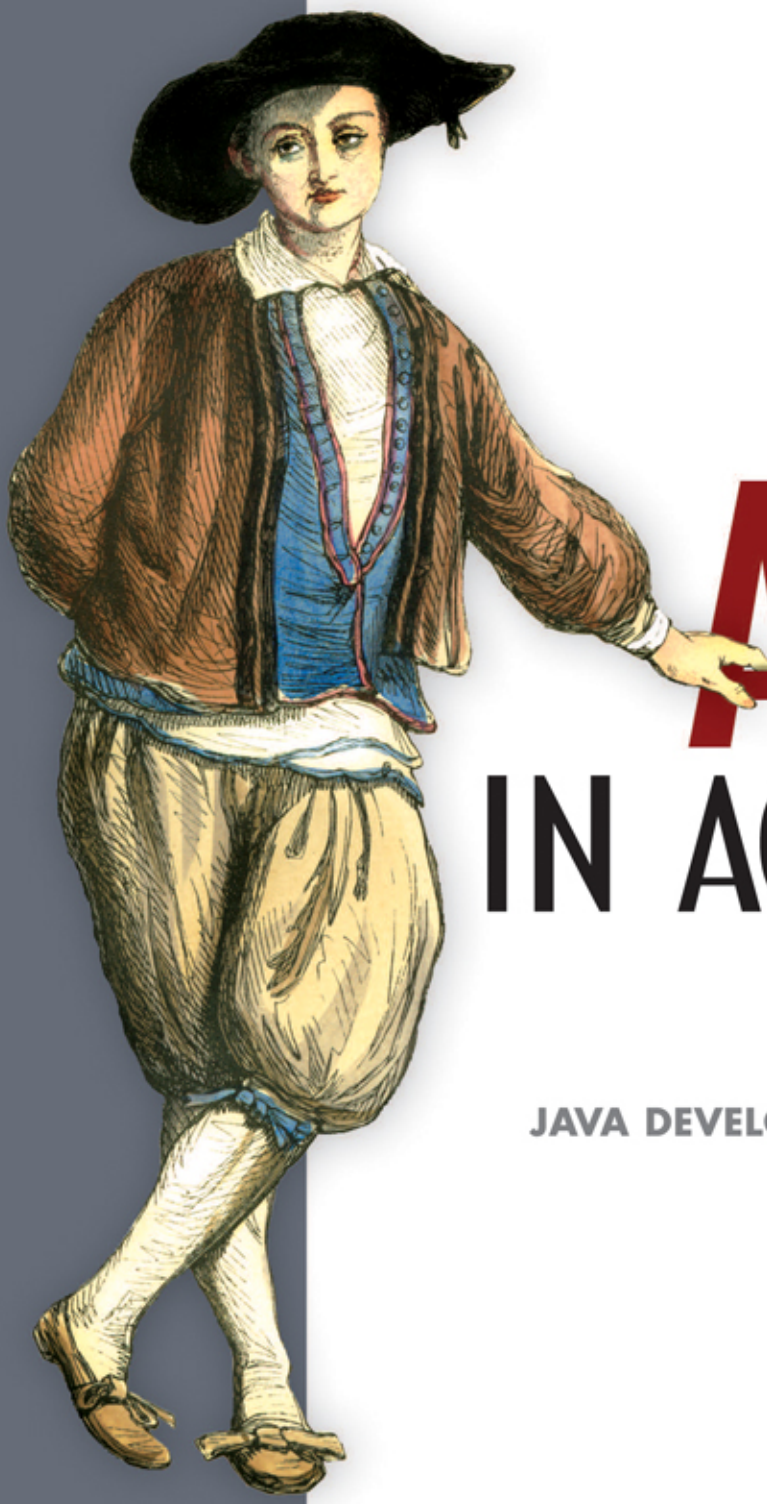
COVERS ANT 1.7

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 1

brief contents

CHAPTER 1

Introducing Ant

Welcome to the future of your build process.

This is a book about Ant. It's more than just a reference book for Ant syntax, it's a collection of best practices demonstrating how to use Ant to its greatest potential in real-world situations. If used well, you can develop and deliver your software projects better than you have done before.

Let's start with a simple question: what is Ant?

1.1 WHAT IS ANT?

Ant is a *build tool*, a small program designed to help software teams develop big programs by automating all the drudge-work tasks of compiling code, running tests, and packaging the results for redistribution. Ant is written in Java and is designed to be cross-platform, easy to use, extensible, and scalable. It can be used in a small personal project, or it can be used in a large, multiteam software project. It aims to automate your entire build process.

The origin of Ant is a fascinating story; it's an example of where a spin-off from a project can be more successful than the main project. The main project in Ant's case is Tomcat, the Apache Software Foundation's Java Servlet engine, the reference implementation of the Java Server Pages (JSP) specification. Ant was written by James Duncan Davidson, then a Sun employee, to make it easier for people to compile

Tomcat on different platforms. The tool he wrote did that, and, with help from other developers, became the way that Apache Java projects were built. Soon it spread to other open source projects, and trickled out into helping Java developers in general.

That happened in early 2000. In that year and for the following couple of years, using Ant was still somewhat unusual. Nowadays, it's pretty much expected that any Java project you'll encounter will have an Ant build file at its base, along with the project's code and—hopefully—its tests. All Java IDEs come with Ant support, and it has been so successful that there are versions for the .NET framework (NAnt) and for PHP (Phing). Perhaps the greatest measure of Ant's success is the following: a core feature of Microsoft's .NET 2.0 development toolchain is its implementation of a verson: MSBuild. That an XML-based build tool, built in their spare time by a few developers, is deemed worthy of having a "strategic" competitor in the .NET framework is truly a measure of Ant's success.

In the Java world, it's the primary build tool for large and multiperson projects—things bigger than a single person can do under an IDE. Why? Well, we'll get to that in section 1.2—the main thing is that it's written in Java and focuses on building and testing Java projects.

Ant has an XML syntax, which is good for developers already familiar with XML. For developers unfamiliar with XML, well, it's one place to learn the language. These days, all Java developers need to be familiar with XML.

In a software project experiencing constant change, an automated build can provide a foundation of stability. Even as requirements change and developers struggle to catch up, having a build process that needs little maintenance and remembers to test everything can take a lot of housekeeping off developers' shoulders. Ant can be the means of controlling the building and deployment of Java software projects that would otherwise overwhelm a team.

1.1.1 The core concepts of Ant

We have just told you why Ant is great, but now we are going to show you what makes it great: its ingredients, the core concepts. The first is the design goal: Ant was designed to be an extensible tool to automate the build process of a Java development project.

A *software build process* is a means of going from your source—code and documents—to the product you actually deliver. If you have a software project, you have a build process, whether or not you know it. It may just be "hit the compile button on the IDE," or it may be "drag and drop some files by hand." Neither of these are very good because they aren't automated and they're often limited in scope.

With Ant, you can delegate the work to the machine and add new stages to your build process. Testing, for example. Or the creation of XML configuration files from your Java source. Maybe even the automatic generation of the documentation.

Once you have an automated build, you can let anyone build the system. Then you can find a spare computer and give it the job of rebuilding the project *continuously*. This is why automation is so powerful: it starts to give you control of your project.

Ant is Java-based and tries to hide all the platform details it can. It's also highly extensible in Java itself. This makes it easy to extend Ant through Java code, using all the functionality of the Java platform and third-party libraries. It also makes the build very fast, as you can run Java programs from inside the same Java virtual machine as Ant itself.

Putting Ant extensions aside until much later, here are the core concepts of Ant as seen by a user of the tool.

Build Files

Ant uses XML files called *build files* to describe how to build a project. In the build file developers list the high-level various goals of the build—the *targets*—and actions to take to achieve each goal—the *tasks*.

A build file contains one project

Each build file describes how to build one project. Very large projects may be composed of multiple smaller projects, each with its own build file. A higher-level build file can coordinate the builds of the subprojects.

Each project contains multiple targets

Within the build file's single project, you declare different targets. These targets may represent actual outputs of the build, such as a redistributable file, or activities, such as compiling the source or running the tests.

Targets can depend on other targets

When declaring a target, you can declare which targets have to be built first. This can ensure that the source gets compiled before the tests are run and built, and that the application is not uploaded until the tests have passed. When Ant builds a project, it executes targets in the order implied by their dependencies.

Targets contain tasks

Inside targets, you declare what work is needed to complete that stage of the build process. You do this by listing the tasks that constitute each stage. When Ant executes a target, it executes the tasks inside, one after the other.

Tasks do the work

Ant tasks are XML elements, elements that the Ant runtime turns into actions. Behind each task is a Java class that performs the work described by the task's attributes and nested data. These tasks are expected to be smart—to handle much of their own argument validation, dependency checking, and error reporting.

New tasks extend Ant

The fact that it's easy to extend Ant with new classes is one of its core strengths. Often, someone will have encountered the same build step that you have and will have written the task to perform it, so you can just use their work. If not, you can extend it in Java, producing another reusable Ant task or datatype.

To summarize, Ant reads in a build file containing a project. In the project are targets that describe different things the project can do. Inside the targets are the tasks, tasks that do the individual steps of the build. Ant executes targets in the order implied by their declared dependencies, and the tasks inside them, thereby building the application. That's the theory. What does it look like in practice?

1.1.2 Ant in action: an example project

Figure 1.1 shows the Ant build file as a graph of targets, each target containing tasks. When the project is built, Ant determines which targets need to be executed, and in what order. Then it runs the tasks inside each target. If a task somehow fails, Ant halts the build. This lets simple rules such as "deploy after compiling" be described, as well as more complex ones such as "deploy only after the unit tests have succeeded."

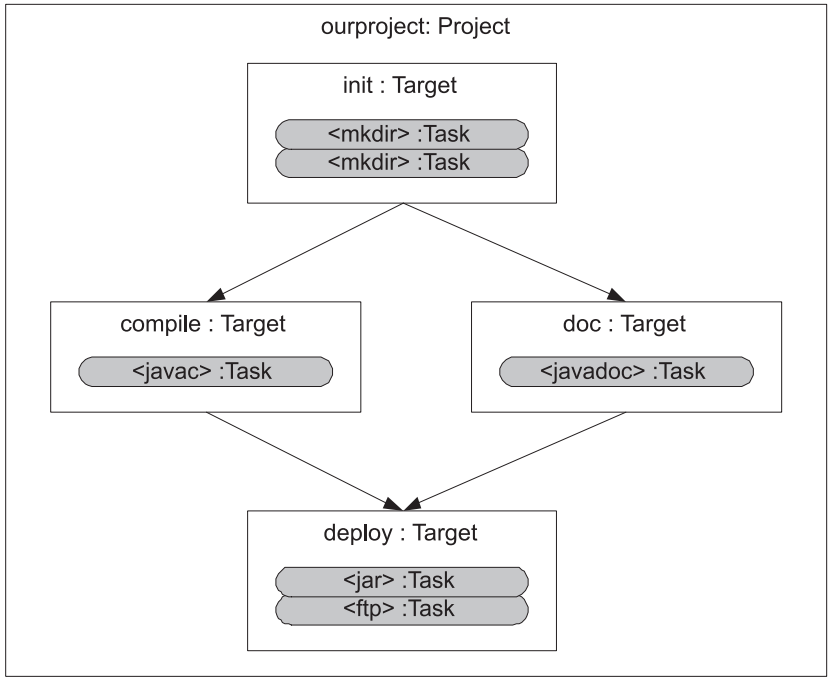


Figure 1.1 Conceptual view of a build file. The project encompasses a collection of targets. Inside each target are task declarations, which are statements of the actions Ant must take to build that target. Targets can state their dependencies on other targets, producing a graph of dependencies. When executing a target, all its dependents must execute first.

Listing 1.1 shows the build file for this typical build process.

Listing 1.1 A typical scenario: compile, document, package, and deploy

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

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

  <target name="compile" depends="init">
    <javac srcdir="src"
      destdir="build/classes"/>
  </target>

  <target name="doc" depends="init" >
    <javadoc destdir="build/classes"
             sourcepath="src"
             packagenames="org.*" />
  </target>

  <target name="package" depends="compile,doc" >
    <jar destfile="dist/project.jar"
         basedir="build/classes"/>
  </target>

  <target name="deploy" depends="package" >
    <ftp server="${server.name}"
         userid="${ftp.username}"
         password="${ftp.password}">
      <fileset dir="dist"/>
    </ftp>
  </target>
</project>
```

Create two output directories for generated files

Compile the Java source

Create the javadocs of all org.* source files

Create a JAR file of everything in build/classes

Upload all files in the dist directory to the ftp server

While listing 1.1 is likely to have some confusing pieces to it, it should be mostly comprehensible to a Java developer new to Ant. For example, packaging (`target name="package"`) depends on the successful javac compilation and javadoc documentation (`depends="compile,doc"`). Perhaps the most confusing piece is the `${...}` notation used in the FTP task (`<ftp>`). That indicates use of Ant properties, which are values that can be expanded into strings. The output of our build is

```
> ant -propertyfile ftp.properties
Buildfile: build.xml

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

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

doc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source files for package
            org.example.antbook.lesson1...
  [javadoc] Constructing Javadoc information...
  [javadoc] Building tree for all the packages and classes...
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...
package:
      [jar] Building jar: /home/ant/ex/dist/project.jar
deploy:
      [ftp] sending files
      [ftp] 1 files sent

BUILD SUCCESSFUL
Total time: 5 seconds.
```

Why did we invoke Ant with `-propertyfile ftp.properties`? We have a file called `ftp.properties` containing the three properties `server.name`, `ftp.username`, and `ftp.password`. The property handling mechanism allows parameterization and reusability of our build file. This particular example, while certainly demonstrative, is minimal and gives only a hint of things to follow. In this build, we tell Ant to place the generated documentation alongside the compiled classes, which isn't a typical distribution layout but allows this example to be abbreviated. Using the `-propertyfile` command-line option is also atypical and is used in situations where forced override control is desired, such as forcing a build to upload to a different server. This example shows Ant's basics well: target dependencies, use of properties, compiling, documenting, packaging, and, finally, distribution.

For the curious, here are pointers to more information on the specifics of this build file: chapter 2 covers build file syntax, target dependencies, and `<javac>` in more detail; chapter 3 explains Ant properties, including `-propertyfile`; chapter 5 delves into `<jar>` and `<javadoc>`; and, finally, `<ftp>` is covered in chapter 7.

Because Ant tasks are Java classes, the overhead of invoking each task is quite small. For each task, Ant creates a Java object, configures it, then calls its `execute()` method. A simple task such as `<mkdir>` would call a Java library method to create a directory. A more complex task such as `<ftp>` would invoke a third-party FTP library to talk to the remote server, and, optionally, perform dependency checking to upload only files that were newer than those at the destination. A very complex task such as `<javac>` not only uses dependency checking to decide which files to compile, it supports multiple compiler back ends, calling Sun's Java compiler in the same Java Virtual Machine (JVM), or executing a different compiler as an external executable.

These are implementation details. Simply ask Ant to compile some files—how Ant decides which compiler to use and what its command line is are issues that you rarely need to worry about. It just works.

That's the beauty of Ant: it just works. Specify the build file correctly, and Ant will work out target dependencies and call the targets in the right order. The targets run through their tasks in order, and the tasks themselves deal with file dependencies and the actual execution of the appropriate Java package calls or external commands needed to perform the work. Because each task is usually declared at a high level, one or two lines of XML is often enough to describe what you want a task to do. Ten lines might be needed for something as complex as creating a database table. With only a few lines needed per task, you can keep each build target small, and keep the build file itself under control.

That is why Ant is popular, but that's not the only reason.

1.2 WHAT MAKES ANT SO SPECIAL?

Ant is the most popular build tool in Java projects. Why is that? What are its unique attributes that helped it grow from a utility in a single project to the primary build system of Java projects?

Ant is free and Open Source

Ant costs nothing to download. It comes with online documentation that covers each task in detail, and has a great online community on the Ant developer and user mail lists. If any part of Ant doesn't work for you, you can fix it. All the Ant developers got into the project by fixing bugs that mattered to them or adding features that they needed. The result is an active project where the end users are the developers.

Ant makes it easy to bring developers into a project

One of the benefits of using Ant comes when a new developer joins a team. With a nicely crafted build process, the new developer can be shown how to get code from the source code repository, including the build file and library dependencies. Even Ant itself could be stored in the repository for a truly repeatable build process.

It is well-known and widely supported

Ant is the primary build tool for Java projects. Lots of people know how to use it, and there is a broad ecosystem of tools around it. These tools include third-party Ant tasks, continuous-integration tools, and editors/IDEs with Ant support.

It integrates testing into the build processes

The biggest change in software development in the last few years has been the adoption of test-centric processes. The *agile* processes, including Extreme Programming and Test-Driven Development, make writing tests as important as writing the

functional code. These *test-first* processes say that developers should write the tests before the code.

Ant doesn't dictate how you write your software—that's your choice. What it does do is let anyone who does write tests integrate those tests into the build process. An Ant build file can mandate that the unit tests must all pass before the web application is deployed, and that after deploying it, the functional tests must be run. If the tests fail, Ant can produce a nice HTML report that highlights the problems.

Adopting a test-centric development process is probably the most important and profound change a software project can make. Ant is an invaluable adjunct to that change.

It enables continuous integration

With tests and an automated build that runs those tests, it becomes possible to have a machine rebuild and retest the application on a regular basis. How regularly? Nightly? How about every time someone checks something into the code repository?

This is what continuous integration tools can do: they can monitor the repository and rerun the build when something changes. If the build and tests work, they update a status page on their web site. If something fails, developers get email notifying them of the problem. This catches errors within minutes of the code being checked in, stopping bugs from hiding unnoticed in the source.

It runs inside Integrated Development Environments

Integrated Development Environments (IDEs) are great for editing, compiling, and debugging code, and they're easy to use. It's hard to convince users of a good IDE that they should abandon it for a build process based on a text file and a command line prompt. Ant integrates with all mainstream IDEs, so users do not need to abandon their existing development tools to use Ant.

Ant doesn't replace an IDE; a good editor with debugging and even refactoring facilities is an invaluable tool to have and use. Ant just takes control of compilation, packaging, testing, and deployment stages of the build process in a way that's portable, scalable, and often reusable. As such, it complements IDEs. The latest generation of Java IDEs all support Ant. This means that developers can choose whatever IDE they like, and yet everyone can share the same automated build process.

1.3 WHEN TO USE ANT

When do you need Ant? When is an automated build tool important? The approximate answer is "whenever you have any project that needs to compile or test Java code." At the start of the project, if only one person is coding, then an IDE is a good starting point. As soon as more people work on the code, its deliverables get more complex, or the test suite starts to be written, then its time to turn to Ant. This is also a great time to set up the continuous integration server, or to add the project to a running one.

Another place to use Ant is in your Java programs, if you want to use its functionality in your own project. While Ant was never designed with this reuse in mind, it can be used this way. Chapter 18 looks at embedding Ant inside another program.

1.4 WHEN NOT TO USE ANT

Although Ant is a great build tool, there are some places where it isn't appropriate.

Ant is not the right tool to use outside of the build process. Its command line and error messages are targeted at developers who understand English and Java programming. You should not use Ant as the only way end-users can launch an application. Some people do this: they provide a build file to set up the classpath and run a Java program, or they use Ant to glue a series of programs together. This works until there's a problem and Ant halts with an error message that only makes sense to a developer.

Nor is Ant a general-purpose workflow engine; it lacks the persistence or failure handling that such a system needs. Its sole options for handling failure are "halt" or "ignore," and while it may be able to run for days at a time, this is something that's never tested. The fact that people do try to use Ant for workflow shows that there's demand for a portable, extensible, XML-based workflow engine. Ant is not that; Ant is a tool for making development easier, not solving every problem you can imagine.

Finally, setting up a build file takes effort. If you're just starting out writing some code, it's easier to stay in the IDEs, using the IDE to set up your classpath, to build, and to run tests. You can certainly start off a project that way, but as soon as you want HTML test reports, packaging, and distribution, you'll need Ant. It's good to start work on the build process early, rather than try to live in the IDE forever.

1.5 ALTERNATIVES TO ANT

Ant is not the only build tool available. How does it fare in comparison to its competition and predecessors? We'll compare Ant to its most widely used comptetitors—IDEs Make, and Maven.

1.5.1 IDEs

IDEs are the main way people code: Eclipse, NetBeans, and IntelliJ IDEA are all great for Java development. Their limitations become apparent as a project proceeds and grows.

- It's very hard to add complex operations, such as XSL stylesheet operations, Java source generation from IDL/WSDL processing, and other advanced tricks.
- It can be near-impossible to transfer one person's IDE settings to another user. Settings can end up tied to an individual's environment.
- IDE-based build processes rarely scale to integrate many different subprojects with complex dependencies.
- Producing replicable builds is an important part of most projects, and it's risky to use manual IDE builds to do so.

All modern IDEs have Ant support, and the IDE teams all help test Ant under their products. One IDE, NetBeans, uses Ant as its sole way of building projects, eliminating any difference between the IDE and Ant. The others integrate Ant within their own build process, so you can call Ant builds at the press of button.

1.5.2 Make

The Unix Make tool is the original build tool; it's the underpinnings of Unix and Linux. In Make, you list targets, their dependencies, and the actions to bring each target up-to-date.

The tool is built around the file system. Each target in a makefile is either the name of a file to bring up-to-date or what, in Make terminology, is called a *phony target*. A named target triggers some actions when invoked. Make targets can depend upon files or other targets. Phony targets have names like `clean` or `all` and can have no dependencies (that is, they always execute their commands) or can be dependent upon real targets.

One of the best parts of Make is that it supports pattern rules to determine how to build targets from the available inputs, so that it can infer that to create a `.class` file, you compile a `.java` file of the same name.

All the actions that Make invokes are actually external programs, so the rule to go from `.java` files to `.class` files would invoke the javac program to compile the source, which doesn't know or care that it has been invoked by Make.

Here's an example of a very simple GNU makefile to compile two Java classes and archive them into a JAR file:

```
all: project.jar

project.jar: Main.class XmlStuff.class
        jar -cvf $@ $<

%.class: %.java
        javac $<
```

The makefile has a phony target, `all`, which, by virtue of being first in the file, is the default target. It depends upon `project.jar`, which depends on two compiled Java files, packaging them with the JAR program. The final rule states how to build class (`.class`) files from Java (`.java`) files. In Make, you list the file dependencies, and the tool determines which rules to apply and in what sequence, while the developer is left tracking down bugs related to the need for invisible tab characters rather than spaces at the start of each action.

When someone says that they use Make, it usually means they use Make-on-Unix, or Make-on-Windows. It's very hard to build across both, and doing so usually requires a set of Unix-compatible applications, such as the Cygwin suite. Because Make handles the dependencies, it's limited to that which can be declared in the file: either timestamped local files or phony targets. Ant's tasks contain their own

dependency logic. This adds work for task authors, but benefits task users. This is because specialized tasks to update JAR files or copy files to FTP servers can contain the code to decide if an entry in a JAR file or a file on a remote FTP server is older than a local file.

Ant versus Make

Ant and Make have the same role: they automate a build process by taking a specification file and using that and source files to create the desired artifacts. However, Ant and Make do have some fundamentally different views of how the build process should work.

With Ant, you list sequences of operations and the dependencies between them, and you let file dependencies sort themselves out through the tasks. The only targets that Ant supports are similar to Make's phony targets: targets that are not files and exist only in the build file. The dependencies of these targets are other targets. You omit file dependencies, along with any file conversion rules. Instead, the Ant build file states the stages used in the process. While you may name the input or output files, often you can use a wildcard or even a default wildcard to specify the source files. For example, here the `<javac>` task automatically includes all Java files in all subdirectories below the source directory:

```
<?xml version="1.0" ?>
<project name="makefile" default="all">
  <target name="all">
    <javac srcdir="."/>
    <jar destfile="project.jar" includes="*.class" />
  </target>
</project>
```

Both the `<javac>` and `<jar>` tasks will compare the sources and the destinations and decide which to compile or add to the archive. Ant will call each task in turn, and the tasks can choose whether or not to do work. The advantage of this approach is that the tasks can contain more domain-specific knowledge than the build tool, such as performing directory hierarchy-aware dependency checking, or even addressing dependency issues across a network. The other subtlety of using wildcards to describe source files, JAR files on the classpath, and the like is that you can add new files without having to edit the build file. This is nice when projects start to grow because it keeps build file maintenance to a minimum.

Ant works best with programs that are wrapped by Java code into a task. The task implements the dependency logic, configures the application, executes the program, and interprets the results. Ant does let you execute native and Java programs directly, but adding the dependency logic is harder than it is for Make. Also, with its Java focus, there's still a lot to be said for using Make for C and C++ development, at least on Linux systems, where the GNU implementation is very good, and where the development tools are installed on most end users' systems. For Java projects, Ant has the

edge, as it is portable, Java-centric, and even redistributable if you need to use it inside your application.

1.5.3 Maven

Maven is a competing build tool from Apache, hosted at http://maven.apache.org. Maven uses templates—archetypes—to define how a specific project should be built. The standard archetype is for a Java library, but others exist and more can be written.

Like Ant, Maven uses an XML file to describe the project. Ant's file explicitly lists the stages needed for each step of the build process, but neglects other aspects of a project such as its dependencies, where the source code is kept under revision control, and other things. Maven's Project Object Model (POM) file declares all this information, information that Maven plugins use to manage all parts of the build process, from retrieving dependent libraries to running tests and generating reports.

Central to Maven is the idea that the tools should encode a set of best practices as to how projects should be laid out and how they should test and release code. Ant, in comparison, has no formal knowledge of best practices; Ant leaves that to the developers to decide on so they can implement their own policy.

Ant versus Maven

There is some rivalry between the two Apache projects, though it is mostly good-natured. The developer teams are friends, sharing infrastructure bits and, sometimes, even code.

Ant views itself as the more flexible of the two tools, while Maven considers itself the more advanced of the pair. There are some appealing aspects to Maven, which can generate a JAR and a web page with test results from only a minimal POM file. It pulls this off if the project is set up to follow the Maven rules, and every library, plugin, and archetype that it depends upon is in the central Maven artifact repository. Once a project starts to diverge from the templates the Maven team have provided, however, you end up looking behind the curtains and having to fix the underpinnings. That transition from "Maven user" to "plugin maintainer" can be pretty painful, by all accounts.

Still, Maven does have some long-term potential and it's worth keeping an eye on, but in our experience it has a hard time building Java projects with complex stages in the build process. To be fair, building very large, very complex Java projects is hard with any tool. Indeed, coping with scale is one of the ongoing areas of Ant evolution, which is why chapters 10 and 11 in this book are dedicated to that problem.

1.6 THE ONGOING EVOLUTION OF ANT

Ant is still evolving. As an Apache project, it's controlled by their bylaws, which cover decision-making and write-access to the source tree. Those with write-access to Ant's source code repository are called *committers*, because they're allowed to commit code changes directly. All Ant users are encouraged to make changes to the code, to extend Ant to meet their needs, and to return those changes to the Ant community.

As table 1.1 shows, the team releases a new version of Ant on a regular basis. When this happens, the code is frozen during a brief beta release program. When they come out, public releases are stable and usable for a long period.

Table 1.1 The release history of Ant. Major revisions come out every one to two years; minor revisions release every three to six months.

Date	Ant version	Notes
March 2000	Ant 1.0	Really Ant 0.9; with Tomcat 3.1
July 2000	Ant 1.1	First standalone Ant release
October 2000	Ant 1.2	
March 2001	Ant 1.3	
September 2001	Ant 1.4	Followed by Ant 1.4.1 in October
July 2002	Ant 1.5	Along with the first edition of *Java Development with Ant*
September 2003	Ant 1.6	With regular 1.6.x patches
June 2005	Ant 1.6.5	Last of the 1.6 branch
December 2006	Ant 1.7.0	The version this edition of the book was written against

New releases come out every 12–24 months; point releases, mostly bug fixes, come out about every quarter. The team strives to avoid breaking existing builds when adding new features and bug fixes. Nothing in this book is likely to break over time, although there may be easier ways of doing things and the tool will offer more features. Build files written for later versions of Ant do not always work in older releases—this book has targeted Ant 1.7.0, which was released in December 2006. Users of older versions should upgrade before continuing, while anyone without a copy of Ant should download and install the latest version. If needed, Appendix A contains instructions on how to do so.

1.7 SUMMARY

This chapter has introduced Ant, a Java tool that can build, test, and deploy Java projects ranging in size from the very small to the very, very large.

- Ant uses XML *build files* to describe what to build. Each file covers one Ant *project*; a project is divided into *targets*; targets contain *tasks*. These tasks are the Java classes that actually perform the construction work. Targets can depend on other targets. Ant orders the execution so targets execute in the correct order.
- Ant is a free, open source project with broad support in the Java community. Modern IDEs support it, as do many developer tools. It also integrates well with modern test-centric development processes, bringing testing into the build process.

- There are other tools that have the same function as Ant—to build software—but Ant is the most widely used, broadly supported tool in the Java world.
- Ant is written in Java, is cross platform, integrates with all the main Java IDEs, and has a command-line interface.

Using Ant itself does not guarantee a successful Java project; it just helps. It is a tool and, like any tool, provides greatest benefit when used properly. We're going to explore how to do that by looking at the tasks and types of Ant, using it to compile, test, package, execute, and then redistribute a Java project. Let's start with a simple Java project, and a simple build file.

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

$49.99 / Can $64.99