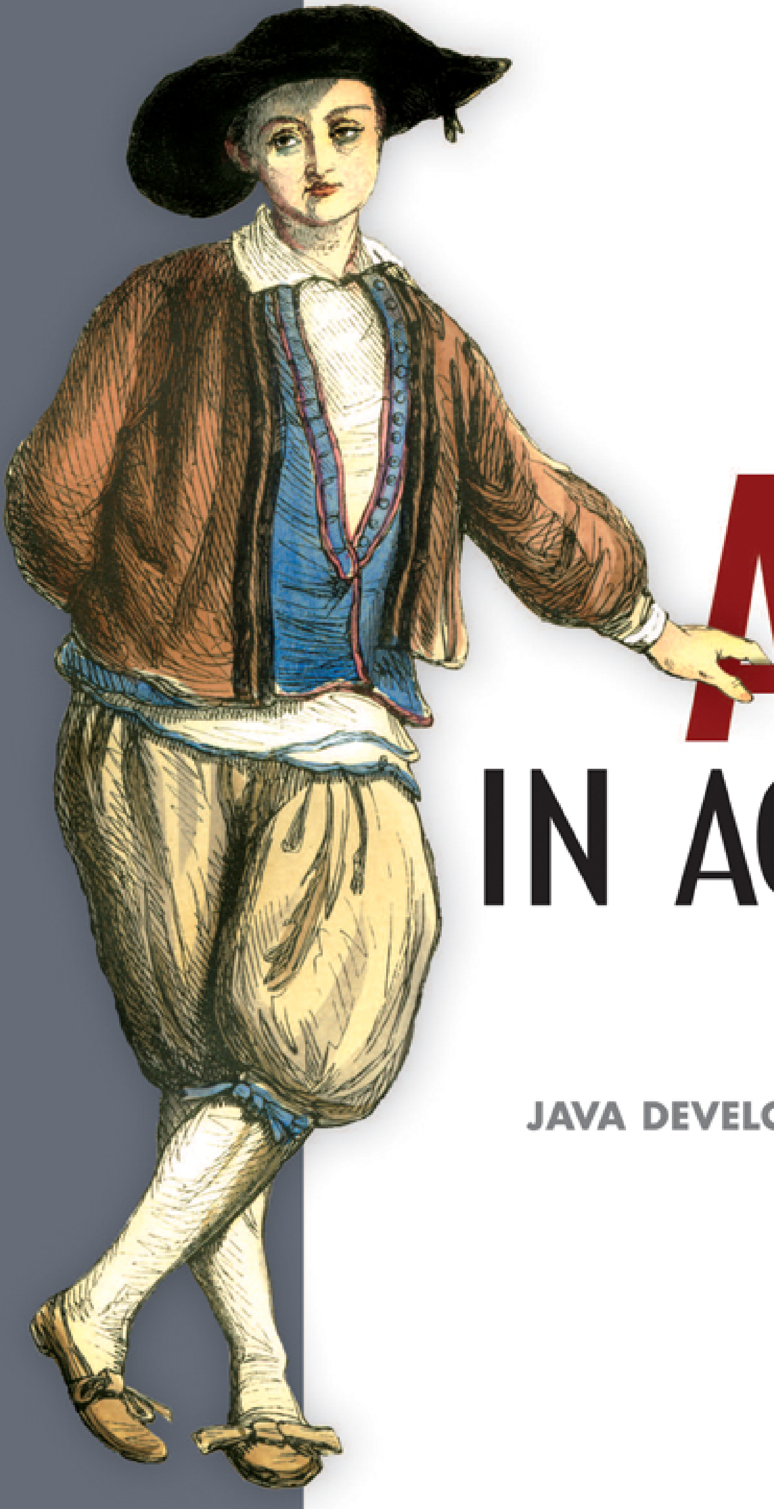
COVERS ANT 1.7

SAMPLE CHAPTER



Steve Loughran
Erik Hatcher

ANT IN ACTION

Second Edition of
JAVA DEVELOPMENT WITH ANT

MANNING

Ant in Action

Steve Loughran and Erik Hatcher

Sample Chapter 2

brief contents

CHAPTER 2

A first Ant build

Let's start this gentle introduction to Ant with a demonstration of what it can do. The first chapter described how Ant views a project: a project contains targets, each of which is a set of actions—tasks—that perform part of the build. Targets can depend on other targets, all of which are declared in an XML file, called a *build file*.

This chapter will show you how to use Ant to compile and run a Java program, introducing Ant along the way.

2.1 DEFINING OUR FIRST PROJECT

Compiling and running a Java program under Ant will introduce the basic concepts of Ant—its command line, the structure of a build file, and some of Ant's tasks.

Table 2.1 shows the steps we will walk though to build and run a program under Ant.

The program will not be very useful, but it will introduce the basic Ant concepts. In normal projects, the build file will be a lot smaller than the source, and not the other way around.

Table 2.1 **The initial steps to building and running a program**

Task	Covered in
Step zero: creating the project directory	Section 2.2
Step one: verifying the tools are in place	Section 2.3
Step two: writing your first Ant build file	Section 2.4
Step three: running your first build	Section 2.5
Step four: imposing structure	Section 2.6
Step five: running our program	Section 2.7

2.2 STEP ZERO: CREATING THE PROJECT DIRECTORY

Before doing anything else, create an empty directory. Everything will go under here: source files, created output files, and the build file itself. All new Java/Ant projects should start this way.

Our new directory, `firstbuild`, will be the base directory of our first project. We then create some real Java source to compile. In the new directory, we create a file called `Main.java`, containing the following minimal Java program:

```
public class Main {

    public static void main(String args[]) {
        for(int i=0;i<args.length;i++) {
            System.out.println(args[i]);
        }
    }
}
```

The fact that this program does nothing but print the argument list is unimportant; it's still Java code that we need to build, package, and execute—work we will delegate to Ant.

2.3 STEP ONE: VERIFYING THE TOOLS ARE IN PLACE

Ant is a command-line tool that must be on the path to be used. Appendix A describes how to set up an Ant development system on both Unix and Windows. To compile Java programs, developers also need a properly installed Java Development Kit.

To test that Ant is installed, at a command prompt type

```
ant -version
```

A good response would be something listing a recent version of Ant, version 1.7 or later:

```
Apache Ant version 1.7 compiled on December 13 2006
```

A bad response would be any error message saying Ant is not a recognized command, such as this one on a Unix system:

```
bash: ant: command not found
```

On Windows, the error contains the same underlying message:

```
'ant' is not recognized as an internal or external command,
operable program or batch file.
```

Any such message indicates you have not installed or configured Ant yet, so turn to Appendix A: Installation, and follow the instructions there on setting up Ant.

2.4 STEP TWO: WRITING YOUR FIRST ANT BUILD FILE

Now the fun begins. We are going to get Ant to compile the program.

Ant is controlled by providing a text file that tells how to perform all the stages of building, testing, and deploying a project. These files are *build files*, and every project that uses Ant must have at least one. The most minimal build file useful in Java development is one that builds all Java source in and below the current directory:

```
<?xml version="1.0"?>
<project name="firstbuild" default="compile" >
  <target name="compile">
    <javac srcdir="." />
    <echo>compilation complete!</echo>
  </target>
</project>
```

This is a piece of XML text, which we save to a file called `build.xml`. It is not actually a very good build file. We would not recommend you use it in a real project, for reasons revealed later in this chapter, but it does compile the code.

It's almost impossible for a Java developer to be unaware of XML, but editing it may be a new experience. Don't worry. While XML may seem a bit hard to read at first, and it can be an unforgiving language, it isn't very complex. Readers new to XML files should look at our brief overview in Appendix B. Now let's examine the build file.

2.4.1 Examining the build file

Let's look at that first build file from the perspective of XML format rules. The `<project>` element is always the *root* element in Ant build files, in this case containing two attributes, `name` and `default`. The `<target>` element is a child of `<project>`. The `<target>` element contains two child elements: `<javac>` and `<echo>`.

This file could be represented as a tree, which is how XML parsers represent XML content when a program asks the parser for a Document Object Model (DOM) of the file. Figure 2.1 shows the tree representation.

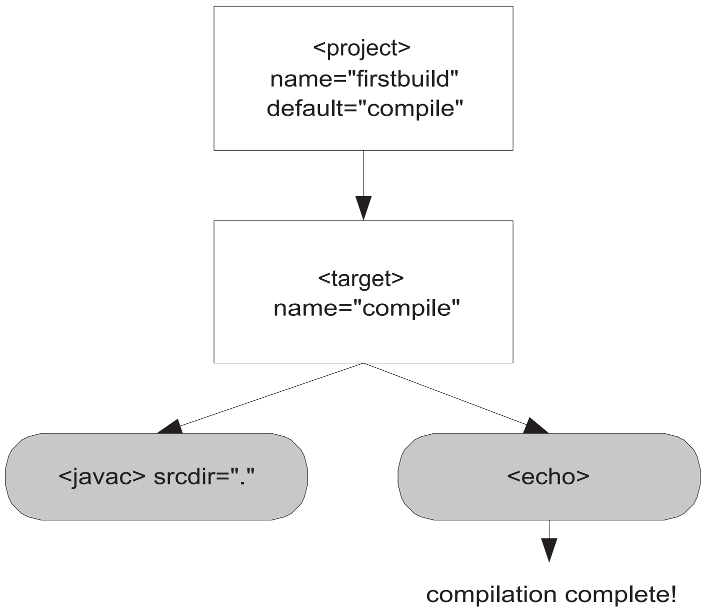


Figure 2.1 The XML Representation of a build file is a tree: the project at the root contains one target, which contains two tasks. This matches the Ant conceptual model: projects contain targets; targets contain tasks.

The graphical view of the XML tree makes it easier to look at a build file, and so the structure of the build file should become a bit clearer. At the top of the tree is a `<project>` element, which has two attributes, `name` and `default`. All Ant build files must contain a single `<project>` element as the root element. It tells Ant the name of the project and, optionally, the default target.

Underneath the `<project>` element is a `<target>` with the name `compile`. A target represents a single stage in the build process. It can be called from the command line or it can be used internally. A build file can have many targets, each of which must have a unique name.

The build file's `compile` target contains two XML elements, one called `<javac>` and one called `<echo>`. The names of these elements should hint as to their function: one calls the `javac` compiler to compile Java source; the other echoes a message to the screen. These are the *tasks* that do the work of this build. The compilation task has one attribute, `srcdir`, which is set to "`.`" and which tells the task to look for source files in and under the current directory. The second task, `<echo>`, has a text child node that will be printed when the build reaches it.

In this build file, we have configured the `<javac>` task with attributes of the task: we have told it to compile files in the current directory. Here, the `<echo>` task uses the text inside it. Attributes on an element describe options and settings that can only set once in the task. A task can support multiple nested elements, such as a list of files to delete. The attributes and elements of every built-in task are listed in Ant's online documentation. Bookmark your local copy of this documentation, as you will use it regularly when creating Ant build files. In the documentation, all *parameters* are XML attributes, and all *parameters specified as nested elements* are exactly that: nested XML elements that configure the task.

Now, let's get our hands dirty by running the build.

2.5 STEP THREE: RUNNING YOUR FIRST BUILD

We've just covered the basic theory of Ant: an XML build file can describe targets to build and the tasks used to build them. You've just created your first build file, so let's try it out. With the Java source and build file in the same directory, Ant should be ready to build the project. At a command prompt in the project directory, type

```
ant
```

If the build file has been typed correctly, then you should see the following response:

```
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file
     [echo] compilation complete!

BUILD SUCCESSFUL

Total time: 2 seconds
```

There it is. Ant has compiled the single Java source file in the current directory and printed a success message afterwards. This is the core build step of all Ant projects that work with Java source. It may seem strange at first to have an XML file telling a tool how to compile a Java file, but soon it will become familiar. Note that we did not have to name the source files; Ant just worked it out somehow. We will spend time in chapter 3 covering how Ant decides which files to work on. For now, you just need to know that the `<javac>` task will compile all Java files in the current directory and any subdirectories. If that's all you need to do, then this build file is adequate for your project. You can just add more files and Ant will find them and compile them.

Of course, a modern project has to do much more than just compile files, which is where the rest of Ant's capabilities, and the rest of this book, come in to play. The first is Ant's ability to report problems.

2.5.1 If the build fails

When you're learning any new computer language, it's easy to overlook mistakes that cause the compiler or interpreter to generate error messages that don't make much sense. Imagine if somehow the XML was mistyped so that the `<javac>` task was misspelled, as in

```
  <javaac srcdir="." />
```

With this task in the target, the output would look something like

```
Buildfile: build.xml

compile:

BUILD FAILED
compile:

BUILD FAILED
C:\AntBook\firstbuild\build.xml:4:
  Problem: failed to create task or type javaac
Cause: The name is undefined.
Action: Check the spelling.
Action: Check that any custom tasks/types have been declared
Action: Check that any <presetdef>/<macrodefs> declarations have taken
place
```

Whenever Ant fails to build, the `BUILD FAILED` message appears. This message will eventually become all too familiar. Usually it's associated with Java source errors or unit test failures, but build file syntax problems result in the same failure message.

If you do get an error message, don't worry. Nothing drastic will happen: files won't be deleted (not in this example, anyway!), and you can try to correct the error by looking at the line of XML named and at the lines on either side of the error. If your editor has good XML support, the editor itself will point out any XML language errors, leaving the command line to find only Ant-specific errors. Editors that are Ant-aware will also catch many Ant-specific syntax errors. An XML editor would also catch the omission of an ending tag from an XML element, such as forgetting to terminate the target element:

```
<?xml version="1.0"?>
<project name="firstbuild" default="compile" >
  <target name="compile">
    <javac srcdir="." />
    <echo>compilation complete!</echo>
</project>
```

The error here would come from the XML parser:

```
C:\AntBook\firstbuild\xml-error.xml:6:
  The element type "target" must be terminated by the matching
  end-tag "</target>".
```

Well-laid-out build files, formatted for readability, help to make such errors visible, while XML-aware editors keep you out of trouble in the first place.

One error we still encounter regularly comes from having an attribute that isn't valid for that task. Spelling the `srcdir` attribute as `sourcedir` is an example of this:

```
    <javac sourcedir="." />
```

If the build file contains that line, you would see this error message:

```
compile:

BUILD FAILED

C:\AntBook\firstbuild\build.xml:4:
 The <javac> task doesn't support the "sourcedir" attribute.
```

This message indicates that the task description contained an invalid attribute. Usually this means whoever created the build file typed something wrong, but it also could mean that the file's author wrote it for a later version of Ant, one with newer attributes or tasks than the version doing the build. That can be hard to fix without upgrading; sometimes a workaround isn't always possible. It's rare that an upgrade would be incompatible or detrimental to your existing build file; the Ant team strives for near-perfect backwards compatibility.

The error you're likely to see most often in Ant is the build halting after the compiler failed to compile your code. If, for example, someone forgot the semicolon after the `println` call, the compiler error message would appear, followed by the build failure:

```
Buildfile: build.xml
compile:
 [javac] Compiling 1 source file
 [javac] /home/ant/firstbuild/Main.java:5: ';' expected
 [javac] System.out.println("hello, world")
 [javac]                                   ^
 [javac] 1 error

BUILD FAILED
/home/ant/firstbuild/build.xml:4: Compile failed, messages
     should have been provided.

Total time: 4 seconds
```

The build failed on the same line as the error in the previous example, line 4, but this time it did the correct action. The compiler found something wrong and printed its messages, and Ant stopped the build. The error includes the name of the Java file and the location within it, along with the compiler error itself.

The key point to note is that failure of a task will usually result in the build itself failing. This is essential for a successful build process: there's no point packaging or delivering a project if it didn't compile. In Ant, the build fails if a task fails. Let's look at the successful build in more detail.

2.5.2 Looking at the build in more detail

If the build does actually succeed, then the only evidence of this is the message that compilation was successful. Let's run the task again, this time in verbose mode, to see what happens. Ant produces a verbose log when invoked with the `-verbose` parameter.

This is a very useful feature when figuring out what a build file does. For our simple build file, it doubles the amount of text printed:

```
> ant -verbose

Apache Ant version 1.7 compiled on December 19 2006
Buildfile: build.xml
Detected Java version: 1.5 in: /usr/java/jdk1.5.0/jre
Detected OS: Linux
parsing buildfile /home/ant/firstbuild/build.xml with URI = file:////home/
ant/firstbuild/build.xml
Project base dir set to: /home/ant/firstbuild/
Build sequence for target(s) 'compile' is [compile]
Complete build sequence is [compile, ]

compile:
    [javac] Main.class skipped - don't know how to handle it
    [javac] Main.java omitted as Main.class is up-to-date.
    [javac] build.xml skipped - don't know how to handle it
     [echo] compilation complete!

BUILD SUCCESSFUL
Total time: 0 seconds
```

For this build, the most interesting lines are those generated by the `<javac>` task. These lines show two things. First, the task did not compile `Main.java`, because it felt that the destination class was up-to-date. The task not only compiles all source files in a directory tree, but it also uses simple timestamp checking to decide which files are up-to-date. All this is provided in the single line of the build file, `<javac srcdir="." />`.

The second finding is that the task explicitly skipped the files `build.xml` and `Main.class`. All files without a `.java` extension are ignored.

What is the log in verbose mode if Ant compiled the source file? Delete `Main.class` then run Ant again to see. The core part of the output provides detail on the compilation process:

```
[javac] Main.java added as Main.class doesn't exist.
[javac] build.xml skipped - don't know how to handle it
[javac] Compiling 1 source file
[javac] Using modern compiler
[javac] Compilation arguments:
[javac] '-classpath'
[javac] '/home/ant/ant/lib/ant-launcher.jar:
  /home/ant/ant/lib/ant.jar:
  /home/ant/ant/lib/xml-apis.jar:
  /home/ant/ant/lib/xercesImpl.jar:
  /usr/java/jdk1.5.0/lib/tools.jar'
[javac] '-sourcepath'
[javac] '/home/ant/firstbuild'
[javac] '-g:none'
```

```
[javac]
[javac] The ' characters around the executable and arguments are
[javac] not part of the command.
[javac] File to be compiled:
[javac]     /home/ant/firstbuild/Main.java
 [echo] compilation complete!

BUILD SUCCESSFUL
```

This time the `<javac>` task does compile the source file, a fact it prints to the log. It still skips the `build.xml` file, printing this fact out before it actually compiles any Java source. This provides a bit more insight into the workings of the task: it builds a list of files to compile, which it passes to the compiler along with Ant's own classpath. The Java-based compiler that came with the Java Development Kit (JDK) is used by default, running inside Ant's own JVM. This keeps the build fast.

The log also shows that we're now running on a Unix system, while we started on a Windows PC. Ant doesn't care what platform you're using, as long as it's one of the many it supports. A well-written build file can compile, package, test, and deliver the same source files on whatever platform it's executed on, which helps unify a development team where multiple system types are used for development and deployment.

Don't worry yet about running the program we compiled. Before actually running it, we need to get the compilation process under control by imposing some structure on the build.

2.6 STEP FOUR: IMPOSING STRUCTURE

The build file is now compiling Java files, but the build process is messy. Source files, output files, and the build file: they're all in the same directory. If this project gets any bigger, things will get out of hand. Before that happens, we must impose some structure. The structure we're going to impose is quite common with Ant and is driven by the three changes we want to make to the project.

- We want to automate the cleanup in Ant. If done incorrectly, this could accidentally delete source files. To minimize that risk, you should always separate source and generated files into different directories.
- We want to place the Java source file into a Java package.
- We want to create a JAR file containing the compiled code. This should be placed somewhere that also can be cleaned up by Ant.

To add packaging and clean-build support to the build, we have to isolate the source, intermediate, and final files. Once source and generated files are separated, it's safe to clean the latter by deleting the output directory, making clean builds easy. These are more reliable than are incremental builds as there is no chance of content sneaking into the output. It's good to get into the habit of doing clean builds. The first step, then, is to sort out the source tree.

2.6.1 Laying out the source directories

We like to have a standard directory structure for laying out projects. Ant doesn't mandate this, but it helps if everyone uses a similar layout. Table 2.2 shows what we use, which is fairly similar to that of Ant's own source tree.

Table 2.2 An Ant project should split source files, compiled classes files, and distribution packages into separate directories. This makes them much easier to manage during the build process.

Directory name	Function
`src`	Source files
`build`	All files generated in a build that can be deleted and recreated
`build/classes`	Intermediate output (created; cleanable)
`dist`	Distributable files (created; cleanable)

The first directory, `src`, contains the Java source. The others contain files that are created during the build. To clean up these directories, the entire directory trees can be deleted. The build file also needs to create the directories if they aren't already present, so that tasks such as `<javac>` have a directory to place their output.

We want to move the Java source into the `src` directory and extend the build file to create and use the other directories. Before moving the Java file, it needs a package name, as with all Java classes in a big project. Here we have chosen `org.antbook.welcome`. We add this name at the top of the source file in a package declaration:

```
package org.antbook.welcome;
public class Main {

    public static void main(String args[]) {
        for(int i=0;i<args.length;i++) {
            System.out.println(args[i]);
        }
    }
}
```

Next, we save the file in a directory tree beneath the source directory that matches that package hierarchy: `src/org/antbook/welcome`. The dependency-checking code in `<javac>` relies on the source files being laid out this way. When the Java compiler compiles the files, it always places the output files in a directory tree that matches the package declaration. The next time the `<javac>` task runs, its dependency-checking code looks at the tree of generated class files and compares it to the source files. It doesn't look inside the source files to find their package declarations; it relies on the source tree being laid out to match the destination tree.

NOTE For Java source file dependency checking to work, you must lay out source in a directory tree that matches the package declarations in the source.

Only when the source is not in any package can you place it in the base of the source tree and expect `<javac>` to track dependencies properly, which is what we've been doing until now. If Ant keeps recompiling your Java files every time you do a build, it's probably because you haven't placed them correctly in the package hierarchy.

It may seem inconvenient having to rearrange your files to suit the build tool, but the benefits become clear over time. On a large project, such a layout is critical to separating and organizing classes. If you start with it from the outset, even on a small project, you can grow more gently from a small project to a larger one. Modern IDEs also prefer this layout structure, as does the underlying Java compiler.

Be aware that dependency checking of `<javac>` is simply limited to comparing the dates on the source and destination files. A regular clean build is a good practice—do so once a day or after refactoring classes and packages.

With the source tree set up, the output directories follow.

2.6.2 Laying out the build directories

Separate from the source directories are the build and distribution directories. We'll configure Ant to put all intermediate files—those files generated by any step in the build process that aren't directly deployed—in or under the build directory. We want to be able to clean up all the generated files simply by deleting the appropriate directory trees. Keeping the directories separate and out of the control of any Software Configuration Management (SCM) tool makes cleanup easy but means that we need to tell Ant to create these directories on demand.

Our project will put the compiled files into a subdirectory of `build`, a directory called `"classes"`. Different intermediate output types can have their own directories alongside this one.

As we mentioned in section 2.5.2, the Java compiler lays out packaged files into a directory tree that matches the package declarations in the source files. The compiler will create the appropriate subdirectories on demand, so we don't need to create them by hand. We do need to create the top-level build directory and the classes subdirectory. We do this with the Ant task `<mkdir>`, which, like the shell command of the same name, creates a directory. In fact, it creates parent directories, too, if needed:

```
<mkdir dir="build/classes">
```

This call is all that's needed to create the two levels of intermediate output. To actually place the output of Ant tasks into the build directory, we need to use each task's attribute to identify a destination directory. For the `<javac>` task, as with many other Ant tasks, the relevant attribute is `destdir`.

2.6.3 Laying out the distribution directories

The `dist` directory contains redistributable artifacts of the project. A common stage in a build process is to package files, placing the packaged file into the `dist` directory. There may be different types of packaging—JAR, Zip, tar, and WAR, for example—and so a subdirectory is needed to keep all of these files in a place where they can be identified and deleted for a clean build. To create the distribution directory, we insert another call to `<mkdir>`:

```
  <mkdir dir="dist">
```

To create the JAR file, we're going to use an Ant task called, appropriately, `<jar>`. We've dedicated chapter 5 to this and the other tasks used in the packaging process. For this introductory tour of Ant, we use the task in its simplest form, when it can be configured to make a named JAR file out of a directory tree:

```
<jar destfile="dist/project.jar" basedir="build/classes" />
```

Doing so shows the advantage of placing intermediate code into the `build` directory: you can build a JAR file from it without having to list what files are included. This is because all files in the directory tree should go in the JAR file, which, conveniently, is the default behavior of the `<jar>` task.

With the destination directories defined, we've now completed the directory structure of the project, which looks like the illustration in figure 2.2. When the build

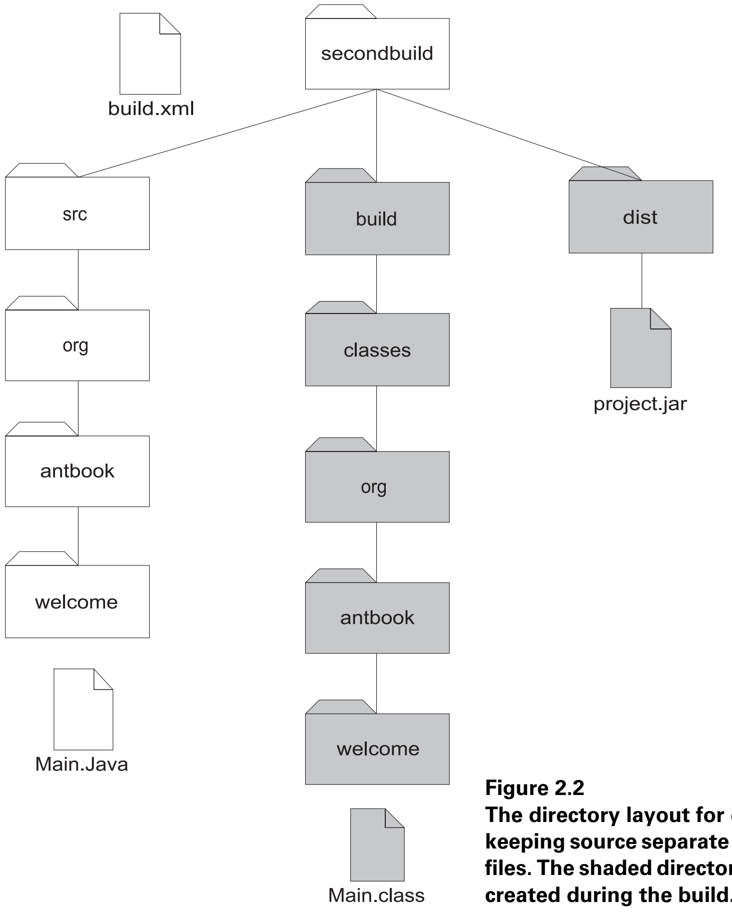


Figure 2.2
The directory layout for our project—keeping source separate from generated files. The shaded directories and files are created during the build.

is executed, a hierarchy of folders will be created in the class directory to match the source tree, but since these are automatically created we won't worry about them.

This is going to be the basic structure of all our projects: source under `src/`, generated files under `build/`, with the compiled classes going under `build/classes`. Future projects will have a lot more files created than just `.class` files, and it's important to leave space for them. With this structured layout, we can have a new build file that creates and uses the new directories.

2.6.4 Creating the build file

Now that we have the files in the right places and we know what we want to do, the build file needs to be rewritten. Rather than glue all the tasks together in one long list of actions, we've broken the separate stages—directory creation, compilation, packaging, and cleanup—into four separate targets inside the build file.

```
<?xml version="1.0" ?>
<project name="structured" default="archive" >

  <target name="init">
    <mkdir dir="build/classes" />          Creates the output
    <mkdir dir="dist" />                   directories
  </target>

  <target name="compile" depends="init" >
    <javac srcdir="src"
      destdir="build/classes"          Compiles into the output directories
      />
  </target>

  <target name="archive" depends="compile" >
    <jar destfile="dist/project.jar"
      basedir="build/classes" />          Creates the archive
  </target>

  <target name="clean" depends="init">
    <delete dir="build" />
    <delete dir="dist"  />          Deletes the output
  </target>                         directories

</project>
```

This build file adds an `init` target to do initialization work, which means creating directories. We've also added two other new targets, `clean` and `archive`. The `archive` target uses the `<jar>` task to create the JAR file containing all files in and below the `build/classes` directory, which in this case means all `.class` files created by the `compile` target. The `clean` target cleans up the output directories by deleting them. It uses a new task, `<delete>`. We've also changed the default target to `archive`, so this will be the target that Ant executes when you run it.

As well as adding more targets, this build file adds another form of complexity. Some targets need to be executed in order. How do we manage this?

2.6.5 Target dependencies

In our current project, for the archive to be up-to-date, all the source files must be compiled, which means the `archive` target must come after the `compile` target. Likewise, `compile` needs the directories created in `init`, so Ant must execute `compile` after the `init` task. *Ant needs to know in what order it should execute targets.*

These are dependencies that we need to communicate to Ant. We do so by listing the direct dependencies in the `depends` attributes of the targets:

```
<target name="compile" depends="init" >
<target name="archive" depends="compile" >
<target name="clean" depends="init">
```

If a target directly depends on more than one target, then we list both dependencies, such as `depends="compile,test"`. In our project, the archive task depends upon both `init` and `compile`, but we don't bother to state the dependency upon `init` because the `compile` target already depends upon it. If Ant must execute `init` before `compile` and `archive` depends upon `compile`, then Ant must run `init` before `archive`. Put formally: *dependencies are transitive.*

What isn't important is the order of targets inside the build file. Ant reads the whole file before it builds the dependency tree and executes targets. There's no need to worry about forward references to targets.

If you look at the dependency tree of targets in the current example, it looks like figure 2.3. Before Ant executes any target, it executes all its predecessor targets. If these predecessors depend on targets themselves, Ant considers those and produces an order that satisfies all dependencies. If two targets in this execution order share a common dependency, then that predecessor will execute only once.

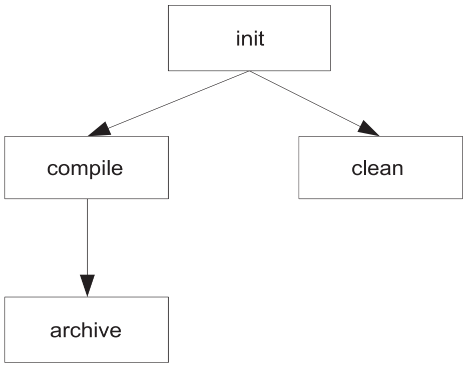


Figure 2.3 Once you add dependencies, the graph of targets gets more complex. Here `clean` depends upon `init`; `archive` depends on `compile`, and, indirectly, `init`. All of a target's dependencies will be executed ahead of the target itself.

Experienced users of Unix's Make tool will recognize that Ant targets resemble that tool's "pseudotargets"—targets in a makefile that you refer to by name in the dependencies of other targets. Usually in Make, you name the source files that a target depends on, and the build tool itself works out what to do to create the target file from the source files. In Ant, you name stages of work as targets, and the tasks inside each target determine for themselves what their dependencies are. Ant builds what is known in computer science circles as a Directed Acyclic Graph (DAG). A DAG is a graph in which the link between nodes has a specific direction—here the *depends* relationship—and in which there are no circular dependencies.

Interlude: circular dependencies

What happens if a target directly or indirectly depends on itself? Does Ant loop? Let's see with a target that depends upon itself:

```
<?xml version="1.0" ?>
<project name="loop" default="loop" >

  <echo>loop test</echo>

  <target name="loop" depends="loop">
    <echo>looping</echo>
  </target>

</project>
```

Run this and you get informed of an error:

```
     [echo] loop test

BUILD FAILED
Circular dependency: loop <- loop

Total time: 0 seconds
Process ant exited with code 1
```

When Ant parses the build file, it builds up the graph of targets. If there is a cycle anywhere in the graph, Ant halts with the error we've just seen.

Any tasks placed in the build files outside of any target will be executed before the target graph is created and analyzed. In our experiment, we had an `<echo>` command outside a target. Ant executes all tasks outside of any target in the order they appear in the build file, before any target processing begins.

With a loop-free build file written, Ant is ready to run it.

2.6.6 Running the new build file

Now that there are multiple targets in the build file, we need a way of specifying which to run. You can simply list one or more targets on the command line, so all of the following are valid:

```
ant
ant init
ant clean
ant compile
ant archive
```

Calling Ant with no target is the same as calling the target named in the `default` attribute of the `<project>`. In the following example, it is the `archive` target:

```
> ant
Buildfile: build.xml

init:
    [mkdir] Created dir: /home/ant/secondbuild/build/classes
    [mkdir] Created dir: /home/ant/secondbuild/dist

compile:
    [javac] Compiling 1 source file to /home/ant/secondbuild/build/classes

archive:
      [jar] Building jar: /home/ant/secondbuild/dist/project.jar

BUILD SUCCESSFUL
Total time: 5 seconds
```

This example demonstrates that Ant has determined the execution order of the targets. As both the `compile` and `archive` targets depend upon the `init` target, Ant calls `init` before it executes either of those targets. It orders the targets so that first the directories get created, then the source is compiled, and finally the JAR archive is built.

The build worked—once. What happens when the build is run a second time?

2.6.7 Incremental builds

Let's look at the log of the build if it's rerun immediately after the previous run:

```
init:

compile:

archive:

BUILD SUCCESSFUL

Total time: 1 second
```

Ant goes through all the targets, but none of the tasks say that they are doing any work. Here's why: all of these tasks in the build file check their dependencies, and do nothing if they do not see a need. The `<mkdir>` task doesn't create directories that already exist, `<javac>` compiles source files when they're newer than the corresponding `.class` file, and the `<jar>` task compares the time of all files to be added to the archive with the time of the archive itself. No files have been compiled, and the JAR is untouched. This is called an *incremental build*.

If you add the `-verbose` flag to the command line, you'll get more detail on what did or, in this case, did not take place.

```
> ant -v
Apache Ant version 1.7 compiled on December 13 2006
Buildfile: build.xml
Detected Java version: 1.5 in: /usr/java/jdk1.5.0/jre
Detected OS: Linux
```

```
parsing buildfile /home/ant/secondbuild/build.xml with
 URI = file:///home/ant/secondbuild/build.xml
Project base dir set to: /home/ant/secondbuild
Build sequence for target(s) 'archive' is [init, compile, archive]
Complete build sequence is [init, compile, archive, clean]

init:

compile:
[javac] org/antbook/welcome/Main.java omitted as
  org/antbook/welcome/Main.class is up-to-date.

archive:
  [jar] org omitted as org/ is up-to-date.
  [jar] org/antbook omitted as org/antbook/ is up-to-date.
  [jar] org/antbook/welcome omitted as
    org/antbook/welcome/ is up-to-date.
  [jar] org/antbook/welcome/Main.class omitted as
    org/antbook/welcome/Main.class is up-to-date.

BUILD SUCCESSFUL
Total time: 1 second
Process ant exited with code 0
```

The verbose run provides a lot of information, much of which may seem distracting. When a build is working well, you don't need it, but it's invaluable while developing that file. Here the build lists the order of target evaluation, which we've boldfaced, and it shows that the `<jar>` task is also dependency-aware: the JAR file was not modified since every file inside it was up-to-date. That shows a powerful feature of Ant: many tasks are dependency-aware, with special logic to handle problems such as timestamps inside Zip/JAR files or to remote FTP sites.

TIP If ever you are unsure why a build is not behaving as expected, add the `-v` or `-verbose` option to get lots more information.

Now that the build file has multiple targets, another question arises. Can we ask for more than one target on the command line?

2.6.8 Running multiple targets on the command line

Developers can run multiple targets in a single build, by listing the targets one after the other on the command line. But what happens when you type `ant compile archive` at the command line? Many people would expect Ant to pick an order that executes each target and its dependencies once only: [`init`, `compile`, `archive`]. Unix Make would certainly do that, but Ant does not. Instead, it executes each target and dependents in turn, so the actual sequence is `init`, `compile`, then `init`, `compile`, `archive`:

```
> ant compile archive
Buildfile: build.xml
```

```
init:
  [mkdir] Created dir: /home/ant/secondbuild/build/classes
  [mkdir] Created dir: /home/ant/secondbuild/dist

compile:
  [javac] Compiling 1 source file to
    /home/ant/secondbuild/build/classes

init:

compile:

archive:
    [jar] Building jar: /home/ant/secondbuild/dist/project.jar

BUILD SUCCESSFUL
Total time: 4 seconds
```

This behavior is a historical accident that nobody dares change. However, if you look closely, the second time Ant executes the `compile` target it does no work; the tasks get executed but their dependency checking prevents existing outputs from being rebuilt.

The final question is this: when a target lists multiple dependencies, does Ant execute them in the order listed? The answer is "yes, unless other dependencies prevent it." Imagine if we modified the `archive` target with the dependency attribute `depends="compile,init"`. A simple left-to-right execution order would run the `compile` target before it was initialized. Ant would try to execute the targets in this order, but because the `compile` target depends upon `init`, Ant will call `init` first. This subtle detail can catch you off guard. If you try to control the execution order by listing targets in order, you may not get the results you expect since explicit dependencies always take priority.

Being able to run multiple targets on the command line lets developers type a sequence of operations such as `ant clean execute` to clean the output directory, rebuild everything, and run the program. Of course, before they can do that, Ant has to be able to run the program.

2.7 STEP FIVE: RUNNING OUR PROGRAM

We now have a structured build process that creates the JAR file from the Java source. At this point the next steps could be to run tests on the code, distribute it, or deploy it. We shall cover those later. For now, we just want to run the program.

2.7.1 Why execute from inside Ant?

We could just call our program from the command line, stating the classpath, the name of the entry point, and the arguments:

```
>java -cp build/classes org.antbook.welcome.Main a b .
a
b
.
```

Calling Java programs from the command line isn't hard, just fiddly. If we run our program from the build file, we get some immediate benefits:

- A target to run the program can depend upon the compilation target, so we know we're always running the latest version of the code.
- It's easy to pass complex arguments to the program.
- It's easy to set up the classpath.
- The program can run inside Ant's own JVM.
- You can halt a build if the return code of the program isn't zero.

Integrating compiling with running a program lets you use Ant to build an application on demand, passing parameters down, including information extracted from other programs run in earlier targets. Running programs under Ant is both convenient and powerful.

2.7.2 Adding an "execute" target

To run the program, we add a new target, `execute`, which depends upon `compile`. It contains one task, `<java>`, that runs our class `Main.class` using the interim `build/classes` directory tree as our classpath:

```
<target name="execute" depends="compile">
  <java
    classname="org.antbook.welcome.Main"
    classpath="build/classes">
    <arg value="a"/>
    <arg value="b"/>
    <arg file="."/>
  </java>
</target>
```

We have three `<arg>` tags inside the `<java>` task; each tag contains one of the arguments to the program: `"a"`, `"b"`, and `"."`, as with the command-line version. Note, however, that the final argument, `<arg file="."/>`, is different from the other two. The first two arguments use the `value` attribute of the `<arg>` tag, which passes the value straight down to the program. The final argument uses the `file` attribute, which tells Ant to resolve that attribute to an absolute file location before calling the program.

Interlude: what can the name of a target be?

All languages have rules about the naming of things. In Java, classes and methods cannot begin with a number. What are Ant's rules about target names?

Ant targets can be called almost anything you want—their names are just strings. However, for the sake of IDEs and Ant itself, here are some rules to follow:

- Don't call targets `""` or `","` because you won't be able to use them.
- Don't use spaces in target names.
- Targets beginning with a minus sign cannot be called from the command line. This means a target name `"-hidden"` could be invoked only by other tasks, not directly by users. IDEs may still allow access to the task.

Ant's convention is to use a minus sign (-) as a separator between words in targets, leading to names such as `"build-install-lite"` or `"functional-tests"`. We would advise against using dots in names, such as `"build.install"`, for reasons we won't get into until the second section of the book entitled, "Applying Ant."

With the `execute` target written, we can compile and run our program under Ant. Let's try it out.

2.7.3 Running the new target

What does the output of the run look like? First, let's run it on Windows:

```
C:\AntBook\secondbuild>ant execute
Buildfile: build.xml

init:

compile:

execute:
     [java] a
     [java] b
     [java] C:\AntBook\secondbuild
```

The `compile` task didn't need to do any recompilation, and the `execute` task called our program. Ant has prefixed every line of output with the name of the task currently running, showing here that this is the output of an invoked Java application. The first two arguments went straight to our application, while the third argument was resolved to the current directory; Ant turned "`.`" into an absolute file reference. Next, let's try the same program on Linux:

```
[secondbuild]> ant execute
Buildfile: build.xml

init:

compile:

execute:
     [java] a
     [java] b
     [java] /home/ant/secondbuild
```

Everything is identical, apart from the final argument, which has been resolved to a different location, the current directory in the Unix path syntax, rather than the DOS one. This shows another benefit of starting programs from Ant rather than from any batch file or shell script: a single build file can start the same program on multiple platforms, transforming filenames and file paths into the appropriate values for the target platform.

This is a very brief demonstration of how and why to call programs from inside Ant, enough to round off this little project. Chapter 6 will focus on the topic of calling Java and native programs from Ant during a build process.

We've nearly finished our quick introduction to Ant, but we have one more topic to cover: how to start Ant.

2.8 ANT COMMAND-LINE OPTIONS

We've already shown that Ant is a command-line program and that you can specify multiple targets as parameters. We've also introduced the `-verbose` option, which allows you to get more information on a build. We want to do some more to run our program. First, we want to remove the `[java]` prefixes, and then we want to run the build without any output unless something goes wrong. Ant's command-line options enable this.

Ant can take a number of options, which it lists if you ask for them with `ant -help`. The current set of options is listed in table 2.3. This list can expand with every version of Ant, though some of the options aren't available or relevant in IDE-hosted versions of the program. Note also that some of the launcher scripts, particularly the Unix shell script, provide extra features, features that the `ant -help` command will list.

Table 2.3 Ant command-line options

Option	Meaning
`-autoproxy`	Bind Ant's proxy configuration to that of the underlying OS.
`-buildfile` *`file`*	Use the named buildfile, use `-f` as a shortcut.
`-debug, -d`	Print debugging information.
`-diagnostics`	Print information that might be helpful to diagnose or report problems.
`-Dproperty=value`	Set a property to a value.
`-emacs`	Produce logging information without adornments.
`-find` *`file`*	Search for the named buildfile up the tree. The shortcut is `-s`.
`-help, -h`	List the options Ant supports and exit.
`-inputhandler` *`classname`*	The name of a class to respond to `<input>` requests.
`-keep-going, -k`	When one target on the command line fails, still run other targets.

continued on next page

Table 2.3 Ant command-line options *(continued)*

Option	Meaning
`-listener` *`classname`*	Add a project listener.
`-logfile` *`file`*	Save the log to the named file.
`-logger` *`classname`*	Name a different logger.
`-main` *`classname`*	Provide the name of an alternate main class for Ant.
`-nice  <number>`	Run Ant at a lower or higher priority.
`-noclasspath`	Discard the CLASSPATH environment variable when running Ant.
`-nouserlib`	Run Ant without using the jar files from .ant/lib under the User's home directory.
`-projecthelp`	Print information about the current project.
`-propertyfile` *`file`*	Load properties from file; -D definitions take priority.
`-quiet, -q`	Run a quiet build: only print errors.
`-verbose, -v`	Print verbose output for better debugging.
`-version`	Print the version information and exit.

Some options require more explanation of Ant before they make sense. In particular, the options related to properties aren't relevant until we explore Ant's properties in chapter 3. Let's look at the most important options first.

2.8.1 Specifying which build file to run

Probably the most important Ant option is `-buildfile`. This option lets you control which build file Ant uses, allowing you to divide the targets of a project into multiple files and select the appropriate build file depending on your actions. A shortcut to `-buildfile` is `-f`. To invoke our existing project, we just name it immediately after the `-f` or `-buildfile` argument:

```
ant -buildfile build.xml compile
```

This is exactly equivalent to calling `ant compile` with no file specified. If for some reason the current directory was somewhere in the source tree, which is sometimes the case when you are editing text from a console application such as `vi`, `emacs`, or even `edit`, then you can refer to a build file by passing in the appropriate relative filename for your platform, such as `../../../build.xml` or `..\..\..\build.xml`. It's easier to use the `-find` option, which must be followed by the name of a build file. This variant does something very special: it searches the directory tree to find the first build file in a parent directory of that name, and invokes it. With this option, when you are deep into the source tree editing files, you can easily invoke the project build with the simple command:

```
ant -find build.xml
```

Note that it can be a bit dangerous to have a build file at the root of the file system, as the `-find` command may find and run it. Most other command-line options are less risky, such as those that control the log level of the program.

2.8.2 Controlling the amount of information provided

We stated that we want to reduce the amount of information provided when we invoke Ant. Getting rid of the `[java]` prefix is easy: we run the build file with the `-emacs` option. This omits the task-name prefix from all lines printed. The option is called `-emacs` because the output is now in the `emacs` format for invoked tools, which enables that and other editors to locate the lines on which errors occurred.

For our exercise, we only want to change the presentation from the command line, which is simple enough:

```
> ant -emacs execute
Buildfile: build.xml

init:

compile:

execute:
   a
   b
   /home/ant/secondbuild

BUILD SUCCESSFUL
Total time: 2 seconds.
```

This leaves the next half of the problem—hiding all the output. Three of the Ant options control how much information is output when Ant runs. Two of these (`-verbose` and `-debug`) progressively increase the amount. The `-verbose` option is useful when you're curious about how Ant works or why a build isn't behaving. The `-debug` option includes all the normal and verbose output and much more low-level information, primarily only of interest to Ant developers. To see nothing but errors or a final build failed/success message, use `-quiet`:

```
> ant -quiet execute

BUILD SUCCESSFUL
Total time: 2 seconds
```

In quiet runs, not even `<echo>` statements appear. One of the attributes of `<echo>` is the `level` attribute, which takes five values: `error`, `warning`, `info`, `verbose`, and `debug` control the amount of information that appears. The default value `info` ensures that messages appear in normal builds and in `-verbose` and `-debug` runs. By inserting an `<echo>` statement into our `execute` target with the `level` set to `warning`, we ensure that the message appears even when the build is running as `-quiet`:

```
  <echo level="warning" message="running" />
```

Such an `<echo>` at the warning level always appears:

```
>ant -q
     [echo] running
```

To eliminate the `[echo]` prefix, we add the `-emacs` option again, calling

```
>ant -q -emacs
```

to get the following output:

```
running

BUILD SUCCESSFUL
Total time: 2 seconds.
```

Asking for `-quiet` builds is good when things are working; asking for `-verbose` is good when they are not. Using `<echo>` to log things at `level="verbose"` can provide extra trace information when things start going wrong. The other way to handle failure is to use the `-keep-going` option.

2.8.3 Coping with failure

The `-keep-going` option tells Ant to try to recover from a failure. If you supply more than one target on the command line, Ant normally stops the moment any of these targets—or any they depend upon—fail. The `-keep-going` option instructs Ant to continue running any target on the command line that doesn't depend upon the target that fails. This lets you run a reporting target even if the main build didn't complete.

2.8.4 Getting information about a project

The final option of immediate relevance is `-projecthelp`. It lists the main targets in a project and is invaluable whenever you need to know what targets a build file provides. Ant lists only those targets containing the optional `description` attribute, as these are the targets intended for public consumption.

```
>ant -projecthelp
Buildfile: build.xml
Main targets:

Other targets:

 archive
 clean
 compile
 execute
 init
Default target: archive
```

This isn't very informative, which is our fault for not documenting the file. If we add a `description` attribute to each target, such as `description="Compiles the source code"` for the `compile` target, and a `<description>` tag right after the project declaration, then the target listing includes these descriptions, marks all the described targets as “main targets,” and hides all other targets from view:

```
> ant -p
Buildfile: build.xml
Compiles and runs a simple program
Main targets:

 archive  Creates the JAR file
 clean    Removes the temporary directories used
 compile  Compiles the source code
 execute  Runs the program

Default target: archive
```

To see both main and sub targets in a project, you must call Ant with the options `-projecthelp` and `-verbose`. The more complex a project is, the more useful the `-projecthelp` feature becomes. We strongly recommend providing description strings for every target intended to act as an entry point to external callers, and a line or two at the top of each build file describing what it does.

Having looked at the options, especially the value of the `-projecthelp` command, let’s return to the build file and add some descriptions.

2.9 EXAMINING THE FINAL BUILD FILE

Listing 2.1 shows the complete listing of the final build file. In addition to adding the description tags, we decided to make the default target run the program. We’ve marked the major changes in bold, to show where this build file differs from the build files and build file fragments shown earlier.

Listing 2.1 Our first complete build file, including packaging and executing a Java program

```
<?xml version="1.0" ?>
<project name="secondbuild" default="execute" >
<description>Compiles and runs a simple program</description>

  <target name="init">
    <mkdir dir="build/classes" />
    <mkdir dir="dist" />
  </target>

  <target name="compile" depends="init"
      description="Compiles the source code">
    <javac srcdir="src"
      destdir="build/classes"
      />
  </target>

  <target name="archive" depends="compile"
      description="Creates the JAR file">
    <jar destfile="dist/project.jar"
      basedir="build/classes"
      />
  </target>

  <target name="clean" depends="init"
      description="Removes the temporary directories used">
    <delete dir="build" />
    <delete dir="dist" />
  </target>

  <target name="execute" depends="compile"
      description="Runs the program">
    <echo level="warning" message="running" />
    <java
      classname="org.antbook.welcome.Main"
      classpath="build/classes">
      <arg value="a"/>
      <arg value="b"/>
      <arg file="."/>
    </java>
  </target>

</project>
```

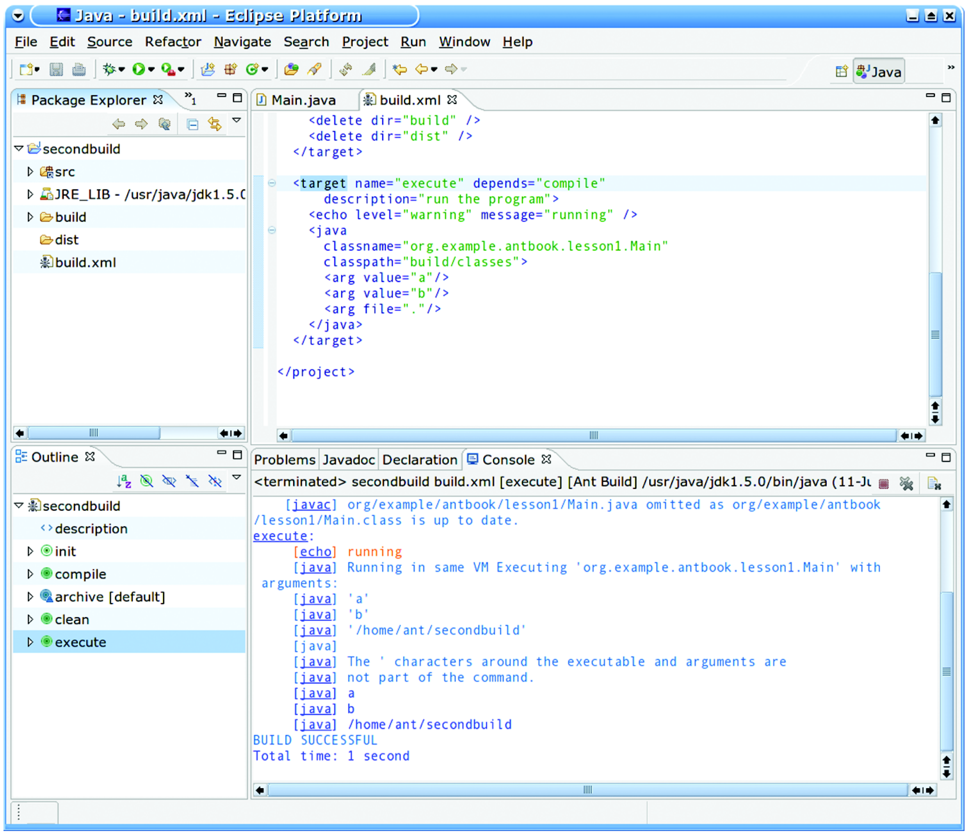
That's forty-plus lines of Ant XML to compile ten lines of Java, but think of what those lines of XML do: they compile the program, package it, run it, and can even clean up afterwards. More importantly, if we added a second Java file to the program, how many lines of code need to change in the build file? Zero. As long as the build process doesn't change, you can now add Java classes and packages to the source tree to build a larger JAR file and perform more useful work on the execution parameters, yet you don't have to make any changes to the build file itself. That is one of the nice features of Ant: you don't need to modify your build files whenever a new source file is added to the build process. It all just works. It even works under an IDE.

2.10 RUNNING THE BUILD UNDER AN IDE

Most modern Java IDEs integrate with Ant. One, NetBeans, is built entirely around Ant. Others, including Eclipse and IntelliJ IDEA, let you add build files to a project and run them from within the GUI.

To show that you can run this Ant under an IDE, figure 2.4 shows a small picture of the `"execute"` target running under Eclipse.

Appendix C covers IDE integration. All the examples in this book were run from the command line for better readability. However, most of the build files were written in IDEs and often were tested there first. Don't think that adopting Ant means abandoning IDE tools; instead you get a build that works everywhere.



```
    <delete dir="build" />
    <delete dir="dist" />
  </target>

  <target name="execute" depends="compile"
      description="run the program">
    <echo level="warning" message="running" />
    <java
      classname="org.example.antbook.lesson1.Main"
      classpath="build/classes">
      <arg value="a"/>
      <arg value="b"/>
      <arg file="."/>
    </java>
  </target>

</project>
```

```
<terminated> secondbuild build.xml [execute] [Ant Build] /usr/java/jdk1.5.0/bin/java (11-Ju
    [javac] org/example/antbook/lesson1/Main.java omitted as org/example/antbook
/lesson1/Main.class is up to date.
execute:
     [echo] running
     [java] Running in same VM Executing 'org.example.antbook.lesson1.Main' with
 arguments:
     [java] 'a'
     [java] 'b'
     [java] '/home/ant/secondbuild'
     [java]
     [java] The ' characters around the executable and arguments are
     [java] not part of the command.
     [java] a
     [java] b
     [java] /home/ant/secondbuild
BUILD SUCCESSFUL
Total time: 1 second
```

Figure 2.4 Our build file hosted under Eclipse. Consult Appendix C for the steps needed to do this.

2.11 SUMMARY

Ant is told what to build by an XML file, a *build file*. This file describes all the actions to build an application, such as creating directories, compiling the source, and determining what to do afterwards; the actions include making a JAR file and running the program.

The build file is in XML, with the root `<project>` element representing a Ant *project*. This project contains *targets*, each of which represents a stage of the project. A target can depend on other targets, which is stated by listing the dependencies in the `depends` attributes of the target. Ant uses this information to determine which targets to execute, and in what order.

The actual work of the build is performed by Ant *tasks*. These tasks implement their own dependency checking, so they only do work if it is needed.

Some of the basic Ant tasks are `<echo>` to print a message, `<delete>` to delete files and directories, `<mkdir>` to create directories, `<javac>` to compile Java source, and `<jar>` to create an archive file. The first three of these tasks look like XML versions of shell commands, but the latter two demonstrate the power of Ant. They contain dependency logic, so that `<javac>` will compile only those source files for which the destination binary is missing or out of date, and `<jar>` will create a JAR file only if its input files are newer than the output.

Running Ant is called *building*; a build either succeeds or fails. Builds fail when there's an error in the build file, or when a task fails by throwing an exception. In either case, Ant lists the line of the build file where the error occurred. Ant can build from the command line, or from within Java IDEs. The command line has many options to control the build and what output gets displayed. Rerunning a build with the `-verbose` option provides more detail as to what is happening. Alternatively, the -`quiet` option runs a build nearly silently. The most important argument to the command line is the name of the targets to run—Ant executes each of these targets and all its dependencies.

After this quick introduction, you're ready to start using Ant in simple projects. If you want to do this or if you have deadlines that insist on it, go right ahead. The next two chapters will show you how to configure and control Ant with its properties and datatypes, and how to run unit tests under it. If your project needs these features, then please put off coding a bit longer, and keep reading.

JAVA DEVELOPMENT

ANT IN ACTION

Steve Loughran • Erik Hatcher

The most widely used build tool for Java projects, Ant is cross-platform, extensible, simple, and fast. It scales from small personal projects to large, multi-team enterprise projects. And, most important, it's easy to learn.

Ant in Action is a complete guide to using Ant to build, test, redistribute and deploy Java applications. A retitled second edition of the bestselling and award-winning *Java Development with Ant,* this book contains over 50% new content including:

- New Ant 1.7 features
- Scalable builds for big projects
- Continuous integration techniques
- Deployment
- Library management
- Extending Ant

Whether you are dealing with a small library or a complex server-side system, this book will help you master your build process. By presenting a running example that grows in complexity, the book covers nearly the entire gamut of modern Java application development, including test-driven development and even how to set up your database as part of the deployment.

Steve Loughran, an Ant committer and member of the Apache Software Foundation, is a Research Scientist at Hewlett-Packard Laboratories.

Erik Hatcher has been an Ant project committer and is a coauthor of Manning's popular *Lucene in Action.*

For more information, code samples, and to purchase an ebook visit www.manning.com/AntinAction

"... you owe it to yourself to read this book."

—Kevin Jackson, Ant Committer

"If you do Java software, and there's only one book you read this year, it should be this one."

—Leo Simons
Apache Gump Developer and
Senior Engineer, Joost

"Don't put your build at risk by not reading this book."

—Jon Skeet
Senior Software Engineer
Audatex (UK)

"Absolutely recommended for any developer."

—Bas Vodde, Manager Agile &
Integrative Product Development
Nokia Siemens Networks

"It's worth buying the book for Chapter 16 alone."

—Julian Simpson
ThoughtWorks Ltd.

ISBN-10 1-932394-80-X
ISBN-13: 978-1-932394-80-1

54999

9 781932 394801

MANNING

$49.99 / Can $64.99