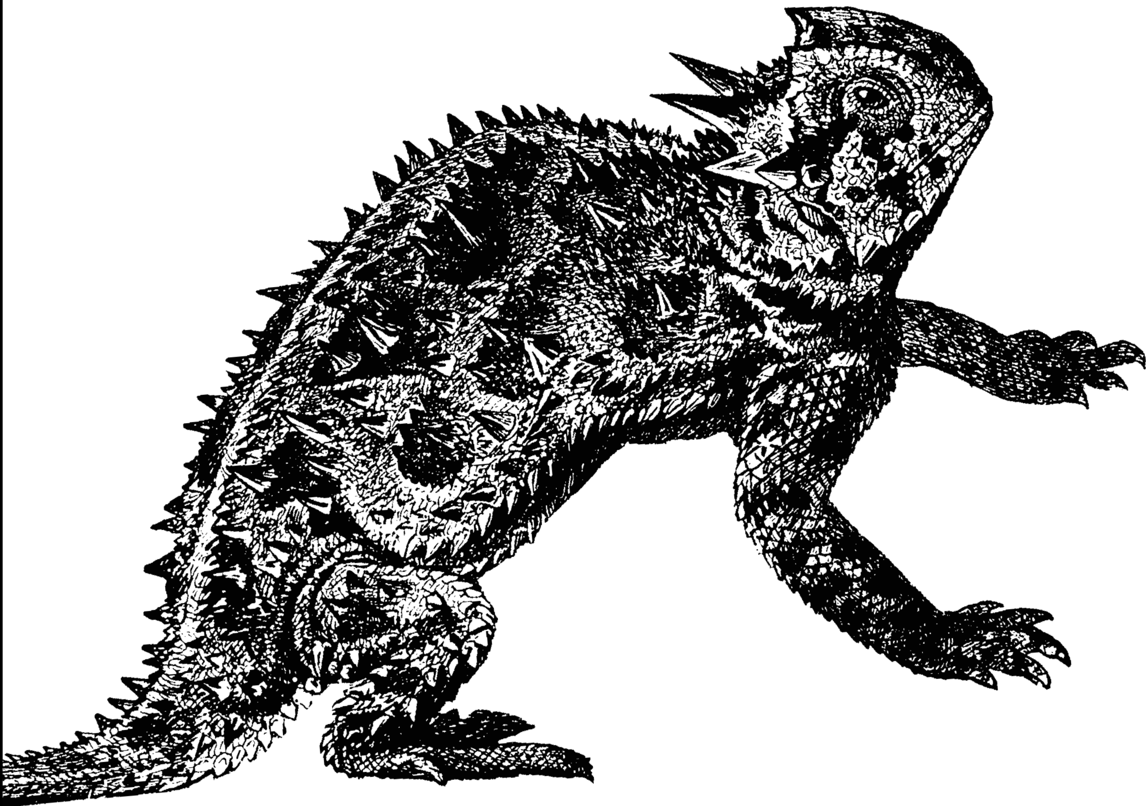
Complete Build Management for Java

Ant

The Definitive Guide



Jesse Tilly & Eric M. Burke

Foreword by James Duncan Davidson

O'REILLY®

Ant

The Definitive Guide

Jesse Tilly and Eric M. Burke

Foreword by James Duncan Davidson

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1

Ant Jumpstart

It is likely that you have already downloaded and installed Ant and are ready to see an example of how it works. If so, then this chapter is for you. Here, we walk through a very basic buildfile example, followed by a full description of Ant's command-line options. If you prefer to walk through the step-by-step installation procedure first, you might want to skip ahead to Chapter 2 and then come back to this material.

We do not attempt to explain every detail of the buildfile in this chapter. For a more comprehensive example, see Chapter 3.

Files and Directories

For our example, we start with the directory and file structure shown in Figure 1-1. The shaded boxes represent files, and the unshaded boxes represent directories.

You can download this example from this book's web page, located at *http://www.oreilly.com/catalog/anttdg/*.

The Ant buildfile, *build.xml*, exists in the project base directory. This is typical, although you are free to use other filenames or put the buildfile somewhere else. The *src* directory contains the Java source code organized into an ordinary package structure. For the most part, the content of the source files is not important. However, we want to point out that *PersonTest.java* is a unit test that will be excluded from the generated JAR file.

Our sample buildfile causes Ant to create the directory tree and files shown inside the shaded, dashed block in Figure 1-2. It also compiles the Java source code, creates *oreilly.jar*, and provides a "clean" target to remove all generated files and directories.

Now let's look at the buildfile that makes this possible.

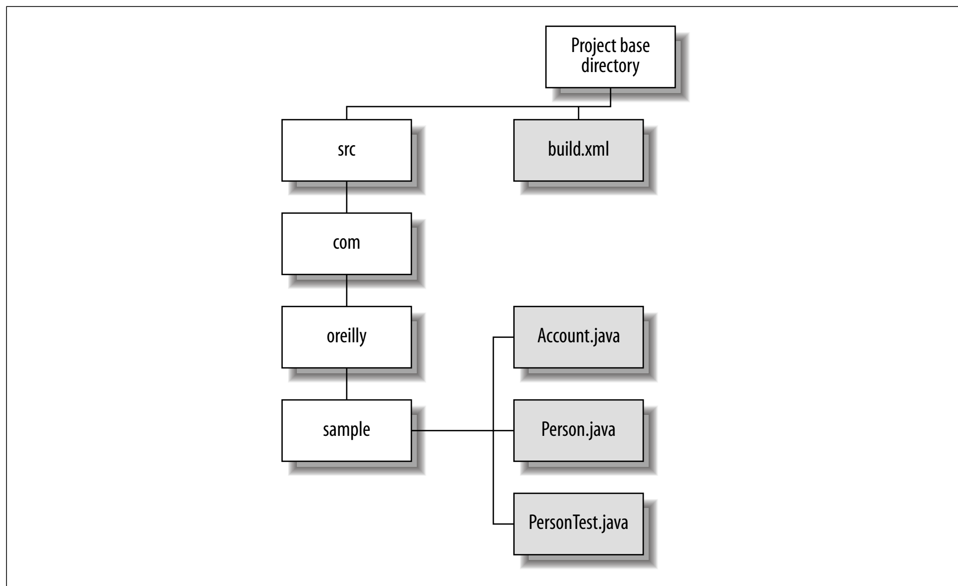


Figure 1-1. Starting point for our example buildfile

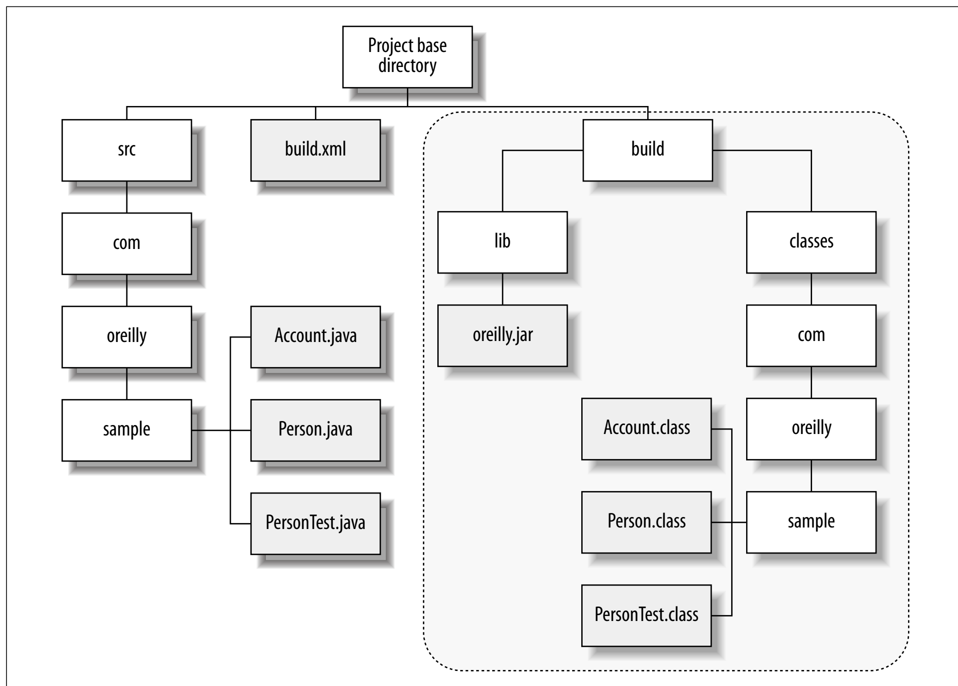


Figure 1-2. Directories and files created by our sample buildfile

The Ant Buildfile

Ant buildfiles are written using XML. Example 1-1 shows the complete Ant buildfile for our example. This is simpler than most real-world buildfiles, but does illustrate several core concepts required by nearly every Java project.

Example 1-1. build.xml

```
<?xml version="1.0"?>

<!-- build.xml - a simple Ant buildfile -->
<project name="Simple Buildfile" default="compile" basedir=".">

  <!-- The directory containing source code -->
  <property name="src.dir" value="src"/>

  <!-- Temporary build directories -->
  <property name="build.dir" value="build"/>
  <property name="build.classes" value="${build.dir}/classes"/>
  <property name="build.lib" value="${build.dir}/lib"/>

  <!-- Target to create the build directories prior to the -->
  <!-- compile target. -->
  <target name="prepare">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes}"/>
    <mkdir dir="${build.lib}"/>
  </target>

  <target name="clean" description="Removes all generated files.">
    <delete dir="${build.dir}"/>
  </target>

  <target name="compile" depends="prepare"
          description="Compiles all source code.">
    <javac srcdir="${src.dir}" destdir="${build.classes}"/>
  </target>

  <target name="jar" depends="compile"
          description="Generates oreilly.jar in the 'dist' directory.">
    <!-- Exclude unit tests from the final JAR file -->
    <jar jarfile="${build.lib}/oreilly.jar"
         basedir="${build.classes}"
         excludes="**/*Test.class"/>
  </target>

  <target name="all" depends="clean,jar"
          description="Cleans, compiles, then builds the JAR file."/>

</project>
```

XML Considerations

Ant buildfiles are XML files that can be created with any text editor. Keep the following points in mind as you create your own buildfiles:

- The first line is the XML declaration. If present, it must be the very first line in the XML file; no preceding blank lines are allowed. In fact, even a single blank space before `<?xml` causes the XML parser to fail.
- XML is very picky about capitalization, quotes, and proper tag syntax. If any of these items are incorrect, Ant fails because its underlying XML parser fails.

Here is an example of an error that occurs if the `</project>` end tag is typed incorrectly as `</Project>`:

```
Buildfile: build.xml

BUILD FAILED

C:\antbook\build.xml:41: Expected "</project>" to terminate
element starting on line 4.

Total time: 2 seconds
```

Buildfile Description

Our buildfile consists of several XML comments, the required `<project>` element, and many *properties*, *tasks*, and *targets*. The `<project>` element establishes the working directory for our project: ".". This is the directory containing the buildfile. It also specifies the default target, which is "compile." The purpose of the default target will become apparent shortly when we describe how to run Ant.

The property definitions allow us to avoid hardcoding directory names throughout the buildfile. These paths are always relative to the base directory specified by the `<project>` element. For example, the following tag sets the name of our source directory:

```
<property name="src.dir" value="src"/>
```

Next, our buildfile defines several targets. Each target has a name, such as "prepare," "clean," or "compile." Developers interact with these when invoking Ant from the command line. Each target defines zero or more dependencies, along with an optional `description` attribute. Dependencies specify targets that Ant must execute first, before the target in question is executed. For example, "prepare" must execute before "compile" does. The `description` attribute provides a human-readable description of a target that Ant will display on command.

Within targets we have tasks, which do the actual work of the build. Ant 1.4.1 ships with over 100 core and optional tasks; you can find all of the tasks described in detail

in Chapters 7 and 8. These tasks perform functions ranging from creating directories to playing music when the build finishes.*

Running Ant

We are going to assume that Ant is installed properly. If you have any doubts on this point, now is the time to read Chapter 2 and get everything up and running.

Examples

To execute the tasks in the default target, `compile`, type the following command from the directory containing our sample *build.xml* file:

```
ant
```

Ant will open the default buildfile, which is *build.xml*, and execute that buildfile's default target (which in our case is `compile`). You should see the following output, assuming your directory is called *antbook*:

```
Buildfile: build.xml

prepare:
    [mkdir] Created dir: C:\antbook\build
    [mkdir] Created dir: C:\antbook\build\classes
    [mkdir] Created dir: C:\antbook\build\lib

compile:
    [javac] Compiling 3 source files to C:\antbook\build\classes

BUILD SUCCESSFUL

Total time: 5 seconds
```

As Ant runs, it displays the name of each target executed. As our example output shows, Ant executes `prepare` followed by `compile`. This is because `compile` is the default target, which has a dependency on the `prepare` target. Ant prints the name of each task within brackets, along with other messages unique to each task.

In our sample output, [javac] is the name of the Ant task, not necessarily the name of the Java compiler. If you are using IBM's Jikes, for instance, [javac] is still displayed because that is the Ant task that is running Jikes behind the scenes.

When you invoke Ant without specifying a buildfile name, Ant searches for a file named *build.xml* in the current working directory. You aren't limited to this default;

* See the sound task in Chapter 8.

you can use any name you like for the buildfile. For example, if we call our buildfile *proj.xml*, we must type this command, instead:

```
ant -buildfile proj.xml
```

We can also explicitly specify one or more targets to run. We can type **`ant clean`** to remove all generated code, for instance. If our buildfile is called *proj.xml*, and we want to execute the `clean` target, we type **`ant -buildfile proj.xml clean`**. Our output would look something like this:

```
Buildfile: proj.xml

clean:
   [delete] Deleting directory C:\antbook\build

BUILD SUCCESSFUL

Total time: 2 seconds
```

We can also execute several targets with a single command:

```
ant clean jar
```

This invokes the `clean` target followed by the `jar` target. Because of the target dependencies in our example buildfile, Ant executes the following targets in order: `clean`, `prepare`, `compile`, `jar`. The `all` target takes advantage of these dependencies, allowing us to clean and rebuild everything by typing **`ant all`**:

```
<target name="all" depends="clean,jar"
        description="Cleans, compiles, then builds the JAR file."/>
```

`all` is dependent on `clean` and `jar`. `jar`, in turn, is dependent on `compile`, and `compile` is dependent on `prepare`. The simple command *ant all* ends up executing all our targets, and in the proper order.

Getting Help

You may have noticed that some of our targets include the `description` attribute, while others do not. This is because Ant distinguishes between main targets and subtargets. Targets containing descriptions are main targets, and those without are considered subtargets. Other than documentation differences, main targets and subtargets behave identically. Typing **`ant -projecthelp`** from our project base directory produces the following output:

```
Buildfile: build.xml
Default target:

 compile  Compiles all source code.

Main targets:

 all      Cleans, compiles, then builds the JAR file.
 clean    Removes all generated files.
```

```
 compile  Compiles all source code.
 jar      Generates oreilly.jar in the 'dist' directory.

Subtargets:

 prepare


BUILD SUCCESSFUL

Total time: 2 seconds
```

This project help feature is particularly useful for large projects containing dozens of targets, provided you take the time to add meaningful descriptions.

For a summary of the Ant command-line syntax, type **`ant -help`**. You will see a brief description of Ant's command-line arguments, which we cover next.

Ant Command-Line Reference

The syntax to use to invoke Ant from the command-line is as follows:

```
ant [option [option...]] [target [target...]]

option := {-help
          |-projecthelp
          |-version
          |-quiet
          |-verbose
          |-debug
          |-emacs
          |-logfile filename
          |-logger classname
          |-listener classname
          |-buildfile filename
          |-Dproperty=value
          |-find filename}
```

The syntax elements are as follows:

-help
: Displays help information describing the Ant command and its options.

-projecthelp
: Displays any user-written help documentation included in the buildfile. This is text from the `description` attribute of any `<target>`, along with any text contained within a `<description>` element. Targets with `description` attributes are listed as "Main targets," those without are listed as "Subtargets."

-version
: Causes Ant to display its version information and exit.

-quiet
: Suppresses most messages not originated by an echo task in the buildfile.

-verbose
: Displays detailed messages for every operation during a build. This option is exclusive to *-debug*.

-debug
: Displays messages that Ant and task developers have flagged as debugging messages. This option is exclusive to *-verbose*.

-emacs
: Formats logging messages so that they're easily parsed by Emacs' *shell-mode*; i.e., prints the task events without preceding them with an indentation and a [*taskname*].

-logfile *filename*
: Redirects logging output to the specified file.

-logger *classname*
: Specifies a class to handle Ant logging. The class specified must implement the *org.apache.tools.ant.BuildLogger* interface.

-listener *classname*
: Declares a listening class for Ant to add to its list of listeners. This option is useful when integrating Ant with IDEs or other Java programs. Read more about listeners in Chapter 6. The specified listening class must be written to handle Ant's build messaging.

-buildfile *filename*
: Specifies the buildfile Ant should operate on. The default buildfile is *build.xml*.

-Dproperty=value
: Defines a property name-value pair on the command line.

-find *filename*
: Specifies the buildfile on which Ant should operate. Unlike the *-buildfile* option, *-find* causes Ant to search for the specified file in the parent directory if it is not found in the current directory. This searching continues through ancestor directories until the root of the filesystem is reached, at which time the build fails if the file is not found.

Buildfile Outline

Shown next is a generic buildfile good for using as a template. A buildfile consists of the <project> element with its nested <target>, <property>, and <path> elements.

```
<project default="all">
  <property name="a.property" value="a value"/>
  <property name="b.property" value="b value"/>
```

```
  <path id="a.path">
    <pathelement location="${java.home}/jre/lib/rt.jar"/>
  </path>

  <target name="all">
    <javac srcdir=".">
      <classpath refid="a.path"/>
    </javac>
  </target>
</project>
```

Some notes about buildfiles to remember:

- All buildfiles require the `<project>` element and at least one `<target>` element.
- There is no default value for the `<project>` element's `default` attribute.
- Buildfiles do not have to be named *build.xml*. However, *build.xml* is the default name for which Ant searches.
- You can have only one `<project>` element per buildfile.

Learning More

We have only scratched the surface in this chapter, but this does give you an idea of what Ant buildfiles look like and how to run the command-line *ant* script. As mentioned earlier, Chapter 3 presents a much more sophisticated example with a great deal more explanation of what is going on in each step.