes, deployment and best practices and other corners of stuff that really make Ant useful. (It would have been easier to just rehash the manual, but that wouldn't have been so useful or as much fun).

For questions related to this document, use the Ant mailing list.

10 Apache Ant Task Design Guidelines

This document covers how to write ant tasks to a standard required to be incorporated into the Ant distribution. You may find it useful when writing tasks for personal use as the issues it addresses are still there in such a case.

10.1 Don't break existing builds

Even if you find some really hideous problem with ant, one that is easy to fix, if your fix breaks an existing build file then we have problems. Making sure that every build file out there still works, is one of the goals of all changes. As an example of this, Ant1.5 passes the single dollar sign "$" through in strings; Ant1.4 and before would strip it. To get this fix in we first had to write the test suite to expose current behavior, then change something so that singe $ was passed through, but double "$$" got mapped to "$" for backwards compatibility.

10.2 Use built in helper classes

Ant includes helper tasks to simplify much of your work. Be warned that these helper classes will look very different in ant2.0 from these 1.x versions. However it is still better to use them than roll your own, for development, maintenance and code size reasons.

10.2.1 Execute

Execute will spawn off separate programs under all the platforms which ant supports, dealing with Java version issues as well as platform issues. Always use this task to invoke other programs.

10.2.2 Java, ExecuteJava

These classes can be used to spawn Java programs in a separate VM (they use execute) or in the same VM - with or without a different classloader. When deriving tasks from this, it often benefits users to permit the classpath to be specified, and for forking to be an optional attribute.

10.2.3 Project and related classes

Project, FileUtils, JavaEnvUtils all have helper functions to do things like touch a file, to copy a file and the like. Use these instead of trying to code them yourself -or trying to use tasks which may be less stable and fiddlier to use.

10.3 Obey the Sun/Java style guidelines

The Ant codebase aims to have a single unified coding standard, and that standard is the Sun Java coding guidelines

It's not that they are better than any alternatives, but they are a standard and they are what is consistently used in the rest of the tasks. Code will not be incorporated into the database until it complies with these.

If you are writing a task for your personal or organisational use, you are free to use whatever style you like. But using the Sun Java style will help you to become comfortable with the rest of the Ant source, which may be important.

One important rule is 'no tabs'. Use four spaces instead. Not two, not eight, four. Even if your editor is configured to have a tab of four spaces, lots of others aren't -spaces have more consistency across editors and platforms. Some IDEs (JEdit) can highlight tabs, to stop you accidentally inserting them

There is an ant build file check.xml in the main ant directory with runs checkstyle over ant's source code.

10.4 Attributes and elements

Use the Ant introspection based mapping of attributes into Java datatypes, rather than implementing all your attributes as setFoo(String) and doing the mapping to Int, bool or file yourself. This saves work on your part, lets Java callers use you in a typesafe manner, and will let the Xdocs documentation generator work out what the parameters are.

The ant1.x tasks are very inconsistent regarding naming of attributes -some tasks use source, others src. Here is a list of preferred attribute names.

failonerror	boolean to control whether failure to execute should throw a BuildException or just print an error. Parameter validation failures should always throw an error, regardless of this flag
destdir	destination directory for output
destfile	destination file for output
srcdir	source directory
srcfile	source file

Yes, this is a very short list. Try and be vaguely consistent with the core tasks, at the very least.

10.5 Support classpaths

Try and make it possible for people to supply a classpath to your task, if you need external libraries, rather than make them add everything to the ANT_HOME\lib directory. This lets people keep the external libraries in their ant-based project, rather than force all users to make changes to their ant system configuration.

10.6 Design for controlled re-use

Keep member variables private. If read access by subclasses is required. add accessor methods rather than change the accessiblity of the member. This enables subclasses to access the contents, yet still be decoupled from the actual implementation.

The other common re-use mechanism in ant is for one task to create and configure another. This is fairly simple.

10.7 Do your own Dependency Checking

Make has the edge over Ant in its integrated dependency checking: the command line apps make invokes dont need to do their own work. Ant tasks do have to do their own dependency work, but if this can be done then it can be done well. A good dependency aware task can work out the dependencies without explicit dependency information in the build file, and be smart enough to work out the real dependencies, perhaps through a bit of file parsing. The depends task is the best example of this. Some of the zip/jar tasks are pretty good too, as they can update the archive when needed. Most tasks just compare source and destination timestamps and work from there. Tasks which don't do any dependency checking do not help users as much as they can, because their needless work can trickle through the entire build, test and deploy process.

10.8 Support Java 1.2 through Java 1.4

Ant1.5 and lower was designed to support Java1.1. Ant1.6 and higher is designed to support Java1.2: to build on it, to run on it. Sometimes functionality of tasks have to degrade in that environment - this is usually due to library limitations; such behaviour change must always be noted in the documentation.

What is problematic is code which is dependent on Java1.3 features -java.lang.reflect.Proxy, or Java1.4 features - java.io.nio for example. Be also aware of extra methods in older classes - like StringBuffer#append(StringBuffer). These can not be used directly by any code and still be able to compile and

run on a Java 1.2 system. If a new method in an existing class is to be used, it must be used via reflection and the NoSuchMethodException handled somehow.

What if code simply does not work on Java1.2? It can happen. It will probably be OK to have the task as an optional task, with compilation restricted to Java1.3 or later through build.xml modifications. Better still, use reflection to link to the classes at run time.

Java 1.4 adds a new optional change to the language itself, the assert keyword, which is only enabled if the compiler is told to compile 1.4 version source. Clearly with the 1.2 compatibility requirement, Ant tasks can not use this keyword. They also need to move away from using the JUnit assert() method and call assertTrue() instead.

Java 1.5 will (perhaps) add a new keyword - enum, one should avoid this for future compatibility.

10.9 Refactor

If the changes made to a task are making it too unwieldy, split it up into a cleaner design, refactor the code and submit not just feature creep but cleaner tasks. A common design pattern which tends to occur in the ant process is the adoption of the adapter pattern, in which a base class (say Javac or Rmi) starts off simple, then gets convoluted with support for multiple back ends -javac, jikes, jvc. A refactoring to split the programmable front end from the classes which provide the back end cleans up the design and makes it much easier to add new back ends. But to carry this off one needs to keep the interface and behaviour of the front end identical, and to be sure that no subclasses have been accessing data members directly -because these data members may not exist in the refactored design. Which is why having private data members is so important.

10.10 Test

Look in ant/src/testcases and you will find JUnit tests for the shipping ant tasks, to see how it is done and what is expected of a new task. Most of them are rudimentary, and no doubt you could do better for your task -feel free to do so!

A well written set of test cases will break the Ant task while it is in development, until the code is actually complete. And every bug which surfaces later should have a test case added to demonstrate the problem, and to fix it.

The test cases are a great way of testing your task during development. A simple call to 'build run-test' in the ant source tree will run all ant tests, to verify that your changes don't break anything. To test a single task, use the one shot ant run-single-test -Dtestcase=${testname} where ${testname} is the name of your test class.

The test cases are also used by the committers to verify that changes and patches do what they say. If you've got test cases it increases your credibility significantly. To be precise, we hate submissions without test cases, as it means we have to write them ourselves. This is something that only gets done if we need the task or it is perceived as utterly essential to many users.

Remember also that Ant 1.x is designed to compile and run on Java1.2, so you should test on Java 1.2 as well as any later version which you use. You can download an old SDK from Sun for this purpose.

Finally, run a full build test before and after you start developing your project, to make sure you havent broken anything else by accident.

10.11 Document

Without documentation, the task can't be used. So remember to provide a succint and clear html (soon, xml) page describing the task in a similar style to that of existing tasks. It should include a list of attributes and

elements, and at least one working example of the task. Many users cut and paste the examples into their build files as a starting point, so make the examples practical and test them too.

You can use the xdocs stuff in proposal/xdocs to autogenerate your documentation page from the javadocs of the source; this makes life easier and will make the transition to a full xdoclet generated documentation build process trivial.

10.12 Licensing and Copyright

Any code submitted to the Apache project must be compatible with the Apache Software License, and the act of submission must be viewed as an implicit transfer of ownership of the submitted code to the Apache Software Foundation.

This is important.

The fairly laissez-faire license of Apache is not compabitible with either the GPL or the Lesser GPL of the Free Software Foundation -the Gnu project. These licenses have stricter terms, "copyleft", which are not in the Apache Software Foundation license. This permits people and organisations to build commercial and closed source applications atop the Apache libraries and source -but not use the Apache, Ant or Jakarta Project names without permission.

Because the Gnu GPL license immediately extends to cover any larger application (or library, in the case of GLPL) into which it is incorporated, the Ant team can not incorporate any task based upon GPL or LGPL source into the Ant codebase. You are free to submit it, but it will be politely and firmly rejected.

Once ant-2 adds better dynamic task incorporation, it may be possible to provide a framework for indirectly supporting [L]GPL code, but still no tasks directly subject to the Gnu licenses can be included in the Ant CVS tree.

If you link to a GPL or LGPL library, by import or reflection, your task must be licensed under the same terms. So tasks linking to (L)GPL code can't go into the Apache managed codebase. Tasks calling such code can use the 'exec' or 'java' tasks to run the programs, as you are just executing them at this point, not linking to them.

Even if we cannot include your task into the Apache codebase, we can still point to where you host it -just submit a diff to xdocs/external.html pointing to your task. If your task links directly to proprietary code, we have a differnt problem: it is really hard to build the tasks. Please use reflection.

10.12.1 Dont re-invent the wheel

We've all done it: written and submitted a task only to discover it was already implemented in a small corner of another task, or it has been submitted by someone else and not committed. You can avoid this by being aware of what is in the latest CVS tree -keep getting the daily source updates, look at manual changes and subscribe to the dev mailing list.

If you are thinking of writing a task, posting a note on your thoughts to the list can be informative -you well get other peoples insight and maybe some half written task to do the basics, all without writing a line of code.

10.13 Submitting to Ant

The process for submitting an Ant task is documented on the jakarta web site. The basic mechanism is to mail it to the dev mailing list. It helps to be on this list, as you will see other submissions, and any debate about your own submission.

You may create your patch file using either of the following approaches. The committers recommend you to take the first approach.

Approach 1 - The Ant Way
Use Ant to generate a patch file to Ant:

ant -f patch.xml

This will create a file named patch.tar.gz that will contain a unified diff of files that have been modified and also include files that have been added. Review the file for completeness and correctness. This approach is recommended because it standardizes the way in which patch files are constructed. It also eliminates the chance of you missing to submit new files that constitute part of the patch.

Approach 2 - The Manual Way
Patches to existing files should be generated with cvs diff -u filename and save the output to a file. If you want to get the changes made to multiple files in a directory , just use cvs diff -u. Then, Tar and GZip the patch file as well as any new files that you have added.

The patches should be sent as an attachment to a message titled [PATCH] and distinctive one-line summary in the subject of the patch. The filename/task and the change usually suffices. It's important to include the changes as an attachment, as too many mailers reformat the text pasted in, which breaks the patch.
Then you wait for one of the committers to commit the patch, if it is felt appropriate to do so. Bug fixes go in quickly, other changes often spark a bit of discussion before a (perhaps revised) commit is made.

New submissions should be proceeded with [SUBMIT]. The mailer-daemon will reject any messages over 100KB, so any large update should be zipped up. If your submission is bigger than that, why not break it up into separate tasks.

We also like submissions to be added to bugzilla, so that they dont get lost. Please submit them by first filing the report with a meaningful name, then adding files as attachments. Use CVS diff files please!

If you hear nothing after a couple of weeks, remind the mailing list. Sometimes really good submissions get lost in the noise of other issues. This is particularly the case just prior to a new point release of the product. At that time anything other than bug fixes will tend to be neglected.

10.14 Checklists

These are the things you should verify before submitting patches and new tasks. Things don't have to be perfect, it may take a couple of iterations before a patch or submission is committed, and these items can be addressed in the process. But by the time the code is committed, everything including the documentation and some test cases will have been done, so by getting them out the way up front can save time. The committers look more favourably on patches and submissions with test cases, while documentation helps sell the reason for a task.

10.14.1 Checklist before submitting a patch

- Added code complies with style guidelines
- Code compiles and runs on Java1.2
- New member variables are private, and provide public accessor methods if access is actually needed.
- Existing test cases succeed.
- New test cases written and succeed.
- Documentation page extended as appropriate.
- Example task declarations in the documentation tested.
- Diff files generated using cvs diff -u
- Message to dev contains [PATCH], task name and patch reason in subject.
- Message body contains a rationale for the patch.
- Message attachment contains the patch file(s).

10.14.2 Checklist before submitting a new task

- Java file begins with Apache copyright and license statement.
- Task does not depend on GPL or LGPL code.
- Source code complies with style guidelines
- Code compiles and runs on Java1.2
- Member variables are private, and provide public accessor methods if access is actually needed.
- Maybe Task has failonerror attribute to control failure behaviour
- New test cases written and succeed
- Documentation page written
- Example task declarations in the documentation tested.
- Patch files generated using cvs diff -u
- patch files include a patch to defaults.properties to register the tasks
- patch files include a patch to coretasklist.html or optionaltasklist.html to link to the new task page
- Message to dev contains [SUBMIT] and task name in subject
- Message body contains a rationale for the task
- Message attachments contain the required files -source, documentation, test and patches zipped up to escape the HTML filter.

11 Writing Your Own Task

It is very easy to write your own task:

1. Create a Java class that extends org.apache.tools.ant.Task or another class that was desgined to be extended.
2. For each attribute, write a setter method. The setter method must be a public void method that takes a single argument. The name of the method must begin with set, followed by the attribute name, with the first character of the name in uppercase, and the rest in lowercase*. That is, to support an attribute named file you create a method setFile. Depending on the type of the argument, Ant will perform some conversions for you, see below.
3. If your task shall contain other tasks as nested elements (like parallel), your class must implement the interface org.apache.tools.ant.TaskContainer. If you do so, your task can not support any other nested elements. See below.
4. If the task should support character data (text nested between the start end end tags), write a public void addText(String) method. Note that Ant does not expand properties on the text it passes to the task.
5. For each nested element, write a create, add or addConfigured method. A create method must be a public method that takes no arguments and returns an Object type. The name of the create method must begin with create, followed by the element name. An add (or addConfigured) method must be a public void method that takes a single argument of an Object type with a no-argument constructor. The name of the add (addConfigured) method must begin with add (addConfigured), followed by the element name. For a more complete discussion see below.
6. Write a public void execute method, with no arguments, that throws a BuildException. This method implements the task itself.

* Actually the case of the letters after the first one doesn't really matter to Ant, using all lower case is a good convention, though.

11.1 The Life-cycle of a Task

1. The task gets instantiated using a no-argument constructor, at parser time. This means even tasks that are never executed get instantiated.
2. The task gets references to its project and location inside the buildfile via its inherited project and location variables.
3. If the user specified an id attribute to this task, the project registers a reference to this newly created task, at parser time.
4. The task gets a reference to the target it belongs to via its inherited target variable.
5. init() is called at parser time.
6. All child elements of the XML element corresponding to this task are created via this task's createXXX() methods or instantiated and added to this task via its addXXX() methods, at parser time.
7. All attributes of this task get set via their corresponding setXXX methods, at runtime.
8. The content character data sections inside the XML element corresponding to this task is added to the task via its addText method, at runtime.
9. All attributes of all child elements get set via their corresponding setXXX methods, at runtime.
10. execute() is called at runtime. While the above initialization steps only occur once, the execute() method may be called more than once, if the task is invoked more than once. For example, if target1 and target2 both depend on target3, then running 'ant target1 target2' will run all tasks in target3 twice.

11.2 Conversions Ant will perform for attributes

Ant will always expand properties before it passes the value of an attribute to the corresponding setter method.

The most common way to write an attribute setter is to use a java.lang.String argument. In this case Ant will pass the literal value (after property expansion) to your task. But there is more! If the argument of you setter method is

- boolean, your method will be passed the value true if the value specified in the build file is one of true, yes, or on and false otherwise.
- char or java.lang.Character, your method will be passed the first character of the value specified in the build file.
- any other primitive type (int, short and so on), Ant will convert the value of the attribute into this type, thus making sure that you'll never receive input that is not a number for that attribute.
- java.io.File, Ant will first determine whether the value given in the build file represents an absolute path name. If not, Ant will interpret the value as a path name relative to the project's basedir.
- org.apache.tools.ant.types.Path, Ant will tokenize the value specified in the build file, accepting : and ; as path separators. Relative path names will be interpreted as relative to the project's basedir.
- java.lang.Class, Ant will interpret the value given in the build file as a Java class name and load the named class from the system class loader.
- any other type that has a constructor with a single String argument, Ant will use this constructor to create a new instance from the value given in the build file.
- A subclass of org.apache.tools.ant.types.EnumeratedAttribute, Ant will invoke this classes setValue method. Use this if your task should support enumerated attributes (attributes with values that must be part of a predefined set of values). See org/apache/tools/ant/taskdefs/FixCRLF.java and the inner AddAsisRemove class used in setCr for an example.

What happens if more than one setter method is present for a given attribute? A method taking a String argument will always lose against the more specific methods. If there are still more setters Ant could chose from, only one of them will be called, but we don't know which, this depends on the implementation of your Java virtual machine.

11.3 Supporting nested elements

Let's assume your task shall support nested elements with the name inner. First of all, you need a class that represents this nested element. Often you simply want to use one of Ant's classes like org.apache.tools.ant.types.FileSet to support nested fileset elements.

Attributes of the nested elements or nested child elements of them will be handled using the same mechanism used for tasks (i.e. setter methods for attributes, addText for nested text and create/add/addConfigured methods for child elements).

Now you have a class NestedElement that is supposed to be used for your nested <inner> elements, you have three options:

```
1. public NestedElement createInner()
2. public void addInner(NestedElement anInner)
3. public void addConfiguredInner(NestedElement anInner)
```

What is the difference?

Option 1 makes the task create the instance of NestedElement, there are no restrictions on the type. For the options 2 and 3, Ant has to create an instance of NestedInner before it can pass it to the task, this means, NestedInner must have a public no-arg constructor or a public one-arg constructor taking a Project class as a parameter. This is the only difference between options 1 and 2.

The difference between 2 and 3 is what Ant has done to the object before it passes it to the method. addInner will receive an object directly after the constructor has been called, while addConfiguredInner gets the object after the attributes and nested children for this new object have been handled.

What happens if you use more than one of the options? Only one of the methods will be called, but we don't know which, this depends on the implementation of your Java virtual machine.

11.4 Nested Types

If your task needs to nest an arbitary type that has been defined using <taskdef> you have two options.

1. `public void add(Type type)`
2. `public void addConfigured(Type type)`

The difference between 1 and 2 is the same as between 2 and 3 in the previous section.

For example suppose one wanted to handle objects object of type org.apache.tools.ant.taskdefs.condition.Condition, one may have a class:

```
public class MyTask extends Task {
    private List conditions = new ArrayList();
    public void add(Condition c) {
        conditions.add(c);
    }
    public void execute() {
     // iterator over the conditions
    }
}
```

One may define and use this class like this:

```
<taskdef name="mytask" classname="MyTask" classpath="classes"/>
<typedef name="condition.equals"
         classname="org.apache.tools.ant.taskdefs.conditions.Equals"/>
<mytask>
    <condition.equals arg1="${debug}" arg2="true"/>
</mytask>
```

A more complicated example follows:

```
public class Sample {
    public static class MyFileSelector implements FileSelector {
         public void setAttrA(int a) {}
         public void setAttrB(int b) {}
         public void add(Path path) {}
         public boolean isSelected(File basedir, String filename, File file) {
             return true;
         }
     }

    interface MyInterface {
        void setVerbose(boolean val);
    }

    public static class BuildPath extends Path {
        public BuildPath(Project project) {
            super(project);
        }

        public void add(MyInterface inter) {}
        public void setUrl(String url) {}
    }
```

```
    public static class XInterface implements MyInterface {
        public void setVerbose(boolean x) {}
        public void setCount(int c) {}
    }
}
```

This class defines a number of static classes that implement/extend Path, MyFileSelector and MyInterface. These may be defined and used as follows:

```
<typedef name="myfileselector" classname="Sample$MyFileSelector"
         classpath="classes" loaderref="classes"/>
<typedef name="buildpath" classname="Sample$BuildPath"
         classpath="classes" loaderref="classes"/>
<typedef name="xinterface" classname="Sample$XInterface"
         classpath="classes" loaderref="classes"/>

<copy todir="copy-classes">
   <fileset dir="classes">
      <myfileselector attra="10" attrB="-10">
         <buildpath path="." url="abc">
            <xinterface count="4"/>
         </buildpath>
      </myfileselector>
   </fileset>
</copy>
```

11.5 TaskContainer

The `TaskContainer` consists of a single method, addTask that basically is the same as an add method for nested elements. The task instances will be configured (their attributes and nested elements have been handled) when your task's execute method gets invoked, but not before that.

When we said execute would be called, we lied ;-). In fact, Ant will call the perform method in org.apache.tools.ant.Task, which in turn calls execute. This method makes sure that Build Events will be triggered. If you execute the task instances nested into your task, you should also invoke perform on these instances instead of execute.

11.6 Examples

Let's write our own task, which prints a message on the System.out stream. The task has one attribute, called message.

```
package com.mydomain;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class MyVeryOwnTask extends Task {
    private String msg;

    // The method executing the task
    public void execute() throws BuildException {
        System.out.println(msg);
    }

    // The setter for the "message" attribute
    public void setMessage(String msg) {
```

```
        this.msg = msg;
    }
}
```

It's really this simple ;-)

Adding your task to the system is rather simple too:
1. Make sure the class that implements your task is in the classpath when starting Ant.
2. Add a <taskdef> element to your project. This actually adds your task to the system.
3. Use your task in the rest of the buildfile.

Example 1

```
<?xml version="1.0"?>

<project name="OwnTaskExample" default="main" basedir=".">
  <taskdef name="mytask" classname="com.mydomain.MyVeryOwnTask"/>

  <target name="main">
    <mytask message="Hello World! MyVeryOwnTask works!"/>
  </target>
</project>
```

Example 2

To use a task directly from the buildfile which created it, place the <taskdef> declaration inside a target *after the compilation*. Use the classpath attribute of <taskdef> to point to where the code has just been compiled.

```
<?xml version="1.0"?>

<project name="OwnTaskExample2" default="main" basedir=".">

  <target name="build" >
    <mkdir dir="build"/>
    <javac srcdir="source" destdir="build"/>
  </target>

  <target name="declare" depends="build">
    <taskdef name="mytask"
        classname="com.mydomain.MyVeryOwnTask"
        classpath="build"/>
  </target>

  <target name="main" depends="declare">
    <mytask message="Hello World! MyVeryOwnTask works!"/>
  </target>
</project>
```

Another way to add a task (more permanently), is to add the task name and implementing class name to the default.properties file in the org.apache.tools.ant.taskdefs package. Then you can use it as if it were a built-in task.

11.7 Build Events

Ant is capable of generating build events as it performs the tasks necessary to build a project. Listeners can be attached to Ant to receive these events. This capability could be used, for example, to connect Ant to a GUI or to integrate Ant with an IDE.

To use build events you need to create an ant Project object. You can then call the addBuildListener method to add your listener to the project. Your listener must implement the org.apache.tools.antBuildListener interface. The listener will receive BuildEvents for the following events

- Build started
- Build finished
- Target started
- Target finished
- Task started
- Task finished
- Message logged

If you wish to attach a listener from the command line you may use the -listener option. For example:

```
ant -listener org.apache.tools.ant.XmlLogger
```

will run Ant with a listener that generates an XML representation of the build progress. This listener is included with Ant, as is the default listener, which generates the logging to standard output.

Note: A listener must not access System.out and System.err directly since ouput on these streams is redirected by Ant's core to the build event system. Accessing these streams can cause an infinite loop in Ant. Depending on the version of Ant, this will either cause the build to terminate or the Java VM to run out of Stack space. A logger, also, may not access System.out and System.err directly. It must use the streams with which it has been configured.

11.8 Source code integration

The other way to extend Ant through Java is to make changes to existing tasks, which is positively encouraged. Both changes to the existing source and new tasks can be incorporated back into the Ant codebase, which benefits all users and spreads the maintenance load around.

Please consult the Getting Involved pages on the Jakarta web site for details on how to fetch the latest source and how to submit changes for reincorporation into the source tree.

Ant also has some task guidelines which provides some advice to people developing and testing tasks. Even if you intend to keep your tasks to yourself, you should still read this as it should be informative.

12 Tasks Designed for Extension

These classes are designed to be extended. Always start here when writing your own task.

Class	Description
Task	Base class for all tasks.
AbstractCvsTask	Another task can extend this with some customized output processing
JDBCTask	Handles JDBC configuration needed by SQL type tasks.
MatchingTask	This is an abstract task that should be used by all those tasks that require to include or exclude files based on pattern matching.
Pack	Abstract Base class for pack tasks.
Unpack	Abstract Base class for unpack tasks.

13 InputHandler

13.1 Overview

When a task wants to prompt a user for input, it doesn't simply read the input from the console as this would make it impossible to embed Ant in an IDE. Instead it asks an implementation of the org.apache.tools.ant.input.InputHandler interface to prompt the user and hand the user input back to the task.

To do this, the task creates an InputRequest object and passes it to the InputHandler Such an InputRequest may know whether a given user input is valid and the InputHandler is supposed to reject all invalid input.

Exactly one InputHandler instance is associated with every Ant process, users can specify the implementation using the -inputhandler command line switch.

13.2 InputHandler

The InputHandler interface contains exactly one method

```
void handleInput(InputRequest request)
    throws org.apache.tools.ant.BuildException;
```

with some pre- and postconditions. The main postcondition is that this method must not return unless the request considers the user input valid, it is allowed to throw an exception in this situation.

Ant comes with two built-in implementations of this interface:

13.2.1 DefaultInputHandler

This is the implementation you get, when you don't use the -inputhandler command line switch at all. This implementation will print the prompt encapsulated in the request object to Ant's logging system and re-prompt for input until the user enters something that is considered valid input by the request object. Input will be read from the console and the user will need to press the Return key.

13.2.2 PropertyFileInputHandler

This implementation is useful if you want to run unattended build processes. It reads all input from a properties file and makes the build fail if it cannot find valid input in this file. The name of the properties file must be specified in the Java system property ant.input.properties.

The prompt encapsulated in a request will be used as the key when looking up the input inside the properties file. If no input can be found, the input is considered invalid and an exception will be thrown.

Note that `ant.input.properties` must be a Java system property, not an Ant property. I.e. you cannot define it as a simple parameter to ant, but you can define it inside the ANT_OPTS environment variable.

13.3 InputRequest

Instances of org.apache.tools.ant.input.InputRequest encapsulate the information necessary to ask a user for input and validate this input.

The instances of InputRequest itself will accept any input, but subclasses may use stricter validations. org.apache.tools.ant.input.MultipleChoiceInputRequest should be used if the user input must be part of a predefined set of choices.

14 Using Ant Tasks Outside of Ant

14.1 Rationale

Ant provides a rich set of tasks for buildfile creators and administrators. But what about programmers? Can the functionality provided by Ant tasks be used in java programs?

Yes, and its quite easy. Before getting into the details, however, we should mention the pros and cons of this approach:

14.1.1 Pros

Robust	Ant tasks are very robust. They have been banged on by many people. Ant tasks have been used in many different contexts, and have therefore been instrumented to take care of a great many boundary conditions and potentially obscure errors.
Cross Platform	Ant tasks are cross platform. They have been tested on all of the volume platforms, and several rather unusual ones (Netware and OS/390, to name a few).
Community Support	Using Ant tasks means you have less of your own code to support. Ant code is supported by the entire Apache Ant community.

14.1.2 Cons

Dependency on Ant Libraries	Obviously, if you use an Ant task in your code, you will have to add "ant.jar" to your path. Of course, you could use a code optimizer to remove the unnecessary classes, but you will still probably require a chunk of the Ant core.
Loss of Flexibility	At some point, if you find yourself having to modify the Ant code, it probably makes more sense to "roll your own." Of course, you can still steal some code snippets and good ideas. This is the beauty of open source!

14.2 Example

Let's say you want to unzip a zip file programmatically from java into a certain directory. Of course you could write your own routine to do this, but why not use the Ant task that has already been written?

In my example, I wanted to be able to unzip a file from within an XSLT Transformation. XSLT Transformers can be extended by plugging in static methods in java. I therefore need a function something like this:

```
/**
 * Unzip a zip file into a given directory.
 *
 * @param zipFilepath A pathname representing a local zip file
 * @param destinationDir where to unzip the archive to
 */
 static public void unzip(String zipFilepath, String destinationDir)
```

The Ant task to perform this function is `org.apache.tools.ant.taskdefs.Expand`. All we have to do is create a dummy Ant Project and Target, set the Task parameters that would normally be set in a buildfile, and call execute().

First, let's make sure we have the proper includes:

```
import org.apache.tools.ant.Project;
import org.apache.tools.ant.Target;
```

```
import org.apache.tools.ant.taskdefs.Expand;
import java.io.File;
```

The function call is actually quite simple:

```
static public void unzip(String zipFilepath, String destinationDir) {

    final class Expander extends Expand {
        public Expander() {
          project = new Project();
          project.init();
          taskType = "unzip";
          taskName = "unzip";
          target = new Target();
      }
    }
    Expander expander = new Expander();
    expander.setSrc(new File(zipfile));
    expander.setDest(new File(destdir));
    expander.execute();
```

In actual practice, you will probably want to add your own error handling code and you may not want to use a local inner class. However, the point of the example is to show how an Ant task can be called programmatically in relatively few lines of code.

The question you are probably asking yourself at this point is: How would I know which classes and methods have to be called in order to set up a dummy Project and Target? The answer is: you don't. Ultimately, you have to be willing to get your feet wet and read the source code. The above example is merely designed to whet your appetite and get you started. Go for it!

15 Tutorial: Writing Tasks

This document provides a step by step tutorial for writing tasks.

Content:

15.1 Set up the build environment

Ant builds itself, we are using Ant too (why we would write a task if not? :-) therefore we should use Ant for our build.

We choose a directory as root directory. All things will be done here if I say nothing different. I will reference this directory as root-directory of our project. In this root-directory we create a text file names build.xml. What should Ant do for us?

- compiles my stuff
- make the jar, so that I can deploy it
- clean up everything

So the buildfile contains three targets.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="jar">

    <target name="clean" description="Delete all generated files">
        <delete dir="classes"/>
        <delete file="MyTasks.jar"/>
    </target>

    <target name="compile" description="Compiles the Task">
        <javac srcdir="src" destdir="classes"/>
    </target>

    <target name="jar" description="JARs the Task">
        <jar destfile="MyTask.jar" basedir="classes"/>
    </target>

</project>
```

This buildfile uses often the same value (src, classes, MyTask.jar), so we should rewrite that using <property>s. On second there are some handicaps: <javac> requires that the destination directory exists; a call of "clean"

with a non existing classes directory will fail; "jar" requires the execution of some steps bofore. So the refactored code is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="jar">

    <property name="src.dir" value="src"/>
    <property name="classes.dir" value="classes"/>

    <target name="clean" description="Delete all generated files">
        <delete dir="${classes.dir}" failonerror="false"/>
        <delete file="${ant.project.name}.jar"/>
    </target>

    <target name="compile" description="Compiles the Task">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
    </target>

    <target name="jar" description="JARs the Task" depends="compile">
        <jar destfile="${ant.project.name}.jar" basedir="${classes.dir}"/>
    </target>

</project>
```

ant.project.name is one of the build-in properties [1] of Ant.

15.2 Write the Task

Now we write the simplest Task - a HelloWorld-Task (what else?). Create a text file HelloWorld.java in the src-directory with:

```
public class HelloWorld {
    public void execute() {
        System.out.println("Hello World");
    }
}
```

and we can compile and jar it with ant (default target is "jar" and via its depends-clause the "compile" is executed before).

15.3 Use the Task

But after creating the jar we want to use our new Task. Therefore we need a new target "use". Before we can use our new task we have to declare it with <taskdef> [2]. And for easier process we change the default clause:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="use">

    ...

    <target name="use" description="Use the Task" depends="jar">
        <taskdef name="helloworld" classname="HelloWorld"
classpath="${ant.project.name}.jar"/>
        <helloworld/>
    </target>
```

```
</project>
```

Important is the classpath-attribute. Ant searches in its /lib directory for tasks and our task isn´t there. So we have to provide the right location.
Now we can type in ant and all should work ...

```
Buildfile: build.xml

compile:
    [mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes
    [javac] Compiling 1 source file to C:\tmp\anttests\MyFirstTask\classes

jar:
      [jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

use:
[helloworld] Hello World

BUILD SUCCESSFUL
Total time: 3 seconds
```

15.4 Integration with TaskAdapter

Our class has nothing to do with Ant. It extends no superclass and implements no interface. How does Ant know to integrate? Via name convention: our class provides a method with signature public void execute(). This class is wrapped by Ant´s org.apache.tools.ant.TaskAdapter which is a task and uses reflection for setting a reference to the project and calling the execute() method.

Setting a reference to the project? Could be interesting. The Project class gives us some nice abilities: access to Ant´s logging facilities getting and setting properties and much more. So we try to use that class:

```
import org.apache.tools.ant.Project;

public class HelloWorld {

    private Project project;

    public void setProject(Project proj) {
        project = proj;
    }

    public void execute() {
        String message = project.getProperty("ant.project.name");
        project.log("Here is project '" + message + "'.", Project.MSG_INFO);
    }
}
```

and the execution with ant will show us the expected

```
use:
Here is project 'MyTask'.
```

15.5 Deriving from Ant´s Task

Ok, that works ... But usually you will extend org.apache.tools.ant.Task. That class is integrated in Ant, get´s the project-reference, provides documentation fiels, provides easier access to the logging facility and (very useful) gives you the exact location where in the buildfile this task instance is used.

Oki-doki - let´s us use some of these:

```
import org.apache.tools.ant.Task;

public class HelloWorld extends Task {
    public void execute() {
        // use of the reference to Project-instance
        String message = getProject().getProperty("ant.project.name");

        // Task´s log method
        log("Here is project '" + message + "'.");

        // where this task is used?
        log("I am used in: " +  getLocation() );
    }
}
```

which gives us when running

```
use:
[helloworld] Here is project 'MyTask'.
[helloworld] I am used in: C:\tmp\anttests\MyFirstTask\build.xml:23:
```

15.6 Attributes

Now we want to specify the text of our message (it seems that we are rewriting the <echo/> task :-). First we well do that with an attribute. It is very easy - for each attribute provide a public void set<attributename>(<type> newValue) method and Ant will do the rest via reflection.

```
import org.apache.tools.ant.Task;
import org.apache.tools.ant.BuildException;

public class HelloWorld extends Task {

    String message;
    public void setMessage(String msg) {
        message = msg;
    }

    public void execute() {
        if (message==null) {
            throw new BuildException("No message set.");
        }
        log(message);
    }
}
```

Oh, what´s that in execute()? Throw a BuildException? Yes, that´s the usual way to show Ant that something important is missed and complete build should fail. The string provided there is written as build-failes-message.

Here it´s necessary because the log() method can´t handle a null value as parameter and throws a NullPointerException. (Of course you can initialize the message with a default string.)

After that we have to modify our buildfile:

```
    <target name="use" description="Use the Task" depends="jar">
        <taskdef name="helloworld"
                 classname="HelloWorld"
                 classpath="${ant.project.name}.jar"/>
        <helloworld message="Hello World"/>
    </target>
```

That´s all.

Some background for working with attributes: Ant supports any of these datatypes as arguments of the set-method:

- elementary data type like int, long, ...
- its wrapper classes like java.lang.Integer, java.lang.Long, ...
- java.lang.String
- some more classes (e.g. java.io.File; see Manual 'Writing Your Own Task' [3])

Before calling the set-method all properties are resolved. So a <helloworld message="${msg}"/> would not set the message string to "${msg}" if there is a property "msg" with a set value.

15.7 Nested Text

Maybe you have used the <echo> task in a way like <echo>Hello World</echo>. For that you have to provide a public void addText(String text) method.

```
...
public class HelloWorld extends Task {
    ...
    public void addText(String text) {
        message = text;
    }
    ...
}
```

But here properties are **not** resolved! For resolving properties we have to use Project´s replaceProperties(String propname) : String method which takes the property name as argument and returns its value (or ${propname} if not set).

15.8 Nested Elements

There are several ways for inserting the ability of handling nested elements. See the Manual [4] for other. We use the first way of the three described ways. There are several steps for that:

1. We create a class for collecting all the infos the nested element should contain. This class is created by the same rules for attributes and nested elements as for the task (set<attributename>() methods).
2. The task holds multiple instances of this class in a list.
3. A factory method instantiates an object, saves the reference in the list and returns it to Ant Core.
4. The execute() method iterates over the list and evaluates its values.

```
import java.util.Vector;
import java.util.Iterator;
```

```
...
    public void execute() {
        if (message!=null) log(message);
        for (Iterator it=messages.iterator(); it.hasNext(); ) {      // 4
            Message msg = (Message)it.next();
            log(msg.getMsg());
        }
    }


    Vector messages = new Vector();                                  // 2

    public Message createMessage() {                                 // 3
        Message msg = new Message();
        messages.add(msg);
        return msg;
    }

    public class Message {                                           // 1
        public Message() {}

        String msg;
        public void setMsg(String msg) { this.msg = msg; }
        public String getMsg() { return msg; }
    }
...
```

Then we can use the new nested element. But where is xml-name for that defined? The mapping XML-name : classname is defined in the factory method: public classname createXML-name(). Therefore we write in the buildfile

```
        <helloworld>
            <message msg="Nested Element 1"/>
            <message msg="Nested Element 2"/>
        </helloworld>
```

15.9 Our task in a little more complex version

For recapitulation now a little refactored buildfile:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir="." default="use">
    <property name="src.dir" value="src"/>
    <property name="classes.dir" value="classes"/>

    <target name="clean" description="Delete all generated files">
        <delete dir="${classes.dir}" failonerror="false"/>
        <delete file="${ant.project.name}.jar"/>
    </target>
    <target name="compile" description="Compiles the Task">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
    </target>
    <target name="jar" description="JARs the Task" depends="compile">
        <jar destfile="${ant.project.name}.jar" basedir="${classes.dir}"/>
    </target>
```

```
    <target name="use.init"
            description="Taskdef the HelloWorld-Task"
            depends="jar">
        <taskdef name="helloworld"
                 classname="HelloWorld"
                 classpath="${ant.project.name}.jar"/>
    </target>
    <target name="use.without"
            description="Use without any"
            depends="use.init">
        <helloworld/>
    </target>
    <target name="use.message"
            description="Use with attribute 'message'"
            depends="use.init">
        <helloworld message="attribute-text"/>
    </target>
    <target name="use.fail"
            description="Use with attribute 'fail'"
            depends="use.init">
        <helloworld fail="true"/>
    </target>
    <target name="use.nestedText"
            description="Use with nested text"
            depends="use.init">
        <helloworld>nested-text</helloworld>
    </target>
    <target name="use.nestedElement"
            description="Use with nested 'message'"
            depends="use.init">
        <helloworld>
            <message msg="Nested Element 1"/>
            <message msg="Nested Element 2"/>
        </helloworld>
    </target>
    <target name="use"
            description="Try all (w/out use.fail)"
            depends="use.without,use.message,use.nestedText,use.nestedElement"
    />
</project>
```

And the code of the task:

```
import org.apache.tools.ant.Task;
import org.apache.tools.ant.BuildException;
import java.util.Vector;
import java.util.Iterator;

/**
 * The task of the tutorial.
 * Print´s a message or let the build fail.
 * @author Jan Matèrne
 * @since 2003-08-19
 */
public class HelloWorld extends Task {
```

```
    /** The message to print. As attribute. */
    String message;
    public void setMessage(String msg) {
        message = msg;
    }

    /** Should the build fail? Defaults to false. As attribute. */
    boolean fail = false;
    public void setFail(boolean b) {
        fail = b;
    }

    /** Support for nested text. */
    public void addText(String text) {
        message = text;
    }

    /** Do the work. */
    public void execute() {
        // handle attribute 'fail'
        if (fail) throw new BuildException("Fail requested.");

        // handle attribute 'message' and nested text
        if (message!=null) log(message);

        // handle nested elements
        for (Iterator it=messages.iterator(); it.hasNext(); ) {
            Message msg = (Message)it.next();
            log(msg.getMsg());
        }
    }

    /** Store nested 'message's. */
    Vector messages = new Vector();

    /** Factory method for creating nested 'message's. */
    public Message createMessage() {
        Message msg = new Message();
        messages.add(msg);
        return msg;
    }

    /** A nested 'message'. */
    public class Message {
        // Bean constructor
        public Message() {}

        /** Message to print. */
        String msg;
        public void setMsg(String msg) { this.msg = msg; }
        public String getMsg() { return msg; }
    }

}
```

And it works:

```
C:\tmp\anttests\MyFirstTask>ant
Buildfile: build.xml

compile:
    [mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes
    [javac] Compiling 1 source file to C:\tmp\anttests\MyFirstTask\classes

jar:
      [jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

use.init:

use.without:

use.message:
[helloworld] attribute-text

use.nestedText:
[helloworld] nested-text

use.nestedElement:
[helloworld]
[helloworld]
[helloworld]
[helloworld]
[helloworld] Nested Element 1
[helloworld] Nested Element 2

use:

BUILD SUCCESSFUL
Total time: 3 seconds
C:\tmp\anttests\MyFirstTask>ant use.fail
Buildfile: build.xml

compile:

jar:

use.init:

use.fail:

BUILD FAILED
C:\tmp\anttests\MyFirstTask\build.xml:36: Fail requested.

Total time: 1 second
C:\tmp\anttests\MyFirstTask>
```

Next step: test ...

15.10 Test the Task

We have written a test already: the use.* tasks in the buildfile. But its difficult to test that automatically. Common (and in Ant) used is JUnit for that. For testing tasks Ant provides a baseclass org.apache.tools.ant.BuildFileTest. This class extends junit.framework.TestCase and can therefore be integrated

into the unit tests. But this class provides some for testing tasks useful methods: initialize Ant, load a buildfile, execute targets, expecting BuildExceptions with a specified text, expect a special text in the output log ...

In Ant it is usual that the testcase has the same name as the task with a prepending Test, therefore we will create a file HelloWorldTest.java. Because we have a very small project we can put this file into src directory (Ant´s own testclasses are in /src/testcases/...). Because we have already written our tests for "hand-test" we can use that for automatic tests, too. But there is one little problem we have to solve: all test supporting classes are not part of the binary distribution of Ant. So you can build the special jar file from source distro with target "test-jar" or you can download a nightly build from http://gump.covalent.net/jars/latest/ant/ant-testutil.jar [5].

For executing the test and creating a report we need the optional tasks <junit> and <junitreport>. So we add to the buildfile:

```
...
<project name="MyTask" basedir="." default="test">
...
    <property name="ant.test.lib" value="ant-testutil.jar"/>
    <property name="report.dir"   value="report"/>
    <property name="junit.out.dir.xml"  value="${report.dir}/junit/xml"/>
    <property name="junit.out.dir.html" value="${report.dir}/junit/html"/>

    <path id="classpath.run">
        <path path="${java.class.path}"/>
        <path location="${ant.project.name}.jar"/>
    </path>

    <path id="classpath.test">
        <path refid="classpath.run"/>
        <path location="${ant.test.lib}"/>
    </path>

    <target name="clean" description="Delete all generated files">
        <delete failonerror="false" includeEmptyDirs="true">
            <fileset dir="." includes="${ant.project.name}.jar"/>
            <fileset dir="${classes.dir}"/>
            <fileset dir="${report.dir}"/>
        </delete>
    </target>

    <target name="compile" description="Compiles the Task">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}" classpath="${ant.test.lib}"/>
    </target>
...
    <target name="junit" description="Runs the unit tests" depends="jar">
        <delete dir="${junit.out.dir.xml}" />
        <mkdir  dir="${junit.out.dir.xml}" />
        <junit printsummary="yes" haltonfailure="no">
            <classpath refid="classpath.test"/>
            <formatter type="xml"/>
            <batchtest fork="yes" todir="${junit.out.dir.xml}">
                <fileset dir="${src.dir}" includes="**/*Test.java"/>
            </batchtest>
        </junit>
    </target>
```

```
    <target name="junitreport" description="Create a report for the rest result">
        <mkdir dir="${junit.out.dir.html}" />
        <junitreport todir="${junit.out.dir.html}">
            <fileset dir="${junit.out.dir.xml}">
                <include name="*.xml"/>
            </fileset>
            <report format="frames" todir="${junit.out.dir.html}"/>
        </junitreport>
    </target>

    <target name="test"
            depends="junit,junitreport"
            description="Runs unit tests and creates a report"
    />
...
```

Back to the src/HelloWorldTest.java. We create a class extending BuildFileTest with String-constructor (JUnit-standard), a setUp() method initializing Ant and for each testcase (targets use.*) a testXX() method invoking that target.

```
import org.apache.tools.ant.BuildFileTest;

public class HelloWorldTest extends BuildFileTest {

    public HelloWorldTest(String s) {
        super(s);
    }

    public void setUp() {
        // initialize Ant
        configureProject("build.xml");
    }

    public void testWithout() {
        executeTarget("use.without");
        assertEquals("Message was logged but should not.", getLog(), "");
    }

    public void testMessage() {
        // execute target 'use.nestedText' and expect a message
        // 'attribute-text' in the log
        expectLog("use.message", "attribute-text");
    }

    public void testFail() {
        // execute target 'use.fail' and expect a BuildException
        // with text 'Fail requested.'
        expectBuildException("use.fail", "Fail requested.");
    }

    public void testNestedText() {
        expectLog("use.nestedText", "nested-text");
    }

    public void testNestedElement() {
        executeTarget("use.nestedElement");
```

```
        assertLogContaining("Nested Element 1");
        assertLogContaining("Nested Element 2");
    }
}
```

When starting ant we´ll get a short message to STDOUT and a nice HTML-report.

```
C:\tmp\anttests\MyFirstTask>ant
Buildfile: build.xml

compile:
    [mkdir] Created dir: C:\tmp\anttests\MyFirstTask\classes
    [javac] Compiling 2 source files to C:\tmp\anttests\MyFirstTask\classes

jar:
      [jar] Building jar: C:\tmp\anttests\MyFirstTask\MyTask.jar

junit:
    [mkdir] Created dir: C:\tmp\anttests\MyFirstTask\report\junit\xml
    [junit] Running HelloWorldTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 2,334 sec

junitreport:
    [mkdir] Created dir: C:\tmp\anttests\MyFirstTask\report\junit\html
[junitreport] Using Xalan version: Xalan Java 2.4.1
[junitreport] Transform time: 661ms

test:

BUILD SUCCESSFUL
Total time: 7 seconds
C:\tmp\anttests\MyFirstTask>
```

15.11 Resources

This tutorial and its resources are available via BugZilla [6]. The ZIP provided there contains

- this tutorial
- the buildfile (last version)
- the source of the task (last version)
- the source of the unit test (last version)
- the ant-testutil.jar (nightly build of 2003-08-18)
- generated classes
- generated jar
- generated reports

The last sources and the buildfile are also available here [7] inside the manual.
Used Links:

[1] http://ant.apache.org/manual/using.html#built-in-props
[2] http://ant.apache.org/manual/CoreTasks/taskdef.html
[3] http://ant.apache.org/manual/develop.html#set-magic
[4] http://ant.apache.org/manual/develop.html#nested-elements
[5] http://gump.covalent.net/jars/latest/ant/ant-testutil.jar
[6] http://nagoya.apache.org/bugzilla/show_bug.cgi?id=22570
[7] tutorial-writing-tasks-src.zip

16 License