Beginning POJOs

From Novice to Professional

Brian Sam-Bodden

Apress®

Beginning POJOs: From Novice to Professional

Copyright © 2006 by Brian Sam-Bodden

ISBN-13 (pbk): 978-159059-596-1

ISBN-10 (pbk): 1-59059-596-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Hastings Hart
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Lori Bring
Indexer: Michael Brinkman
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `www.apress.com`.

The source code for this book is available to readers at `www.apress.com` in the Source Code section.

CHAPTER 3

Building with Ant

The traditional definition of a build process entails converting source code into an executable deliverable. In the world of enterprise Java development this definition falls short. In this chapter you'll learn how to use the popular build tool Ant to set the stage for the build system used in the TechConf application. A production J2EE application build system will typically need to do much more than simply compiling and packaging your code. Some sample tasks that can be performed by a build include the following:

- **Version control:** Obtaining the latest version of a project's source code from a version control repository
- **Build plan:** Determining what to build
- **Generate:** Generating any source code from several sources such as annotated code, database tables, and Unified Modeling Language (UML) diagrams
- **Formatting:** Correcting syntax and style
- **Checking:** Validating syntax and style
- **Compiling:** Generating .class files from .java files
- **Testing:** Running automated tests
- **Validating:** Verifying components' validity
- **Javadoc:** Generating API documentation
- **Metrics:** Generating code metrics reports
- **Packaging:** Generating JAR, web archive (WAR), and enterprise archive (EAR) files
- **Deploying:** Deploying applications to servers
- **Distributing:** Distributing packaged applications
- **Notifying:** Notifying developers and managers of important build-related events

This relatively short list of activities should give you an idea of how involved the build process can become. How many times have you heard the dreaded, "But it was working just fine on my machine!" A reproducible build is of paramount importance for keeping your code base healthy and your project in a known state at all times. Having a reproducible and stable build process takes more than just having a dedicated team of developers. Without automation, even a small project with few developers can rapidly get out of hand.

By using an automated build tool, developers can define the steps in the process of building their software and execute those steps reliably under different environments and circumstances. Typically such tools will account for individual configuration differences between developers' environments and production systems. Most build tools have some sort of configuration or script that describes the build process in discrete, atomic steps.

A typical build process also covers aspects of both the production and the development stages of an application. For example, in a database-driven application, individual developers might need to initialize a database with sample data needed for testing, while in a production environment such a step would not be required.

Although integrated development environments (IDEs) have always provided a level of support for the building process, this support usually falls short of developers' needs and expectations. Most of these build solutions aren't portable across environments; it's hard enough to get one developer's IDE project file to work on any environment except for its creator's. Not only are these facilities IDE-independent, but they're also very different from the work that an application assembler or deployer has to do for a production application. Common sense should tell you that the closer your development environment is to the production environment, the fewer problems you'll have going into production. By having a build process that is consistent across development and production environments (and any other environments in between), you can eradicate many development maladies that come from using multiple IDEs, operating systems, and Java versions.

As the build process is automated and becomes transparent to programmers, other issues such as testing and documentation generation find their way into the build process. Most developers find that they begin with a build system that evolves to accomplish more than simply "building." From testing to document generation, a finely crafted build process eventually becomes a reflection of a team's development process.

In J2EE, a consistent build system brings together the roles of the application developer, assembler, and deployer. As part of the J2EE specification, Sun defined several roles in its definition of the J2EE platform. Newcomers to J2EE might quickly put themselves in one of these categories and disregard the details of the other roles. But the reality is that unless you have an understanding of every role's responsibility, your understanding of the J2EE platform will not be complete. In particular, the roles of the application assembler and the application deployer are reflected in the build process, and unless your developers can duplicate what happens in production you're likely to experience a painful transition from development into production.

Introduction to Ant

A project with a few files and very few dependencies makes the process of building almost not a process at all. By simply using the Java compiler and maybe the JAR command-line utility, you can build simple Java applications.

Before Ant, developers typically started with a set of simple batch files or shell scripts as an initial step towards automation. But as the number of files, components, target platforms, and virtual machine (VM) versions increases so does the build time, the complexity of the build, and the likelihood that human errors will contribute to irreproducible and inconsistent builds. After a while, you end up realizing that maintaining a non-portable, platform-dependent homemade solution is cumbersome and error-prone.

For the few teams in which developers actually agree on the choice of an IDE, the first choice is usually the build functionality provided by the IDE. Most IDEs provide wizards that build simple applications. These wizards cover only part of the equation, and they tie your team to the particular IDE.

Besides the aforementioned problems, both approaches treat development and production environments as being conceptually separate. What's needed is a low-level tool that can unify the build process across multiple IDEs, stages of development, platforms, and so on.

For many years, UNIX programmers have had a way to build their applications via the make utility and all of its variants (GNU Make, nmake, and so on). Like make, Ant is at its core a build tool, but as the Ant website states, Ant "is kind of like Make, but without Make's wrinkles" (`http://ant.apache.org/`).

Ant's simplicity has contributed to its rapid adoption and made it the de facto standard for building applications in the Java world. Ant, together with the Concurrent Versions System (CVS), has played an important role in fostering open source by providing a universal way for individuals to obtain, build, and contribute to the open source community. Ant has also become an indispensable tool for most Java developers, especially those developing J2EE applications.

Ant has made life easier for Java developers worldwide. Although far from perfect, it has demonstrated that it can cover what a Java developer needs, from gaining control over the build process to cutting the umbilical cord from proprietary build systems.

The most relevant reasons to choose Ant are as follows:

- **Platform independence:** A typical corporate Java environment includes development teams that work on Wintel machines and deploy to UNIX machines for production. Ant, being a pure Java tool, makes it possible to have a consistent build process regardless of the platform, thereby making the development, staging, integration, and production environments closer to each other. Ant also has built-in capabilities that handle platform differences. Your Java code is portable; your build should be too!!

- **Adoption:** Ant is everywhere! Yes, by itself this is a poor reason to favor a technology, but the strengths that ubiquity brings to the table are many, including hiring, training, and marketability of skills. Ant also has been integrated into many of the leading IDEs, thereby making it the one consistent factor between developers. This is partly due to the choice, for good and bad reasons, of XML as its language.

- **Functionality and flexibility:** For the majority of Java projects, Ant is extensible and highly configurable; it provides the required functionality right out of the box. For Java developers, any class can easily become an Ant task, although in our experience we seldom have to write our own tasks (because someone in the open source community always seems to beat you to the punch). If desired, you can plug scripting engines and run platform-specific commands.

- **Syntax:** Like it or not, XML has become a globally recognized data format. Most Java developers have worked with XML, and J2EE developers deal with XML on a daily basis. XML makes Ant buzzword-compliant. But XML also has some advantages. XML is ideal for representing structured data because of its hierarchical nature. The abundance of commercial and open source parsers, and the ability to easily check an XML file for being well-formed and valid has made the use of XML pervasive in the industry.

Ant's architecture is similar to the make utility in that it's based on the concept of a target. In Ant a target is a modular unit of execution that uses tasks to accomplish its work. An Ant target has dependencies and can be conditionally executed. A build is usually composed of some main targets that will accomplish some coarse-grained process related to an application's build, such as compiling the code or packaging a component. These main targets might make use of other subtargets (usually via dependencies) to accomplish their job.

Underneath the covers, tasks are plain Java classes that extend the `org.apache.tools.ant.Task` class, although any class that exposes a method with the signature void execute() can become an Ant task. One of Ant's great advantages is its extensibility. Ant tasks are pluggable plain Java classes. To write a task all you need to do is extend the `Task` class and add some code to the `execute` method. Ant comes loaded with myriad tasks to accomplish many of the things needed during a typical build. These tasks are referred to as the core tasks and the optional tasks. There are also a countless number of third-party tasks, whether they're commercial, freeware, or open source.

The scope of Ant's contribution to Java development isn't obvious at first, especially on small projects. But once complexity begins to creep in and you have multiple developers, you'll find that Ant becomes the glue that can help your team work in synchronization. It can basically remove the need for a full-time build "engineer." This is largely the case with most open source Java projects, and their success should be a testament to the effectiveness of the integration power of using Ant.

Ant isn't without its critics, however. Many have failed to understand that Ant was never meant to be a full-fledged scripting language but a Java-friendly way to automate the build process in a simple declarative, goal-oriented fashion. Since its inception, many scriptinglike features have been added to Ant in the form of custom tasks, and the arguments between camps that want a full scripting language and ones that want a simple, dependency-driven build system continue to this day. In my opinion there is no right answer; scripting is programming, and you know the issues that arise with that. On the other hand, Ant's simple declarative ways make it hard to do write-once and reuse builds across different projects. Ant's reusability is at the task level. In his essay "Ant in Anger" (`http://ant.apache.org/ant_in_anger.html`), Steve Loughran recommends that to achieve the level of complexity that most developers turn to scripting to achieve, Ant builds can be dynamically generated on a per-project basis using something like eXtensible Stylesheet Language Transformations (XSLT).

Fortunately, Ant version 1.6 provides new features that make Ant build reuse a reality. We will cover some of the relevant features that enable reuse later in this chapter.

Obtaining and Installing Ant

Ant can be obtained from `http://ant.apache.org` in binary and source distributions, or you can obtain the source code through CVS. Ant is a pure Java application. Therefore, the only requirement to run it is that you have a compliant JDK installed and a parser compliant with Java API for XML Processing (JAXP). Ant ships with the latest Apache Xerces2 parser. Ant is distributed as a compressed archive (.zip, tar.gz, and tar.bz2). Once the archive has been uncompressed to a directory (this directory is referred to as ANT_HOME), it's recommended that you add the environment variable ANT_HOME to your system and the bin directory under the ANT_HOME directory to your system's executable path. The bin directory contains scripts in many different formats for the most popular platforms. These scripts facilitate the execution of Ant and include DOS batch, UNIX shell, and Perl and Python scripts. Ant also relies on the JAVA_HOME environment variable to determine the JDK to be used.

Caution If you have only the JRE installed (a rare case for most Java developers) many of Ant's tasks will not work properly.

To verify that Ant is installed correctly, at the command prompt type:

```
ant -version
```

If the installation was successful you should see a message showing the version of Ant and the compilation date:

```
Apache Ant version 1.6.5 compiled on June 2 2005
```

Ant's Command-Line Options

Ant is typically used from the command line by running one of the scripts in the bin directory. Ant's command line can take a set of options (prefixed with a dash) and any number of targets to be executed, as follows:

```
ant [options] [target target2 ... targetN]
```

Table 3-1 shows the options available from the command line. You can access them by typing `ant -help`. By default, Ant will search for a file named build.xml unless a different file is specified via the `buildfile` option.

Table 3-1. *Ant Command-Line Options*

Option	Purpose
help | h	Prints the help message showing all available options
projecthelp | p	Displays all targets for which the `description` attribute has been set
version	Prints the version of Ant
diagnostics	Prints a diagnostics report that shows information like file sizes and compilation dates; useful for reporting bugs

Table 3-1. *Ant Command-Line Options*

Option	Purpose
quiet | q	Minimizes the amount of console output produced by Ant
verbose | v	Maximizes the amount of console output produced by Ant
debug | d	Prints debugging information to the console
emacs | e	Removes all indentation and decorations from the console output
lib <path>	Configures a file system path to search for JARs and Java classes
logfile | l <file>	Redirects all console output to the specified log file
logger <classname>	Uses the specified class for logging (it must implement `org.apache.tools.ant.BuildLogger`)
listener <classname>	Adds an instance of a class that can receive logging events from the build (it must implement `org.apache.tools.ant. BuildListener`)
noinput	Prevents interactive input from blocking the build process
buildfile | file | f <file>	Specifies the buildfile to be processed
D <property>=<value>	Passes a property to the build
keep-going | k	Tells Ant to execute all targets whose dependencies succeed
propertyfile <filename>	Loads all properties in a properties file; properties passed with the `D` option take precedence.
inputhandler <class>	Specifies a class to handle input request; by default input requests are handled via the standard in (stdin)
find | s <file>	Tells Ant to search for the given filename by traversing upwards from the current directory until it finds the file
nice (1..10)	Specifies a niceness value for the main thread; 1 (lowest) to 10 (highest); 5 is the default
nouserlib	Tells Ant not to load any JAR files in the user's ${user.home}/.ant/lib directory
noclasspath	Tells Ant to run without using the System's CLASSPATH

A Simple Ant Example

Figure 3-1 shows a simplified view of what a simple Ant build entails. The root of an Ant build is the `project` element, which contains one or more targets and at least one default target. In this case the simple build contains three targets named Target A, Target B, and Target C, with Target C being the default target. As shown in the zoomed view of Target B, a target can contain zero or more tasks.

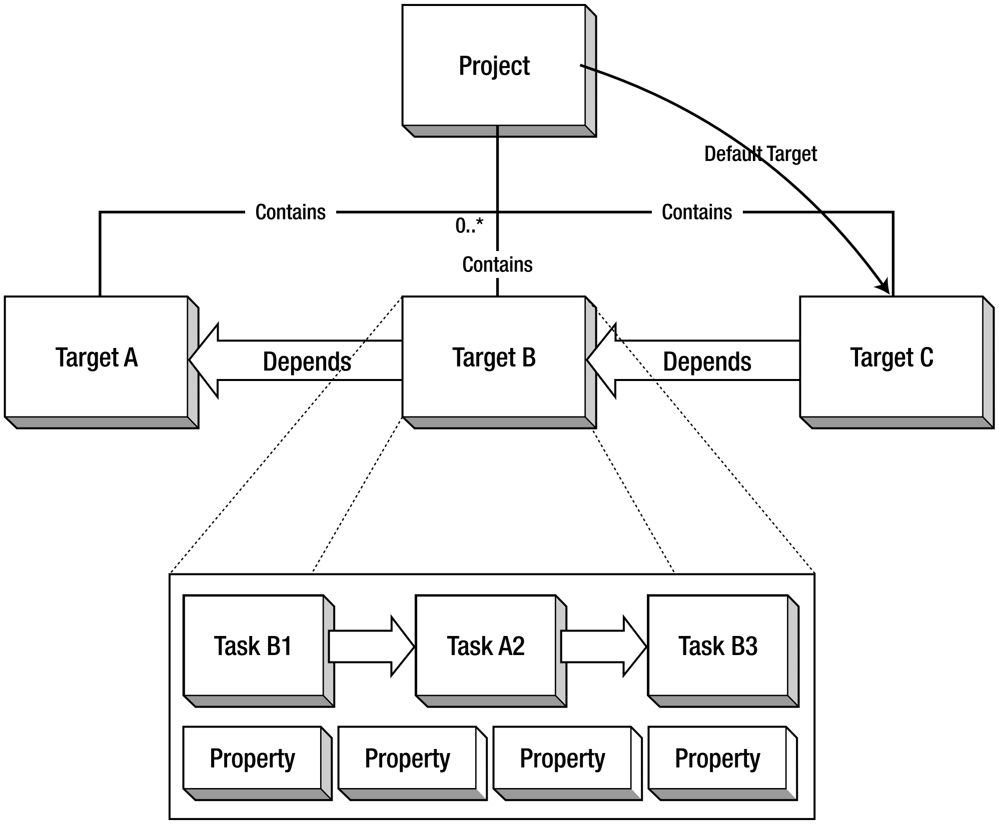

Figure 3-1. *A simplified view of an Ant build*

Ant controls the build process with a description file. In Ant the description file is typically referred to as a buildfile or build script. The Ant buildfile is an XML file whose root is the `project` element that contains child nodes that represent the targets. An Ant buildfile representing a build similar to the one depicted in Figure 3-1 would look like Listing 3-1.

Listing 3-1. *Simple Ant Buildfile*

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="Target C" name="MyProject">

    <target name="Target A" description="Performs Step A">
        <echo>Performing Step A</echo>
    </target>
    <target name="Target B" depends="Target A" description="Performs Step B">
        <echo>Performing Step B</echo>
        <echo>Echo is one of many Core Tasks</echo>
    </target>

    <target name="Target C" depends="Target B" description="Performs Step C">
        <echo>Performing Step C</echo>
    </target>

</project>
```

As you can see, for a simple buildfile the XML format makes it easier to discern targets from one another.

Project

The `project` element can have three attributes: `name`, `default`, and `basedir`. Only the `default` attribute is required, but I recommend that you use the `name` attribute especially because many IDE Ant editors use this attribute for display purposes. The `name` attribute comes in handy when dealing with more than one buildfile.

■**Best Practice** For a project with a single buildfile (build.xml) I recommend that you use the name of the project for the `name` attribute of the `project` element. For projects with multiple buildfiles I recommend that you name each one according to its intended functionality and that the `name` attribute should be the same as the filename without the .xml extension.

The `default` attribute determines the default target to be executed for the buildfile. Finally, the `basedir` attribute determines the base directory for all file-related operations during the course of a build. In the previous example it's simply the current directory where the buildfile resides, and since this is the default value, the attribute could have been omitted. This setting is important especially if you're using multiple buildfiles in different subdirectories of an application directory structure and you want a uniform way to refer to paths across all buildfiles.

■**Best Practice** Make the basedir directory the root directory of your project. This is a common convention, and it will make your buildfiles easy to understand.

The Build Stages

An Ant build has two stages: the parsing stage and the running stage. During the parsing stage the XML buildfile is parsed and an object model is constructed. This object model reflects the structure of the XML file in that it contains one project object at the root with several target objects, which themselves contain other objects representing the contents of a target such as tasks, datatypes, and properties.

■ **Note** Ant scripts can contain top-level items other than targets. These can include certain tasks and datatypes. These elements are grouped in order of appearance into an implicit target that gets executed right after the parsing process ends and before any other targets are executed.

During the runtime phase Ant determines the build sequence of targets to be executed. This sequence is determined by resolving the target's dependencies. By default, unless a different target is specified, Ant will use the default target attribute as the entry point so it can determine the build sequence.

Let's execute the sample buildfile for the sample build shown in Figure 3-1 in order to get acquainted with Ant and some of the command-line options shown in Table 3-1. First type the contents shown in the listing to a text file and save it as build.xml. To run it, simply change to the directory where the buildfile is located and type the following:

```
ant
```

The output should look like this:

```
Buildfile: build.xml

Target A:
     [echo] Performing Step A

Target B:
     [echo] Performing Step B
     [echo] Echo is one of many Core Tasks

Target C:
     [echo] Performing Step C

BUILD SUCCESSFUL
Total time: 1 second
```

The output shows that Ant executed the buildfile successfully and that it took one second to execute (execution times will vary from system to system). From the output, you can see that the targets were executed in the following sequence: Target A, Target B, and Target C. To see a bit more detail you can run Ant again using the -v command-line option, which will show you some extra information:

```
Apache Ant version 1.6.5 compiled on June 2 2005
Buildfile: build.xml
...
Build sequence for target 'Target C' is [Target A, Target B, Target C]
Complete build sequence is [Target A, Target B, Target C]
...
BUILD SUCCESSFUL
Total time: 1 second
```

First, notice that the output shows that the intended target is Target C, which was defined as the build's default target. Ant resolved the default target dependencies to arrive at the build sequence [Target A, Target B, Target C] as shown at the top of the console output.

The text enclosed in the `echo` elements in each of the targets is shown on the console as each target is executed. The echo task is one of many built-in tasks provided by Ant. For example, a quick browse of the online documentation shows that the echo task sends the text enclosed to an Ant logger. By default Ant uses the `DefaultLogger`, which is a class that "listens" to the build and outputs to the standard out. Specific loggers can be selected on the command line by using the `-logger` option. Further examination shows that the echo task is well integrated with the logging system and that it can be provided with a `level` attribute to control the level at which the message is reported.

■ **Note** I decided against regurgitating the contents of the online documentation; therefore I'll explain some of Ant's tasks in context as you set out to build the tiers of the TechConf system. The best place to learn about all the available Ant tasks is from the online manual located at `http://ant.apache.org/manual/index.html`.

The previous run of the sample script assumed that you wanted to run the default target. To run a specific target you can indicate the target in the command line as follows:

```
ant "Target A"
```

Notice that target names are case sensitive and that double quotes are required for any target names that contain spaces. The resulting output should look like this:

```
Buildfile: build.xml

Target A:
     [echo] Performing Step A

BUILD SUCCESSFUL
Total time: 1 second
```

More on Targets

Targets are meant to represent a discrete step in the build process. Targets use tasks, datatypes, and property declarations to accomplish their work. Targets are required to have a `name` attribute and an optional comma-separated list of dependent targets.

■**Best Practice** Use simple action verbs to name your targets, such as “build,” “test,” or “deploy.”

A typical buildfile is composed of several main targets: those that are meant to be called directly by the user and subtargets, which are targets that provide functionality to a main target.

■**Best Practice** Add a `description` attribute to a build’s main targets. Targets containing a description are shown in the automatic project help, which is displayed when Ant is invoked with the -p or -projecthelp command-line options. For subtargets, prefix the name with a hyphen to make it easy to differentiate them from main targets.

Targets can be conditionally executed, and for this purpose Ant supports the `if` and `unless` attributes. Targets using either or both of these are said to be conditional targets. Both `if` and `unless` take the name of a property as a value, which is a test for existence. You can see an example of this if you modify Target A from the sample buildfile and add an `if` attribute with a value of `do_a` as shown in Listing 3-2.

Listing 3-2. *Conditional Ant Target*

```
<target name="Target A" description="Performs Step A" if="do_a">
    <echo>Performing Step A</echo>
</target>
```

The target should be executed only if the Ant property by the name `do_a` exists in the context of the build. Executing the buildfile produces the following result:

```
Buildfile: build.xml

Target A:

Target B:
     [echo] Performing Step B
     [echo] Echo is one of many Core Tasks

Target C:
     [echo] Performing Step C

BUILD SUCCESSFUL
Total time: 1 second
```

Notice that the output shows the banner for Target A but that the echo tasks contained within were never executed. You can run the buildfile again using the `-D` option to pass the property `do_a` to the build as shown:

```
ant -D "do_a="
```

The output now shows that Target A is being executed. You add the double quotes around the name-value pairs for the command-line argument parser so you can recognize the end of the argument. Any value could have been passed and the results would have been the same. Remember with `if` and `unless`, the value of the property is irrelevant; what matters is whether or not the property has been defined.

Target Dependencies

From the simple buildfile shown previously you can see that targets can depend on other targets. This example shows a very simple and linear dependency chain in which Target C depends on Target B, which in turn depends on Target A.

Ant will resolve any circular dependencies and will consequently fail the build. For example, you can modify the sample script to add Target C as a dependency of Target A as shown in Listing 3-3.

Listing 3-3. *Ant Target Dependencies*

```
<target name="Target A" depends="Target C" description="Performs Step A">
    <echo>Performing Step A</echo>
</target>
```

The resulting execution of the script will produce output similar to the following:

```
Buildfile: build.xml

BUILD FAILED
Circular dependency: Target C <- Target A <- Target B <- Target C

Total time: 1 second
```

Dependencies are resolved recursively using a topological sorting algorithm. The resulting build sequence ensures that a target in the dependency chain will only get executed once. You can see a great example of this in the Ant online manual, which shows a build with dependencies as shown in Figure 3-2.

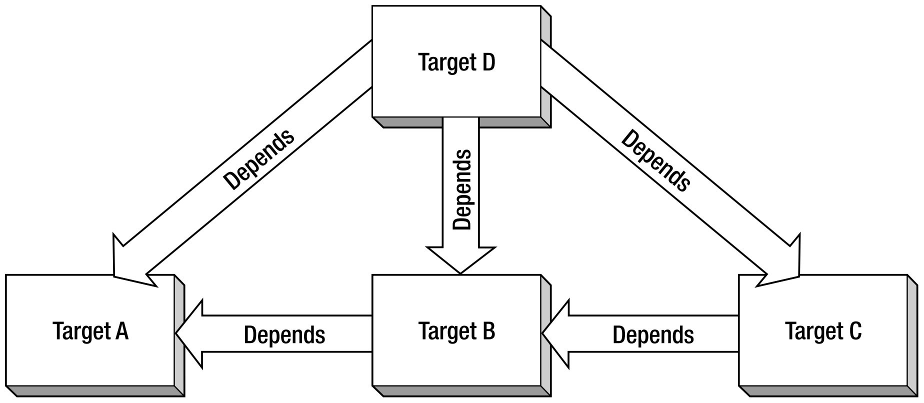

Figure 3-2. *Script dependencies*

A buildfile for the build in Figure 3-2 would look like Listing 3-4.

Listing 3-4. *Simple Ant Buildfile Showing Dependencies*

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="D" name="dependencies">
    <target name="A"/>
    <target name="B" depends="A"/>
    <target name="C" depends="B,A"/>
    <target name="D" depends="C,B,A"/>
</project>
```

Understanding how dependencies work is very important as your build process grows in complexity. Figure 3-3 shows a depiction of the dependency resolution process.

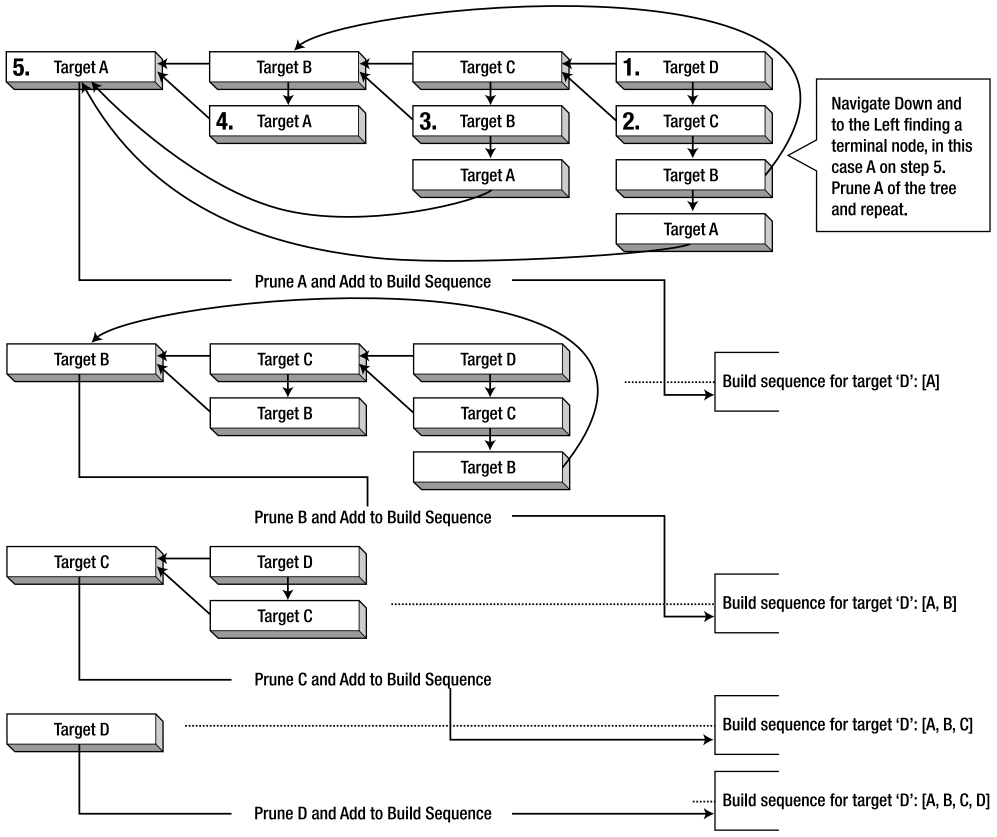
5. Target A
Target B
Target C
1. Target D
4. Target A
3. Target B
2. Target C
Target A
Target B
Target A
Navigate Down and to the Left finding a terminal node, in this case A on step 5. Prune A of the tree and repeat.
Prune A and Add to Build Sequence
Target B
Target C
Target D
Target B
Target C
Target B
Build sequence for target 'D': [A]
Prune B and Add to Build Sequence
Target C
Target D
Target C
Build sequence for target 'D': [A, B]
Prune C and Add to Build Sequence
Target D
Build sequence for target 'D': [A, B, C]
Prune D and Add to Build Sequence
Build sequence for target 'D': [A, B, C, D]

Figure 3-3. *Dependency resolution in Ant*

To test the dependencies example, save the buildfile as dependencies.xml and run it using Ant's `-f` parameter in order to indicate the buildfile as follows:

```
ant -f dependencies.xml -v
```

The output should look like this:

```
...
Buildfile: dependencies.xml
...
Build sequence for target 'D' is [A, B, C, D]
Complete build sequence is [A, B, C, D]

A:

B:

C:

D:
```

```
BUILD SUCCESSFUL
Total time: 1 second
```

■**Best Practice** Keep a build's dependencies as simple and linear as possible.

Tasks

Tasks are used within a target to achieve certain functionality. Think of a task element as a way to invoke a Java class's functionality. Ant provides a plethora of tasks that are divided in the following two categories:

- **Core:** Core tasks include basic foundational facilities needed in the build process like file manipulation, file dependencies, directory operations, source-code compilation, API document generation, archiving and packaging, XML file manipulation, SQL execution, and others.
- **Optional:** This includes tasks for some commercial products (like EJB/J2EE servers and third-party Version Control Systems) as well as nonbuild-specific tasks like unit testing, XML validation, and others.

Properties

Ant provides the ability for a project to have a set of properties. Properties are simple strings that you can access using the ${propertyName} notation. Whether you need to specify the location of a needed library many times or the name of a CVS repository, properties give you the flexibility to defer until runtime a set of values to be used in the build.

There are several ways to set a property. You can set it individually to the Ant buildfile via the `D` command-line option (see Table 3-1), or in bulk, from standard Java properties files by using the `propertyfile` option.

There are also several tasks that deal with properties. The property task enables the setting of a property by name. All property tasks are idempotent, which means that once a property's value has been set it will remain unchanged for the remainder of the build. The immutability of properties in Ant is often a source of confusion, because as developers you're often used to thinking with the use of variables.

■**Note** The <ant> and <antcall> tasks both span a new build by calling another buildfile. The <ant> task calls an external buildfile, and the <antcall> tasks calls a target on the current buildfile. Since version 1.6, Ant provides the <macro-def>, <import>, and <subant> tasks, which eliminate the need for using <ant> in most cases.

The simplest way to set a property's value is to use the property task. For example, to set a property named `src`, which could be later accessed using `${src}`, you would use the property task as follows:

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

The `src` property would be an absolute path that refers to the location of the src directory relative to the basedir directory.

■ **Best Practice** Properties should be used with care. The two main uses of properties are for items whose value might change from build to build or for items whose value is calculated and used more than once during the build.

Many Ant properties are also available implicitly and are composed from the system properties, such as `${java.version}`.

For any but the simplest project you can load a property file using the `file` attribute of the property task, thereby taking into account differences in user configurations, as follows:

```
    <property file="build.properties"/>
```

Other tasks that deal directly with properties include the following:

- **LoadProperties:** Loads the contents of a file as properties (equivalent to using the `file` attribute for the property task).
- **LoadFile:** Load a text file into a single property.
- **XMLProperty:** Loads properties from an XML file. See the Ant documentation for the specific format of the XML file.
- **EchoProperties:** Displays all available properties in the project.

Many other tasks use properties as a way to take parameters in or out. For example, a common practice is for a task to have an attribute that takes the name of an inexistent property to be set in case of a specific event such as the possibility of the task failing.

■ **Best Practice** I recommend using a properties file named build.properties to store any overridden default values. This property file shouldn't be kept in the source-code repository but instead should add a sample properties file named build.properties.sample along with instructions on how to configure an individual build.properties file.
