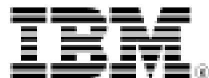


developerWorks®

Apache Ant 101: Make Java builds a snap

Skill Level: Introductory

Matt Chapman
Software engineer
IBM

17 Dec 2003

Whether you're a veteran user of Apache Ant or just starting out with this open source Java-based build tool, this tutorial provides a wealth of information. With Java developer and Ant enthusiast Matt Chapman from the IBM Java Technology Centre, you'll walk through the steps involved in writing a build file for a simple Java project, and then look at some of Ant's other useful functions, including filesystem operations and pattern matching. You'll finish the course by writing our own Java class that extends Ant's functionality.

Section 1. Getting started

What is this tutorial about?

In this tutorial, you'll learn about Ant, a build tool for Java TM projects. Ant has quickly become popular among Java developers because of its flexibility and ease of use, so you it owe it to yourself to find out more about it.

You don't need any prior experience with or knowledge of Ant before you proceed. We'll first see the basic structure of Ant build files, and learn how to invoke the tool. We'll walk through the steps involved in writing a build file for a simple Java project, and then look at some of Ant's other useful functions, including filesystem operations and pattern matching. We'll finish by writing our own Java class that extends Ant's functionality.

In the course of the tutorial, we'll show you how to run Ant both from the command line and from within the open source Eclipse IDE. You don't need both environments

© Copyright IBM Corporation 2003. All rights reserved.

to try out the examples; you can choose either, or even a different development environment, as long as it supports Ant. If you choose to use Ant from the command line, you will need to follow the installation instructions on the Ant home page (see Resources for a link) if Ant is not installed on your machine already. If instead you decide to use only the Eclipse environment, you don't need to install Ant separately, as it is included with Eclipse. If you don't have it already, you can download Eclipse from Eclipse.org (see Resources).

Should I take this tutorial?

If you write Java code and you're not already using Ant, then this tutorial is for you. Whether you're currently using a different build tool, or none at all, learning more about Ant will probably encourage you to make the effort to switch to it.

If you are already using Ant, then you might still find something of interest here. Perhaps you find some of Ant's behavior unexpected or difficult to fully understand; this tutorial will help. Or maybe you're familiar with the basics but want to know about advanced topics such as chaining builds together, working with CVS repositories, or writing custom tasks; we'll cover all of these topics here.

Ant is designed primarily for building Java projects, but that's not its only use. Many find it helpful for other tasks, such as performing filesystem operations in a cross-platform way. Also, there are a number of third-party Ant tasks available, and writing a custom Ant task is relatively simple, so it's easy to tailor Ant to a particular application.

Section 2. Ant basics

Ant basics introduction

In this section, we'll give you an overview of Ant's capabilities and strengths, and discuss its history and rise in popularity. We then delve straight into the fundamentals of Ant by examining the basic structure of a minimal build file. We'll also introduce the concepts of *properties* and *dependencies.*

What is Ant?

Apache Ant is a Java-based build tool. According to its original author, James

© Copyright IBM Corporation 2003. All rights reserved.

Duncan Davidson, the name is an acronym for *another neat tool.*

Build tools are used in software development to transform source code and other input files into an executable form (and maybe into installable product images as well). As an application's build process becomes more complex, it becomes increasingly important to ensure that precisely the same build steps are carried out during each build, with as much automation as possible, in order to produce consistent builds in a timely manner. Traditional projects in C or C++ often use the `make` tool for this purpose, where the build tasks are performed by invoking shell commands, and dependencies are defined between each of the build tasks so that they are always performed in the necessary order.

Ant is similar to `make` in that it defines dependencies between build tasks; however, instead of implementing build tasks using platform-specific shell commands, it uses cross-platform Java classes. With Ant, you can write a single build file that operates consistently on any Java platform (as Ant itself is implemented in the Java language); this is Ant's greatest strength.

Ant's other key strengths are its powerful simplicity and its ability to be seamlessly extended with custom capabilities. Hopefully, you will appreciate these strengths after completing the rest of this tutorial.

A little history

Ant started out as an internal component of Tomcat, the servlet container that is used in the reference implementation for the Java Servlet and JavaServer Pages (JSP) technologies. The Tomcat codebase was donated to the Apache Software Foundation; there, it became part of the Apache Jakarta project, which has a mission to produce open source, server-side solutions for the Java platform. The usefulness of Ant was quickly recognized and its usage spread throughout the other Jakarta subprojects. Thus, it was made into a Jakarta subproject of its own, with the first independent release being made in July of 2000.

Since then, Ant's popularity has grown and grown. It has won numerous industry awards and become the *de facto* standard for building open source Java projects. In November of 2002, this success was acknowledged by Ant's promotion to a top-level Apache project.

At the time of writing, the current stable release of Ant is 1.5.4, which supports all JDK versions from 1.1 onwards. Beta versions of the next release, 1.6, are also available; these require JDK 1.2 or later. Planning is also underway for a future 2.0 release, which will involve a major rearchitecture. Ant 2.0 will feature improved consistency and enhanced functionality, whilst still retaining Ant's core goals of simplicity, ease of understanding, and extensibility.

© Copyright IBM Corporation 2003. All rights reserved.

Anatomy of an Ant build file

Ant does not define its own custom syntax; instead, its build files are written in XML (see Resources). There are a defined set of XML elements that Ant understands, and, as you'll see in a later section, new elements can be defined to extend Ant's capabilities. Every build file consists of a single top-level `project` element, which in turn contains one or more `target` elements. A target is a defined step in a build that performs any number of operations, such as compiling a set of source files. The operations themselves are performed by other specialized task tags, as we'll see later. These tasks are then grouped together into `target` elements, as desired. All of the operations required for a build could be placed into a single `target` element, but that would reduce flexibility. It is usually preferable to split the operations into logical build steps, with each step contained in its own `target` element. This allows you to perform one individual part of the overall build without necessarily performing other parts. For instance, by only invoking certain targets, you could compile your project's source code without creating an installable project image.

The top-level `project` element needs to contain a `default` attribute that specifies the target to be executed if none is specified when Ant is invoked. Then the target itself needs to be defined, using the `target` element. Here is a minimal build file:

```
<?xml version="1.0"?>
<project default="init">
    <target name="init">
    </target>
</project>
```

Note that this is well-formed XML, with an XML declaration specifying the version of XML being used (this is not currently required by Ant, but it is good practice), and with every element being properly closed. It is also possible to open and close an element in one go. So, instead of the separate start and end tags for the `target` element above, we could have written it as follows:

```
<target name="init"/>
```

This more concise form can be clearer when an element doesn't have any enclosed content.

Adding descriptions

The build file we saw in the previous section is nice and concise, but it doesn't say much about the actual project being built. There are a number of ways to make it more descriptive without changing the functionality. Here's an example:

© Copyright IBM Corporation 2003. All rights reserved.

Trademarks

```
<?xml version="1.0"?>
<project default="init" name="Project Argon">
     <description>
A simple project introducing the use of descriptive tags in Ant build files.
    </description>

    <!-- XML comments can also be used -->
    <target name="init" description="Initialize Argon database">
        <!-- perform initialization steps here -->
    </target>
</project>
```

As you can see, XML comments can be used throughout build files to improve clarity. Also, Ant defines its own `description` element and `description` attribute, which can be used to provide more structured annotations.

Properties

Properties in Ant are like programming language variables in that they have a name and a value. However, unlike normal variables, properties in Ant cannot be changed once they have been set; they are immutable, like `String` objects in the Java language. This might seem restrictive at first, but it is in keeping with Ant's philosophy of simplicity: after all, it is a build tool, not a programming language. If you do attempt to give an existing property a new value, this isn't treated as an error, but the property does retain its existing value. (This behavior can be useful, as we'll see.)

Based on the descriptive names of the elements and attributes seen so far, it should come as no surprise that the mechanism for setting properties in Ant looks like this:

```
<property name="metal" value="beryllium"/>
```

To refer to this property in other parts of the build file, you would use this syntax:

```
${metal}
```

For example, to use the value of one property as part of the value of another, you would write a tag that looks like this:

```
<property name="metal-database" value="${metal}.db"/>
```

There are number of predefined properties in Ant. Firstly, all of the system properties set by the Java environment used to run Ant are available as Ant properties, such as `${user.home}`. In addition to these, Ant defines a small set of its own properties, including `${ant.version}`, which contains the version of Ant, and `${basedir}`,

© Copyright IBM Corporation 2003. All rights reserved.

which is the absolute path of the project's directory (as defined by the directory containing the build file, or by the optional `basedir` attribute of the `project` element).

Properties are often used to refer to files and directories on the filesystem, but this could cause problems across different platforms with different path separator characters (/ versus \, for example). Ant's `location` attribute is specifically designed to hold filesystem paths in a platform-neutral way. You would use `location` in place of `value` like so:

```
<property name="database-file" location="archive/databases/${metal}.db"/>
```

The path separator characters used for the `location` attribute are converted to the correct ones for the current platform; and, as the filename is relative, it is taken to be relative to the project's base directory. We could just as easily have written this instead:

```
<property name="database-file" location="archive\databases\${metal}.db"/>
```

Both versions of this tag will behave in the same fashion across different platforms. The only thing to avoid if portability is required is DOS-style drive letters in filenames. It is also more flexible to use relative path names instead of absolute ones where possible.

Defining dependencies

Building a project generally requires a number of steps -- first compiling source code and then packaging it into Java Archive Files (JARs) files, for example. (See Resources for more information on using JARs.) Many of these steps have a clearly defined order -- for instance, you can't package the classfiles until they have been generated by the compiler from the source code. Instead of specifying the targets in sequence order, Ant takes the more flexible approach of defining *dependencies,* just as `make` and similar build tools do. Each target is defined in terms of all the other targets that need to be completed before it can be performed. This is done using the `depends` attribute of the `target` element. For example:

```
<target name="init"/>
<target name="preprocess" depends="init"/>
<target name="compile" depends="init,preprocess"/>
<target name="package" depends="compile"/>
```

This approach allows you to request a build of any stage of the project; Ant will execute the defined prerequisite stages first. In the example above, if you ask Ant to

© Copyright IBM Corporation 2003. All rights reserved.

complete the `compile` step, it determines that the targets `init` and `preprocess` need to be executed first. The `init` target doesn't depend on anything, so that will be executed first. Then Ant will look at the `preprocess` target and find that it depends on the `init` target; because it has already run that, it doesn't do so again, and executes the `preprocess` target. Finally, the `compile` task itself can be executed. Note that the order in which the targets appear in the build file is not important: it is only the `depends` attributes that determine execution order.

Section 3. Running Ant

Running Ant introduction

The Apache Ant tool can be invoked in various ways. On its own, Ant comes in command-line form, and is usually invoked from a UNIX or Linux shell prompt or a Windows command prompt, with the build files being written using your text editor of choice. This is fine if the project to be built is being developed in this way, but many find IDEs more convenient. Most of these offer some level of support for Ant, so at a minimum Ant builds can be invoked without the hassle of having to leave the IDE to perform command-line operations.

In this section, we'll see how to use Ant from the command line, and learn some useful command-line options. Then we'll take a brief tour of the Ant support offered by the open source Eclipse platform. (You should be at least passingly familiar with Eclipse in order to get the most out of those panels.)

Running Ant from the command line

Invoking Ant from your command prompt can be as simple as just typing **ant** on its own. If you do so, Ant uses the default build file; the default target specified in that build file is the one that Ant attempts to build. It is also possible to specify a number of command-line options, followed by any number of build targets, and Ant will build each of these targets in order, resolving any dependencies along the way.

Here is some typical output from a simple Ant build executed from the command line:

```
Buildfile: build.xml

init:
    [mkdir] Created dir: E:\tutorials\ant\example\build
```


© Copyright IBM Corporation 2003. All rights reserved.

```
    [mkdir] Created dir: E:\tutorials\ant\example\dist

compile:
    [javac] Compiling 8 source files to E:\tutorials\ant\example\build

dist:
      [jar] Building jar: E:\tutorials\ant\example\dist\example.jar

BUILD SUCCESSFUL
Total time: 2 seconds
```

We'll see just what all that means as we move on through the tutorial.

Command-line options

Just as the `make` tool looks for a build file with the name `makefile` by default, Ant looks for a file with the name `build.xml`. Therefore, if your build file uses this name, you don't need to specify it on the command line. Naturally, it is sometimes convenient to have build files with other names, in which case you need to use the `-buildfile <file>` arguments to Ant (`-f <file>` is the abbreviated version).

Another useful option is `-D`, which is used to set properties that can then be used in the build file. This is very useful for configuring the build you want to initiate in some way. For example, to set the `name` property to a particular value, you would use an option that looks something like this:

```
-Dmetal=beryllium
```

This capability can be used to override initial property settings in the build file. As noted earlier, you cannot change the value of a property once it has been set. The `-D` flag sets a property before any information in the build file is read; because the assignment in the build file comes after this initial assignment, it therefore doesn't change the value.

IDE integration

Due to Ant's popularity, most modern IDEs now have integrated support for it, and many others offer support for it in plugins. The list of supported environments includes the JEdit and Jext editors, Borland JBuilder, IntelliJ IDEA, the Java Development Environment for Emacs (JDEE), NetBeans IDE, Eclipse, and WebSphere (R) Studio Application Developer.

See Resources for further information on the wide-ranging IDE support for Ant. In the rest of this section, we will explore the Ant support included with the open source Eclipse environment.

© Copyright IBM Corporation 2003. All rights reserved.

Eclipse support for Ant

The open source Eclipse project offers a great deal of support for Ant. The core of this support is Eclipse's Ant editor, which features syntax highlighting. The editor is illustrated below:

```
Welcome    build.xml
<?xml version="1.0"?>
<project default="init" name="Project Argon">
     <description>
A simple project introducing the use of descriptive tags in Ant build files.
    </description>

    <!-- XML comments can also be used -->
    <target name="init" description="Initialize Argon database">
            <!-- perform initialization steps here -->
    </target>
</project>
```

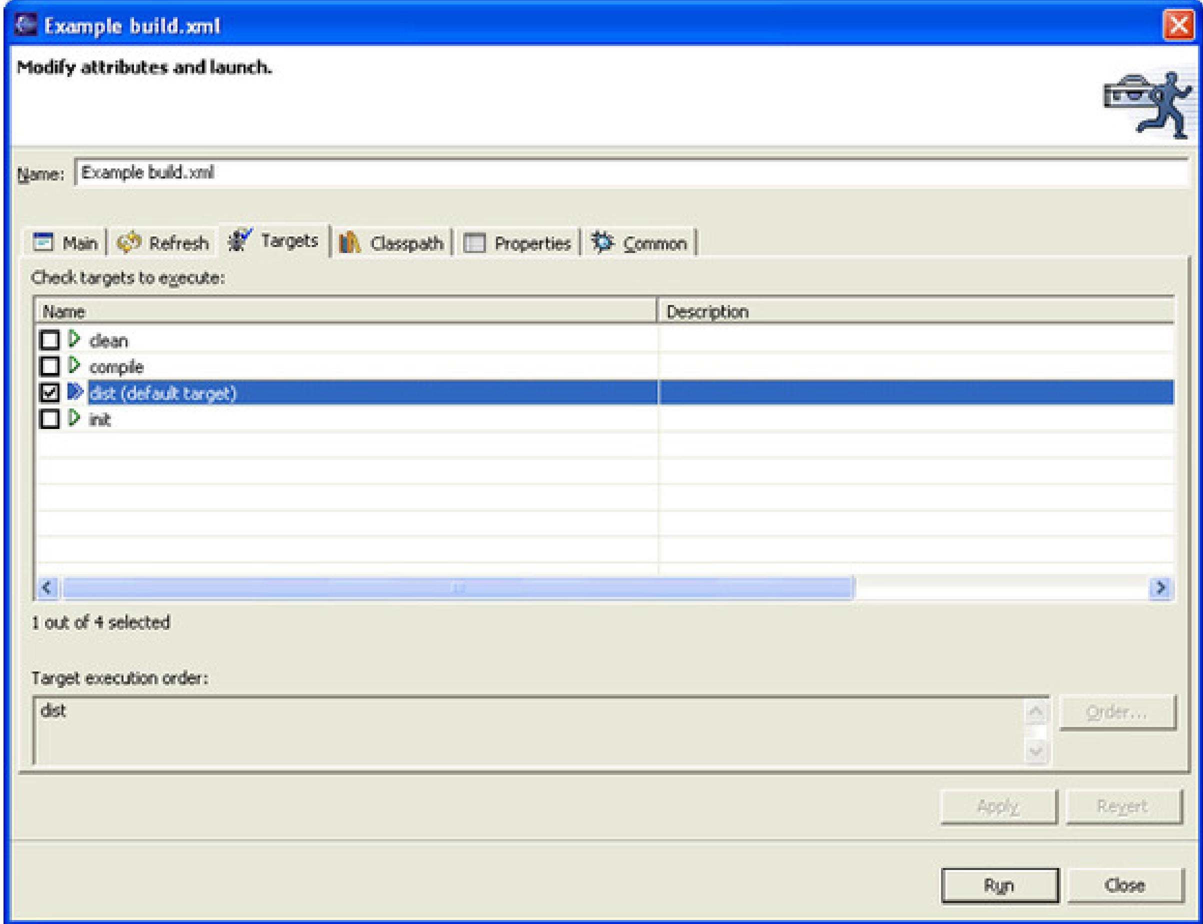
This editor provides content assistance -- for example, typing `<pro` and then pressing Ctrl-Space will show a completion list containing the `<property>` tag, and a short explanation of that task.

You can also see an outline view that shows the structure of the build file and provides a quick way of navigating through the file. There is also an Ant view that allows targets to be built from a number of different Ant files.

Running Ant builds from within Eclipse

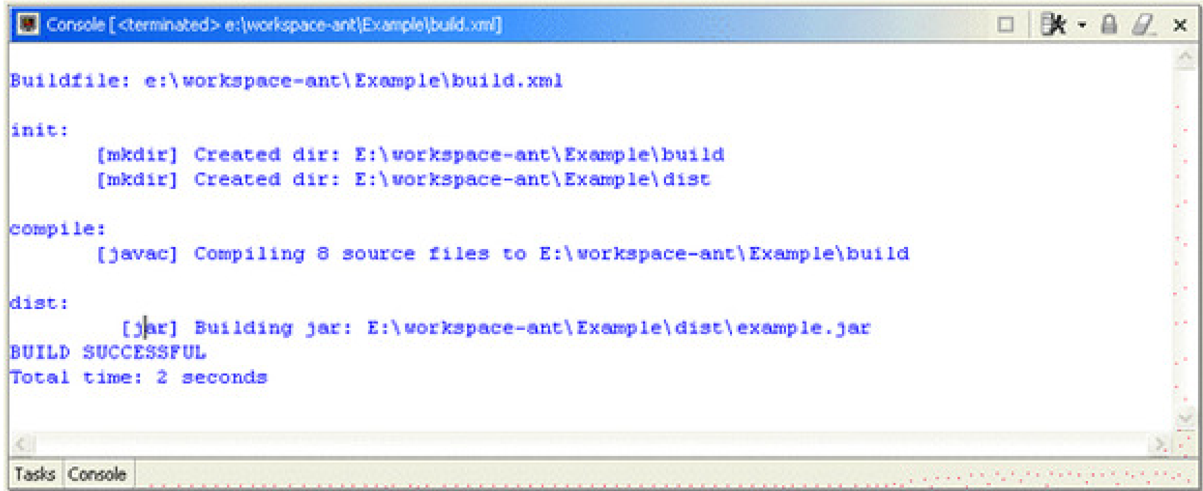
Files named build.xml are identified and decorated in Eclipse's navigator view with an Ant icon. (See File associations in Eclipse if your build file has a different filename.) Right-clicking these files provides a **Run Ant...** menu option that opens a dialog like the one shown below:

© Copyright IBM Corporation 2003. All rights reserved.



All of the targets from the build file are shown, with the defaults selected. After you've decided whether or not to change the default targets, press the **Run** button to launch Ant. Eclipse will switch to the Console view to show the output, as illustrated below. Errors are shown in a different color, and you can click on task names in the output to jump to the corresponding invocation in the build file.

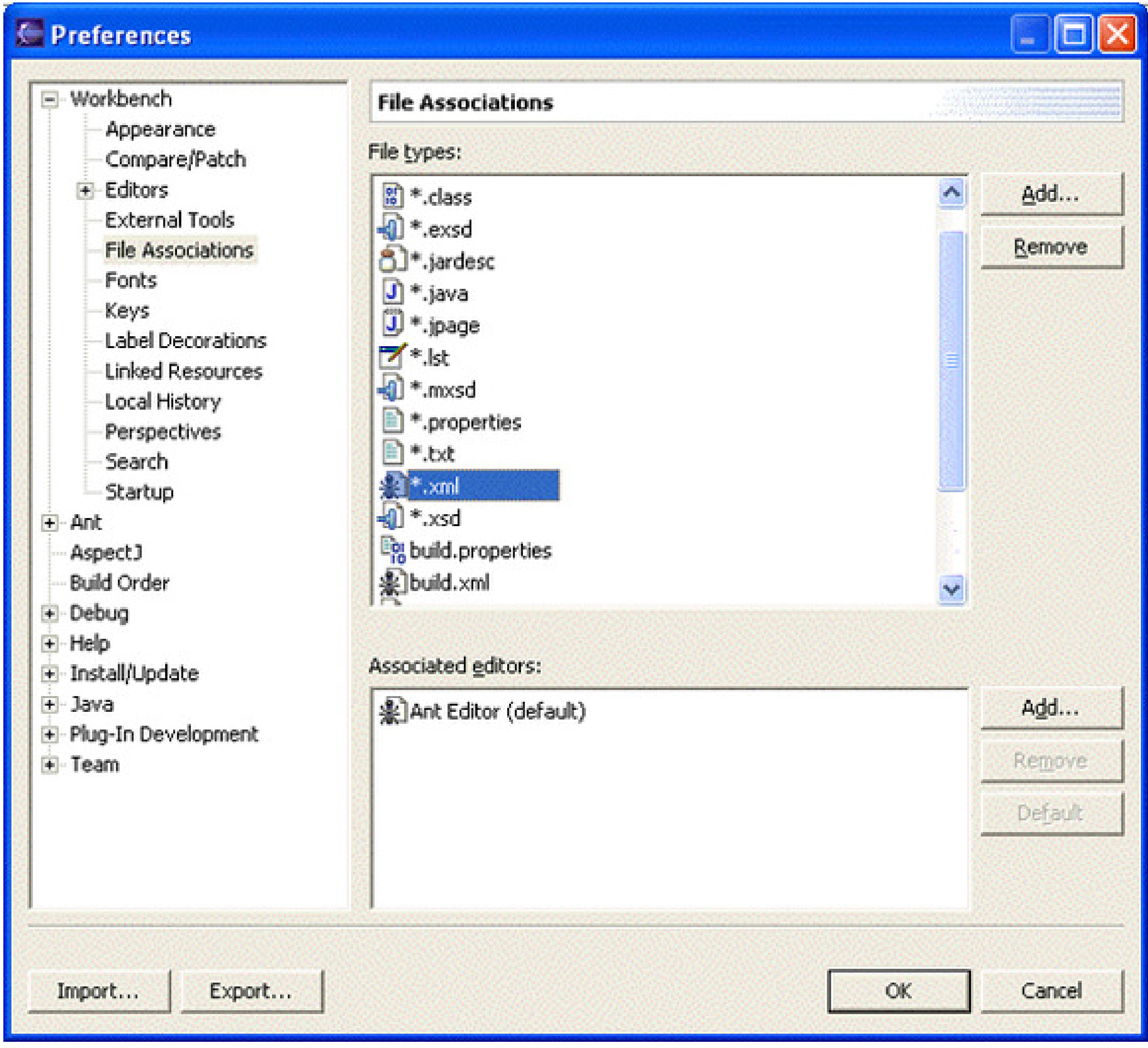
© Copyright IBM Corporation 2003. All rights reserved.



File associations in Eclipse

By default, Eclipse only uses the Ant editor on files named build.xml, but you can easily configure the editor to recognize files with other names. Select **Window=>Preferences** from the menu, then expand the **Workbench** group, and select the **File Associations** preference page. Then add a new file type for the desired filenames. You could, for example, add a new file type for all files named mybuild.xml. You could even use *.xml if you want to do the same for all files with an .xml suffix (except for specific filenames, such as plugin.xml, which in Eclipse override the wildcard specifications). Finally, add an associated editor for this new file type, and select **Ant editor** from the editor selection list, as shown below:

© Copyright IBM Corporation 2003. All rights reserved.



Section 4. Building a simple Java project

Building a simple Java project introduction

Now that we have seen the format of Ant build files, and learned how to define properties and dependencies and how to run Ant, we are ready to construct a build environment for a basic Java project. This will involve learning the Ant tasks for compiling source code and assembling JAR files.

© Copyright IBM Corporation 2003. All rights reserved.

Compiling source code

As Ant is primarily aimed at building Java applications, it should come as no surprise that it has excellent built-in support for invoking the `javac` compiler and other Java-related tasks. Here's how to write a task that compiles some Java code:

```
<javac srcdir="src"/>
```

This tag looks for all files ending in .java in the `src` directory and invokes the `javac` compiler on them, generating the classfiles in the same directory. Of course, it is usually cleaner to put the classfiles under a separate directory structure; you can have Ant do this by adding a `destdir` attribute. Other useful attributes:

- `classpath`: Equivalent to the `-classpath` option on `javac`.
- `debug="true"`: Indicates to the compiler that the source files should be compiled with debug information.

An important characteristic of the `javac` task is that it will only compile those source files that it believes it needs to. If a classfile already exists, and that classfile's corresponding source file hasn't been changed since the classfile was generated, then that source file won't be recompiled. The output of the `javac` task shows the number of source files that were actually compiled. It is good practice to write a `clean` target that removes any generated classfiles from that target directory. This can then be used if you want to make sure that all of the source files are compiled. This behavior characterizes many of Ant's tasks: If the task can determine that the requested operation doesn't need to be performed, then it will be skipped.

Like Ant, the `javac` compiler is itself implemented in the Java language. This is used to good advantage by the `javac` task in Ant, because it usually invokes the compiler classes in the same Java virtual machine (JVM) in which Ant itself is running in. Other build tools would typically need to launch a new `javac` process each time Java code needed to be compiled, which would in turn require a new JVM instance. But with Ant, only a single JVM instance is required, both to run Ant itself and to perform all of the required compilation tasks (along with other related tasks, such as manipulating JAR files). This is a far more efficient use of resources and can lead to greatly improved build times.

Compiler options

As we saw in the previous section, the default behavior of the Ant `javac` task is to invoke the standard compiler of whichever JVM is being used to run Ant itself, and to invoke that compiler in-process (that is, without launching a separate JVM).

© Copyright IBM Corporation 2003. All rights reserved.

However, there may be times when you want the compiler to be invoked separately -- when you wish to specify certain memory options to the compiler, for instance, or when you need to use a different level of the compiler. To achieve this, simply set `javac` 's `fork` attribute to `true`, like this:

```
<javac srcdir="src" fork="true"/>
```

And if you want to specify a different `javac` executable, and pass a maximum memory setting to it, you would do something like this:

```
<javac srcdir="src" fork="true" executable="d:\sdk141\bin\javac"
memoryMaximumSize="128m"/>
```

You can even configure Ant to use a different compiler altogether. Supported compilers include the open source Jikes compiler and the GCJ compiler from the GNU Compiler Collection (GCC). (See Resources for information on these two compilers.) These can be specified in two ways: either by setting the `build.compiler` property, which will apply to all uses of the `javac` task, or by setting the `compiler` attribute in each `javac` task as required.

The `javac` task supports many other options. Consult the Ant manual for more details (see Resources).

Creating a JAR file

After Java source files have been compiled, the resulting classfiles are typically packaged into a JAR file, which is similar to a zip archive. Every JAR file contains a manifest file that can specify properties of the JAR file.

Here's a simple use of the `jar` task in Ant:

```
<jar destfile="package.jar" basedir="classes"/>
```

This creates a new JAR file called `package.jar` and adds all the files in the `classes` directory to it (JAR files can contain any type of file, not just classfiles). In this case, no manifest file was specified, so Ant will provide a basic one.

The `manifest` attribute allows you to specify a file to use as the manifest for the JAR file. The contents of a manifest file could also be specified within the build file using the `manifest` task. This task can write a manifest file to the filesystem, or can actually be nested inside the `jar` task so that the manifest file and JAR file are created all in one go. For example:

© Copyright IBM Corporation 2003. All rights reserved.

```
<jar destfile="package.jar" basedir="classes">
  <manifest>
    <attribute name="Built-By" value="${user.name}"/>
    <attribute name="Main-class" value="package.Main"/>
  </manifest>
</jar>
```

Time-stamping builds

It is often desirable in a build environment to use the current time and date to mark the output of a build in some way, in order to record when it was built. This might involve editing a file to insert a string to indicate the date and time, or incorporating this information into the filename of the JAR or zip file.

This need is addressed by the simple but very useful `tstamp` task. This task is typically called at the start of a build, such as in an `init` target. No attributes are required, so just `<tstamp/>` is sufficient is many cases.

The `tstamp` task doesn't produce any output; instead, it sets three Ant properties based on the current system time and date. Here are the properties that `tstamp` sets, an explanation for each, and examples of what they might be set to:

Property	Explanation	Example
`DSTAMP`	This is set to the current date, the default format being `yyyymmdd`	20031217
`TSTAMP`	This is set to the current time, the default format being `hhmm`	1603
`TODAY`	This is set to a string describing the current date, with the month written in full	December 17 2003

For example, in the previous section we created a JAR file as follows:

```
<jar destfile="package.jar" basedir="classes"/>
```

After having called the `tstamp` task, we could name the JAR file according to the date, like this:

```
<jar destfile="package-${DSTAMP}.jar" basedir="classes"/>
```

So, if this task were invoked on December 17, 2003, the JAR file would be named `package-20031217.jar`.

© Copyright IBM Corporation 2003. All rights reserved.

The `tstamp` task can also be configured to set different properties, apply an offset before or after the current time, or format the strings differently. All of this is done using a nested `format` element, as follows:

```
<tstamp>
   <format property="OFFSET_TIME"
           pattern="HH:mm:ss"
           offset="10" unit="minute"/>
</tstamp>
```

The listing above sets the `OFFSET_TIME` property to a time in hours, minutes, and seconds that is 10 minutes after the current time.

The characters used to define the format strings are the same as those defined by the `java.text.SimpleDateFormat` class.

Putting it all together

The previous sections have given us enough knowledge to build a simple Java project. We will now assemble the code snippets into a complete build file that compiles all the source code under a `src` directory, puts the resulting classfiles under a `build` directory, and then packages everything up into a JAR file in a `dist` directory, ready for distribution. In order to try this out for yourself, all you need is a `src` directory containing one or more Java source code files -- anything from a simple "Hello World" program to a large number of source files from an existing project. If you need to add JAR files or anything else to the Java classpath in order to successfully compile your source code, simply add a `classpath` attribute for it in the `javac` task.

The build file looks like this:

```
<?xml version="1.0"?>
<project default="dist" name="Project Argon">
     <description>A simple Java project</description>

    <property name="srcDir" location="src"/>
    <property name="buildDir" location="build"/>
    <property name="distDir" location="dist"/>

    <target name="init">
       <tstamp/>
       <mkdir dir="${buildDir}"/>
       <mkdir dir="${distDir}"/>
    </target>

    <target name="compile" depends="init">
       <javac srcdir="${srcDir}" destdir="${buildDir}"/>
    </target>

    <target name="dist" depends="compile">
       <jar destfile="${distDir}/package-${DSTAMP}.jar" basedir="${buildDir}">
```


© Copyright IBM Corporation 2003. All rights reserved.

```
          <manifest>
            <attribute name="Built-By" value="${user.name}"/>
            <attribute name="Main-Class" value="package.Main"/>
          </manifest>
        </jar>
        <jar destfile="${distDir}/package-src-${DSTAMP}.jar" basedir="${srcDir}"/>
    </target>

    <target name="clean">
      <delete dir="${buildDir}"/>
      <delete dir="${distDir}"/>
    </target>
</project>
```

And here is some sample output from a build that was run with that file (yours may vary, depending on the contents of your `src` directory):

```
Buildfile: build.xml

init:
    [mkdir] Created dir: E:\tutorial\javaexample\build
    [mkdir] Created dir: E:\tutorial\javaexample\dist

compile:
    [javac] Compiling 10 source files to E:\tutorial\javaexample\build

dist:
      [jar] Building jar: E:\tutorial\javaexample\dist\package-20031217.jar
      [jar] Building jar: E:\tutorial\javaexample\dist\package-src-20031217.jar

BUILD SUCCESSFUL
Total time: 5 seconds
```

Notice that the JAR file is named according to the current date, and that a manifest entry is set for the main class of the application, so that it can be launched directly with a command along the lines of `java -jar package-20031217.jar`. We also create a JAR file consisting of just the source code of the project.

Section 5. Filesystem operations

Filesystem operations introduction

We've learned enough about Ant now to be able to build a basic Java project, but naturally real-world projects are rarely as simple as our example. In the next few sections, we'll look some of Ant's numerous additional capabilities and the situations in which they can be used.

In this section, we'll see how to perform common file operations, such as creating

© Copyright IBM Corporation 2003. All rights reserved.

directories and unpacking files. One of the good features of Ant is that the tasks to perform these operations generally behave the same on all platforms.

Creating and deleting directories

One of the most basic filesystem operations is the creation of a directory or folder. The Ant task for this is called `mkdir`, and it is unsurprisingly very similar to the Windows and UNIX/Linux commands of the same name:

```
<mkdir dir="archive/metals/zinc"/>
```

Note first the use of `/` as the directory separator, which is the convention for UNIX and Linux platforms. You might think that this isn't very platform neutral, but Ant knows how to handle it, and will do the right thing for whichever platform it is running on, in the same way as we saw earlier when defining location-based properties. We could just as easily have used `\` instead, regardless of platform -- Ant handles either form, or even a mixture of both.

Another useful feature of the `mkdir` task is its ability to create parent directories if they don't exist already. Consider the listing above, and imagine that the `archive` directory exists, but not the `metals` directory. If you were using the underlying platform's `mkdir` commands, you would need to first explicitly create the `metals` directory, and then create the `zinc` directory using a second invocation of the `mkdir` command. But the Ant task is smarter than this, and will create both directories in one go. Similarly, if the target directory already exists, the `mkdir` task will not complain, but will just assume that its work is already done, and do nothing.

Deleting directories is just as easy:

```
<delete dir="archive/metals/zinc"/>
```

This will delete the specified directory, along with any files and sub-directories it contains. Use the `file` attribute instead of `dir` to specify a single file to be deleted.

Copying and moving files and directories

Making a copy of a file is simple in Ant. For example:

```
<copy file="src/Test.java" tofile="src/TestCopy.java"/>
```

You can also use `move` to perform a rename operation instead of copying the file:

© Copyright IBM Corporation 2003. All rights reserved.

```
<move file="src/Test.java" tofile="src/TestCopy.java"/>
```

Another common filesystem operation is to copy or move a file into another directory. The Ant syntax for this is just as straightforward:

```
<copy file="src/Test.java" todir="archive"/>

<move file="src/Test.java" todir="archive"/>
```

By default, Ant only outputs a summary of the move and copy operations it performs, including such information as the number of files it has moved or copied. If you wish to see more detailed information, including the names of the files involved, you can set the `verbose` attribute to `true`.

Creating and unpacking zip and tar files

In the previous section, we saw how to create a JAR file. The process is almost exactly the same for creating other archive files. Here is an Ant task that creates a zip file instead:

```
<zip destfile="output.zip" basedir="output"/>
```

The same syntax can also be used to create tar files. Files can also be compressed using the GZip and BZip2 tasks. For example:

```
<gzip src="output.tar" zipfile="output.tar.gz"/>
```

Uncompressing and extracting files is just as straightforward:

```
<unzip src="output.tar.gz" dest="extractDir"/>
```

The `overwrite` attribute can also be included to control the overwrite behavior. The default is to overwrite any existing files with matching entries from the archive being extracted. The related task names are `untar`, `unjar`, `gunzip`, and `bunzip2`.

Replacing tokens in files

The final filesystem operation we will look at in this section is the `replace` task, which performs search and replace operations within files. The `token` attribute specifies the string to be searched for, and the `value` attribute specifies the new

© Copyright IBM Corporation 2003. All rights reserved.

string that all occurrences of the token string should be replaced with. For example:

```
<replace file="input.txt" token="old" value="new"/>
```

The replacement occurs in place, within the file itself. To provide more detailed output, set the `summary` attribute to `true`. This causes the task to output the number of occurrences of the token string that were found and replaced.

Section 6. Useful tasks and techniques

Useful tasks and techniques introduction

Before looking at custom tasks, we'll first cover some useful functionality that we haven't already encountered. There is a great deal of functionality that comes standard with Ant, so we can only pick out a few of the most useful pieces here. Pattern matching and file selectors are powerful mechanisms that greatly enhance the capabilities of some of the tasks we've already seen, and chaining builds together and working with CVS repositories are two areas that have been found to be of great practical use.

Pattern matching

When we looked at filesystem tasks earlier, we used only individually named files and directories. However, it is often useful to perform those operations on a group of files at once -- on all files in a given directory that end with `. java`, for instance. Just as the equivalent DOS and UNIX commands provide this sort of functionality, so does Ant. This is done using wildcard characters: `*`, which matches zero or more characters, and `?` , which matches exactly one character. The pattern to match all files ending with .java would therefore be simply `*.java`.

Matching is also performed on directories. For example, the pattern `src*/*.java` would match all Java files in any directories with a prefix of `src`. There is one additional pattern construct: `**`, which matches any number of directories. For example, the pattern `**/*.java` would match all Java files under the current directory structure.

You can make use of patterns with filesystem tasks in a fairly consistent way, such as with a nested `fileset` element. Previously, we copied a single file with this task:

© Copyright IBM Corporation 2003. All rights reserved.

```
<copy file="src/Test.java" todir="archive"/>
```

If we wanted to use a pattern instead, we could replace the `file` attribute with a `fileset` element, like this:

```
<copy todir="archive">
  <fileset dir="src">
    <include name="*.java"/>
  </fileset>
</copy>
```

The `fileset` would by default consist of all the files under the specified `src` directory, so to select just the Java files, we use the `include` element with a pattern. In a similar way, we could add an `exclude` element with another pattern, which would potentially remove matches from the `include` specification. We could even specify multiple `include` and `exclude` elements; this would result in a set of files and directories consisting of all the matches from the `include` patterns joined together, with all of the matches from the `exclude` elements removed.

Note that there is one feature of filesets that is usually very helpful, but occasionally leads to confusion for those who are not aware of it. This feature is known as *default excludes*: a built-in list of patterns that are automatically excluded from the contents of filesets. The list includes entries to match directories called `CVS`, and files ending with the ~ character, which may be backup files. You often don't want to include these sorts of files and directories in filesystem operations, so that is the default behavior. However, if you really want to select *all* files and directories without exception, you can set the `defaultexcludes` attribute of your fileset to `no`.

Using selectors

As we've seen, a fileset is used to specify a group of files, and the contents of this group can be defined using `include` and `exclude` patterns. You can also use special elements called *selectors* in conjunction with the `include` and `exclude` to select files. Here is a list of the core selectors available with Ant:

- `size`: This is used to select files based on their size in bytes (unless the `units` attribute is used to specify a different unit). The `when` attribute is used to set the nature of the comparison -- `less`, `more`, or `equal` -- and the `value` attribute defines the target size against which the size of each file is compared.
- `contains`: Only files containing the given text string (specified by the `text` attribute) match this selector. By default, the search is case sensitive; adding `casesensitive="no"` changes this.

© Copyright IBM Corporation 2003. All rights reserved.

- `filename`: The `name` attribute specifies the pattern to match filenames against. This is essentially the same as the `include` element, and the same as the `exclude` element when `negate="yes"` is specified.
- `present`: Selects those files from the current directory structure that have the same name and relative directory structure as those in a specified `targetdir` directory.
- `depend`: This selector has the same effect as the `present` selector, except that the matching files are restricted to those that have been modified more recently than the corresponding files in the `targetdir` location.
- `date`: This selects files based on the date on which they were last modified. The `when` attribute specifies whether the comparison is `before`, `after`, or `equal`, and the `datetime` attribute specifies the date and time to compare against, given in a fixed format of `MM/DD/YYYY HH:MM AM_or_PM`. Note that on Windows platforms there is a built-in margin of 2 seconds either way to allow for inaccuracies in the underlying filesystem -- this may cause more files to match than otherwise expected. The amount of leeway allowed can be changed with the `granularity` attribute (specified in milliseconds).
- `depth`: This selector looks at the number of levels in the directory structure of each file. Attributes for `min` and/or `max` are used to select files with a desired number of directory levels.

Selectors can also be combined together by nesting one or more selectors inside a selector *container.* The most common selector container, `and`, selects only those files that are selected by all of the selectors it encloses. Other selector containers include `or`, `not`, `none`, and `majority`.

Here is an example of a fileset that selects only those files that are larger than 512 bytes and also contain the string "hello":

```
<fileset dir="dir">
    <and>
        <contains text="hello"/>
        <size value="512" when="more"/>
    </and>
</fileset>
```

Chaining builds together

There are two different approaches to building large projects. One is to have a single monolithic build file that does everything; the other is to split the build up into a number of smaller pieces by having high-level build files that call out to other build

© Copyright IBM Corporation 2003. All rights reserved.

files to perform specific tasks.

It is easy to call one Ant build from another using the `ant` task. In a simple case, you can specify just the build file to use, using the `antfile` attribute, and Ant will build the default target in that build file. For example:

```
<ant antfile="sub-build.xml"/>
```

Any properties defined in the parent build file will by default be passed into the sub-build, although this can be avoided by specifying `inheritAll="false"`. It is also possible to pass in explicit properties by using a nested `property` element -- these still apply even if `inheritAll` is set to `false`. This facility is good for passing in parameters to sub-builds.

Let's consider an example. Here is a build file that we wish to call:

```
<?xml version="1.0"?>
<project default="showMessage">
    <target name="showMessage">
        <echo message="Message=${message}"/>
    </target>
</project>
```

(We haven't come across the `echo` task before -- it simply outputs the given message.)

And here is a second build file that calls out to the first, passing in the `message` property:

```
<?xml version="1.0"?>
<project default="callSub">
    <target name="callSub">
        <ant antfile="sub.xml" target="showMessage" inheritAll="false">
            <property name="message" value="Hello from parent build"/>
        </ant>
    </target>
</project>
```

The output from running the second build file is as follows:

```
Buildfile: build.xml

callSub:

showMessage:
     [echo] Message=Hello from parent build

BUILD SUCCESSFUL
Total time: 0 seconds
```


© Copyright IBM Corporation 2003. All rights reserved.

Working with CVS repositories

CVS is an acronym for *concurrent versions system.* It is a source code control system that is designed to keep track of changes made by a number of different developers. It is extremely popular, particularly with open source projects. Ant provides close integration with CVS. This is very useful for automated build environments, as a single build file can extract one or more modules from the source code repository, build the project, and even generate patch files based on the changes that have been made since the previous build.

Note that in order to make use of the `cvs` task in Ant, you need to have the `cvs` command installed on your machine and available from the command line. This command is included in most Linux distributions; it is also available for Windows in several forms, -- as part of the invaluable Cygwin environment, for instance. (See Resources for more on Cygwin.)

Here is an example build file that extracts a module from a CVS repository:

```
<?xml version="1.0"?>
<project name="CVS Extract" default="extract" basedir=".">
<property name="cvsRoot" value=":pserver:anonymous@dev.eclipse.org:/home/eclipse"/>
    <target name="extract">
        <cvs cvsRoot="${cvsRoot}"
             package="org.eclipse.swt.examples"
                dest="${basedir}"/>
    </target>
</project>
```

The main attribute to the `cvs` task is `cvsRoot`, which is the complete reference to the CVS repository, including connection method and user details. The format of this parameter is:

```
[:method:][[user][:password]@]hostname[:[port]]/path/to/repository
```

In the above example, we connect to the central repository for the Eclipse project as an anonymous user. The other attributes then specify the module or package we wish to extract and the destination for the extracted files. Extraction is the default operation for the CVS task; other operations can be specified using the `command` attribute.

See Resources for more information on CVS.

© Copyright IBM Corporation 2003. All rights reserved.

Section 7. Extending Ant with custom tasks

Extending Ant with custom tasks introduction

As we've seen throughout the previous sections, Ant is very powerful, with a number of core tasks covering a broad set of functionality. There are a number of additional core tasks not covered here, plus a number of optional tasks providing a wide variety of additional functionality, as well as other tasks that are available as part of the Ant-Contrib project; finally, there are even more externally available tasks listed on the Apache Ant home page. Given all of this, it might seem unlikely that you'd ever need any other tasks, but the real power of Ant lies in its ease of extensibility. In fact, it is precisely this extensibility that has resulted in such a large number of additional tasks being developed.

There can be occasions when creating your own custom task is a good idea. For example, imagine that you have created a command-line tool to perform a particular operation; this tool might be a good candidate to be made available as an Ant task (particularly if the tool was written in the Java language, although this doesn't have to be the case). Instead of having Ant call the tool externally using the `exec` task (which introduces dependencies and makes it harder to use your build file across different environments), you can incorporate it directly into the build file. The regular fileset and wildcard matching capabilities of Ant can also be made available to your custom task.

In this section, we'll take a look at the construction of a simple custom task. This task will perform a sort operation on the lines of a file, and write the sorted set of lines out to a new file.

Creating a custom task

To implement a simple custom task, all we need to do is extend the `org.apache.tools.ant.Task` class and override the `execute()` method. So, as a skeleton of our file sorter custom task, we have the following:

```
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class FileSorter extends Task {
        // The method executing the task
        public void execute() throws BuildException {}
}
```


© Copyright IBM Corporation 2003. All rights reserved.

Notice that the `execute()` method is declared to throw a `BuildException`. If anything goes wrong with our task, we throw this to signal the failure back to Ant.

Most tasks, core and custom alike, make use of attributes to control their behavior. For our simple task, we need an attribute to specify the file to be sorted, and another to specify the output file for the sorted contents. Let's call these attributes `file` and `tofile`, respectively.

Ant makes it very easy to support attributes in custom tasks. We do this simply by implementing a method with a specifically formatted name that Ant can call with the values of the corresponding attribute specified in the build file. The name of the method needs to be `set` plus the name of the attribute, so in our case we need methods called `setFile()` and `setTofile()`. When Ant encounters an attribute setting in the build file, it looks for a method of the appropriate name, known as a *setter* method, in the associated task.

Attributes in the build file are specified as strings, so the argument to our setter methods can be a string. In such a case, Ant will call our method with the string value of the attribute, after expanding any properties referenced by the value. But sometimes we want to treat the value of the attribute as a different type. This is true for our sample task, where the attribute values refer to files on the filesystem rather than just arbitrary strings. We can do this very easily by declaring our method argument to be of type `java.io.File`. Ant will take the string value of the attribute and interpret it as a file, and then pass this to our method. If the file is specified with a relative path name, this will be converted into an absolute path name relative to the project's base directory. Ant can perform similar conversions for other types, such as `boolean` s and `int` s. If you provide two methods with the same name but with different argument types, Ant will use the more specific one, so a file type will be preferred to a string type.

The two setter methods we need for our custom task look like this:

```
    // The setter for the "file" attribute
    public void setFile(File file) {}

    // The setter for the "tofile" attribute
    public void setTofile(File tofile) {}
```

Implementing a custom task

Using the skeleton developed in the previous section, we can now complete the implementation of our simple file sorter task:

```
import java.io.*;
import java.util.*;
```


© Copyright IBM Corporation 2003. All rights reserved.

```
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

/**
 * A simple example task to sort a file
 */
public class FileSorter extends Task {
    private File file, tofile;

    // The method executing the task
    public void execute() throws BuildException {
        System.out.println("Sorting file="+file);
        try {
            BufferedReader from =
                new BufferedReader(new FileReader(file));
            BufferedWriter to =
                new BufferedWriter(new FileWriter(tofile));
            List allLines = new ArrayList();
            // read in the input file
            String line = from.readLine();
            while (line != null) {
                allLines.add(line);
                line = from.readLine();
            }
            from.close();
            // sort the list
            Collections.sort(allLines);
            // write out the sorted list
            for (ListIterator i=allLines.listIterator(); i.hasNext(); ) {
                String s = (String)i.next();
                to.write(s); to.newLine();
            }
            to.close();
        } catch (FileNotFoundException e) {
            throw new BuildException(e);
        } catch (IOException e) {
            throw new BuildException(e);
        }
    }

    // The setter for the "file" attribute
    public void setFile(File file) {
        this.file = file;
    }
    // The setter for the "tofile" attribute
    public void setTofile(File tofile) {
        this.tofile = tofile;
    }
}
```

The two setter methods simply store the values of the attributes so they can be used in the `execute()` method. Here, the input file is read line by line into a list, which is then sorted and written out line by line to the output file. Note that to keep things simple we perform very little error checking -- for example, we don't even check that the required attributes have actually been set by the build file. We do at least catch any I/O exceptions from the operations performed, and rethrow these as Ant `BuildException`s.

We can now compile our custom task with the `javac` compiler, or from within an IDE. In order to resolve the Ant classes we have used, you need to add the location of the ant.jar file to your classpath. This is should be in the `lib` directory of your Ant installation.

© Copyright IBM Corporation 2003. All rights reserved.

Using a custom task

Now that we've developed and compiled our custom task, we are in a position to make use of it from a build file.

Before we can call our custom task, we need to *define* it by giving it a name, and telling Ant the classfile that implements it and any classpath setting required to locate that classfile. This is done using the `taskdef` task, like so:

```
<taskdef name="filesorter"
    classname="FileSorter"
    classpath="."/>
```

That's it! The task can now be used in the same way as Ant's core tasks. Here is a complete build file, showing the definition and use of our custom task:

```
<?xml version="1.0"?>
<project name="CustomTaskExample" default="main" basedir=".">
    <taskdef name="filesorter"
        classname="FileSorter"
       classpath="."/>
    <target name="main">
        <filesorter file="input.txt" tofile="output.txt"/>
    </target>
</project>
```

Now, create an `input.txt` file in the present working directory to test the custom task. For example:

```
Hello there
This is a line
And here is another one
```

Here is the console output after running the above build file:

```
Buildfile: build.xml

main:
[filesorter] Sorting file=E:\tutorial\custom\input.txt

BUILD SUCCESSFUL
Total time: 0 seconds
```

Notice that our relative pathname of `input.txt` gets converted into an absolute pathname in the current directory. This is because we specified the argument to the setter method to be of type `java.io.File` instead of `java.lang.String`.

© Copyright IBM Corporation 2003. All rights reserved.

Now let's see if the task actually worked. A file called `output.txt` should have been created in the same directory with the following contents:

```
And here is another one
Hello there
This is a line
```

You might try specifying an input file that doesn't exist to see how a "file not found" exception is reported back to Ant.

Congratulations: you have now developed and used a custom Ant task! There are many additional aspects involved in creating more complex tasks, and Resources contains links to sources of further information in this area.

Section 8. Wrap-up

Summary

We hope you have found this tour of Ant useful. Whilst Ant aims to be as simple and straightforward as possible, it provides a great deal of functionality through a large number of tasks, each with a number of options, which can be overwhelming at times. We haven't been able to explore all of Ant's functionality in this single tutorial, but we hope to have introduced all of the fundamental concepts and enough of the basic functionality to start you down the road of using Ant on real-world projects. Here is a summary of what we've covered:

- How Ant build files are structured
- How to run Ant from the command line and from Eclipse
- How to build simple Java projects by compiling source code, creating JAR files, and time-stamping files to identify the output of each build
- How to perform basic filesystem operations in Ant
- The basic concepts of pattern matching and selectors, plus how to call one build from another, and how to perform CVS operations
- How to extend Ant's standard capabilities by writing Java classes

To continue your Ant journey, follow some of the links on the next section, and familiarize yourself with the Ant manual, which is the definitive source of reference

© Copyright IBM Corporation 2003. All rights reserved.

when writing Ant build files. Good luck!

© Copyright IBM Corporation 2003. All rights reserved.

Resources

- Visit the Apache Ant home page. It's a great source of information, and it includes downloads, the Ant manual, and links to other resources.
- Find out more about Ant's IDE and editor integration.
- Learn more about Eclipse, a universal tool platform.
- The *developerWorks* Open source projects zone has a wealth of Eclipse-based content.
- If you're new to XML, visit the *developerWorks* XML zone.
- Cygwin is a Linux-like environment for Windows.
- CVS is the open standard for version control.
- Learn more about WebSphere Studio Application Developer, which is based on the Eclipse platform.
- Find out more about WebSphere Application Server's support for Ant tasks.
- Explore the power of the JAR file format in " JAR files revealed " (*developerWorks*, October 2003).
- *developerWorks* features a number of articles on Ant:
  - " Incremental development with Ant and JUnit, " Malcolm Davis, (November 2000).
  - " Extending Ant to support interactive builds, " Anthony Young-Garner (November 2001).
  - " Enhance Ant with XSL transformations, " Jim Creasman (September 2003).
- Ant also supports the open source Jikes compiler and the GNU Compiler for Java (GCJ).
- Find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.
- Also see the Java technology zone tutorials page for a complete listing of free Java-related tutorials from *developerWorks*.

About the author

Matt Chapman

© Copyright IBM Corporation 2003. All rights reserved.

Matt Chapman is an advisory software engineer in the IBM Java Technology Centre in Hursley, U.K. He has spent the last seven years working with Java technology, including Java virtual machine implementations for a variety of platforms, the user interface toolkits Swing and AWT, and, more recently, tooling for the Eclipse platform. Matt has a degree in computer science and is also a Sun-certified Java programmer. You can contact him at mchapman@uk.ibm.com.

© Copyright IBM Corporation 2003. All rights reserved.