



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
<code>help h</code>	Prints the help message showing all available options
<code>projecthelp p</code>	Displays all targets for which the <code>description</code> attribute has been set
<code>version</code>	Prints the version of Ant
<code>diagnostics</code>	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 <code>org.apache.tools.ant.BuildLogger</code>)
listener <classname>	Adds an instance of a class that can receive logging events from the build (it must implement <code>org.apache.tools.ant.BuildListener</code>)
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 <code>\$(user.home)/.ant/lib</code> 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.

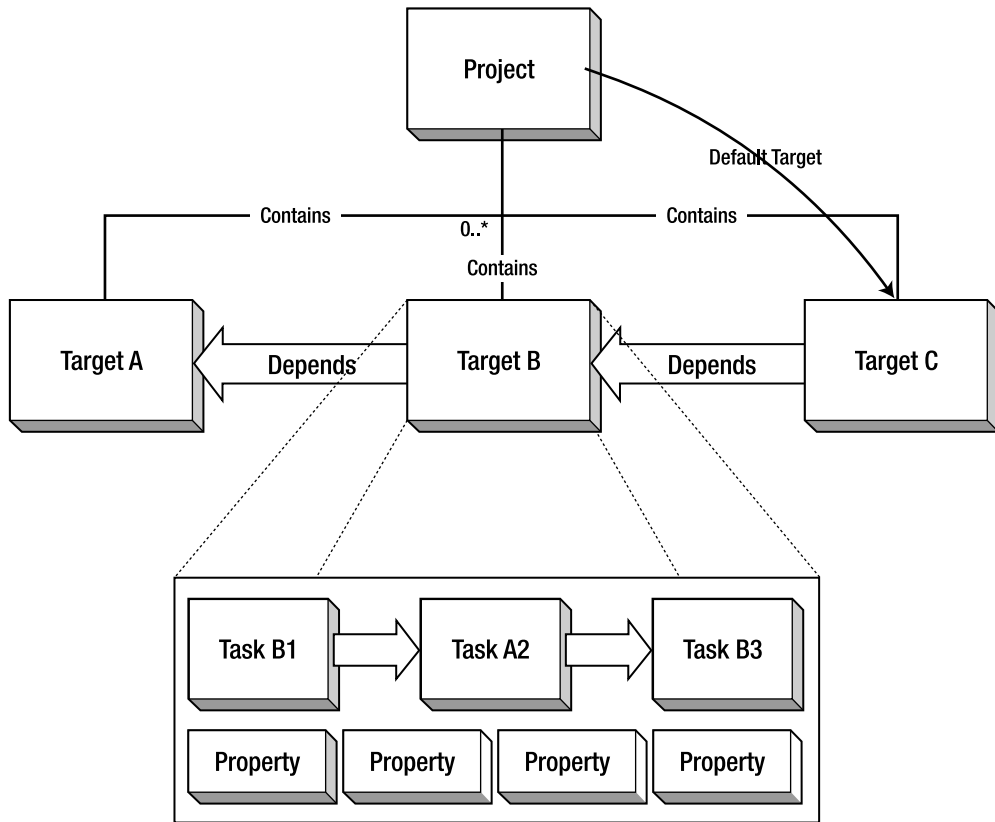


Figure 3-1. A simplified view of an Ant build

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.

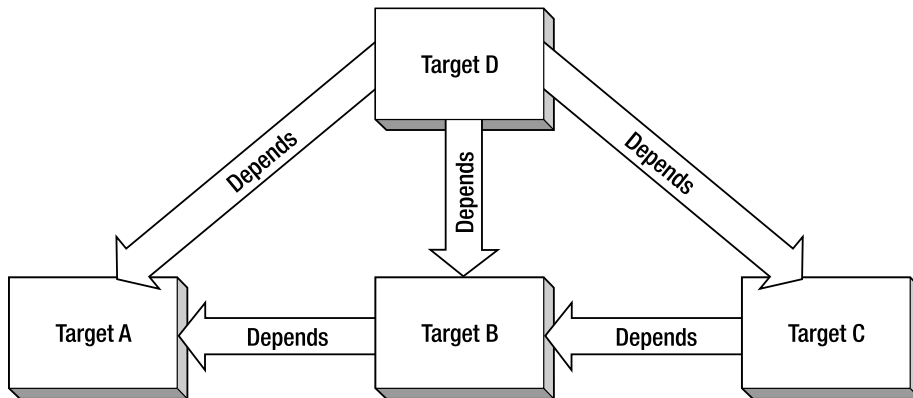


Figure 3-2. Script dependencies

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.

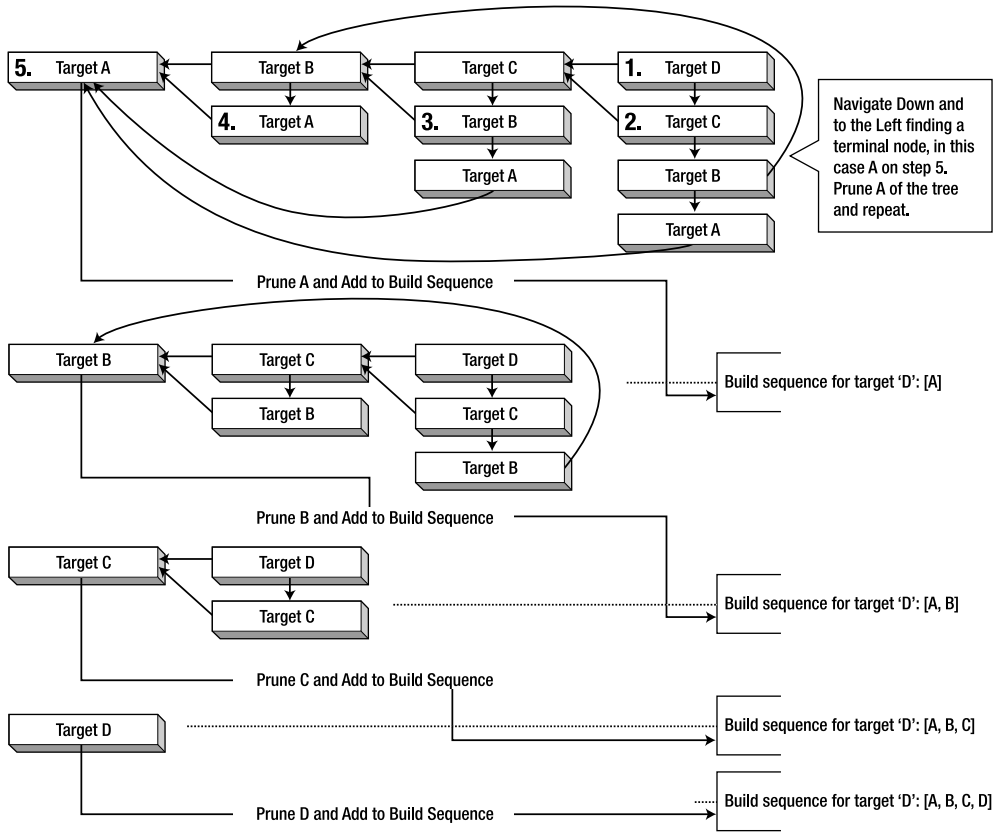


Figure 3-3. *Dependency resolution in Ant*

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.

Datatypes

Ant's datatypes are primitive constructs that provide frequently required information in the processing of a buildfile. Their purpose is to simplify a task by encapsulating some information required and providing a simple way to manipulate it.

Several of Ant's built-in datatypes provide a structure that encapsulates information about a set of related resources such as files, environment variables, or even complex mappings between input and output files. Knowing how to properly use the Ant's datatypes will help you keep your buildfiles simple and efficient.

Datatypes and Properties in Action: A Simple Example

Many of Ant's tasks need to manipulate a file or groups of files. A typical need in a build is to specify a set of JAR files to be included in the classpath for certain tasks. Imagine that you're building a simple application with a directory structure, as shown in Figure 3-4.



Figure 3-4. Sample directory structure for datatypes and properties

The sample buildfile in Listing 3-5 shows a build for which two path structures (datatypes) are defined, one with an id of `class.path` and the other with an id of `all.source.path`. These two datatypes are then used in the target named “compile”, which uses the `javac` task to compile the classes referenced by the path reference by the id `all.source.path`.

Listing 3-5. Simple Ant Buildfile Showing Datatypes

```

<?xml version="1.0"?>
<project name="My Project" default="all" basedir=".">
...
  <property name="lib" location="lib"/>
  <property name="src" location="src"/>
  <property name="classes" location="classes"/>
  <property name="build" location="build"/>

  <property name="src-java" location="${src}/java"/>
  <property name="src-test" location="${src}/test"/>
  <property name="some-lib" location="${lib}/some-lib"/>
...
  <path id="class.path">
    <fileset dir="${lib}">
      <include name="*.jar"/>
    </fileset>
  </path>

```

```

    </fileset>
    <fileset dir="{some-lib}">
        <include name="*.jar"/>
    </fileset>
</path>

<path id="all.source.path">
    <pathelement path="{src-java}"/>
    <pathelement path="{src-test}"/>
</path>
...
<target name="compile" description="Compiles all sources.">
...
    <javac
        destdir="{classes}"
        classpathref="class.path"
        debug="on"
        deprecation="on"
        optimize="off">
        <src>
            <path refid="all.source.path"/>
        </src>
    </javac>
</target>

```

The `class.path` path structure uses two instances of the `fileset` datatype to group under a common classpath all the JAR files included in the directories referenced by the `lib` and `struts-lib` properties. The `pathelement` is an example of an indispensable datatype that enables you to reuse path information in your builds. The `fileset` datatype is a typical example of Ant's pathlike structures. It encapsulates a group of files defined via nested `patternset` structures. For example, to create a `fileset` that includes all JAR files under the `{lib}` directory, you can use the following `fileset` definition:

```

<fileset dir="{lib}">
    <patternset>
        <include name="*.jar"/>
    </patternset>
</fileset>

```

The `fileset` datatype contains an implicit `patternset` structure, which means that you can use shorthand to rewrite the `fileset` definition as follows:

```

<fileset dir="{lib}">
    <include name="*.jar"/>
</fileset>

```

We can further compact the `fileset` definition by using the `include` as a property rather than as a nested element:

```

<fileset dir="{lib}" include="*.jar" />

```

The path datatype can also make use of nested path elements, as shown in the definition of the `all.source.path` path structure. It uses the path element datatype to reference the locations defined by `src-java` and `src-generated` properties.

Path is a typical Ant pathlike structure. When dealing with paths or classpaths, Ant's task makes use of pathlike structures to perform its function. In the previous example, you can see that the two path elements defined at the top of the buildfile are then used by reference in the context of the `javac` task. The `class.path` path is passed to the `classpathref` attribute of `javac` to determine the classpath for compilation and the `all.source.path` is used by creating a new path element, which is nested inside the `src` nested element of the `javac` task.

As a build's complexity increases so do the patterns for selecting files. Pathlike structures enable the reuse of path information and help keep the growth of buildfiles under control.

Note One of the criteria used in choosing many of the tools in this chapter was whether the tool provided an Ant task.

Case Study: Building TechConf with Ant

To set the stage for the development throughout the rest of the book, you need to first create a suitable directory structure (see Figure 3-5) as well as an initial Ant buildfile for the TechConf system.

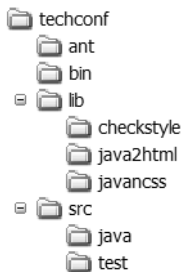


Figure 3-5. Sample directory structure for the TechConf project

The project's root directory is TechConf. Under this directory you'll place the project's main buildfile, named `build.xml`. The subdirectories under TechConf are organized as follows:

- **lib:** Contains any libraries required at runtime by the application(s)
- **ant:** Contains Ant macrodef in a single file, `macros.xml`
- **src:** The root directory for all non-generated sources
- **src/java:** The root directory for all non-J2EE Java sources
- **src/test:** The root directory for all test classes
- **src/j2ee:** The root directory for all J2EE source files

Now that you have a suitable directory structure, your next step should be to start putting together the TechConf buildfile. The project element contains the name of your project and a nested description element.

Best Practice Use the `description` element, which allows you to enter a detailed description of the project. This description is shown on the console when invoking Ant with the `-projecthelp` or `-p` command-line option.

The default target will be the `all` target, which you'll develop later in the chapter. The `basedir` is set to be the directory where the buildfile resides, which in this case is the TechConf directory.

```
<?xml version="1.0"?>
<project name="TechConf" default="all" basedir=".">

  <description>
    This build script was developed to be a generic enterprise development
    build script using ANT 1.6.5 (ant.apache.org). To customize it or use it for
    other projects modify the build.properties file.
  </description>
  ...
```

Next, properties are defined for the created directories. Notice that you can define properties using other properties as with the `lib-dev` property. Properties that represent a directory are defined using the `location` attribute instead of the `value` attribute. The `location` attribute gets resolved to the full path relative to the `basedir` specified in the project element.

Best Practice Making all paths relative to the project's `basedir` directory and avoiding the use of absolute paths guarantees that your buildfile will work anywhere. If your build depends on a resource whose location might change from environment to environment, you should place the location of said resource in a properties file or use environment variables such as `${os.name}`.

The build directory is the root directory for all products of the build process, such as the classes directory, where the results of compiling the classes under `src/java` will be placed.

```
<!-- ===== -->
<!-- Initialization -->
<!-- ===== -->
<property file="build.properties"/>

<!-- ===== -->
<!-- Directories -->
```

```

<!-- ===== -->
<property name="build" location="build" />
<property name="lib" location="lib" />

<!-- Source -->
<property name="src" location="src" />
<property name="src-java" location="${src}/java" />
<property name="src-test" location="${src}/test" />
<property name="src-j2ee" location="${src}/j2ee" />

<property name="docs" location="docs" />
<property name="docs-api" location="${docs}/api" />
<property name="docs-html-source" location="${docs}/source" />
<property name="docs-test" location="${docs}/tests" />
<property name="src-web" location="web" />

```

Paths representing all the JAR files under the lib directory (class.path) and all class files under the classes directory are created.

Best Practice A common practice in Ant buildfiles is to have an init task that all other tasks depend on. I advocate not using the init task for setting up properties, loading properties files, paths, patternsets, or taskdefs. Instead, just place them before the first target, and they will be added to the implicit target. As mentioned earlier, the contents of the implicit target always get called and you don't have to remember making all other targets dependent on an init target.

A patternset is also used to filter a directory for non-source files. In the case where resources are part of the source directory such as property files or images, a patternset can be used to copy them to the location of the compiled classes which will require said resources.

```

<!-- Paths -->
<path id="class.path">
  <fileset dir="${lib}">
    <include name="*.jar"/>
  </fileset>
</path>

<path id="app.class.path">
  <pathelement location="${classes}"/>
  <path refid="class.path"/>
</path>

<!-- Patternsets -->
<patternset id="non.source.set">
  <exclude name="**/*.java"/>
  ...

```

```

    <exclude name="**/read-me.txt"/>
    <exclude name="**/package.html"/>
</patternset>

```

Compiling

Now it's time to add the first target to the buildfile, the compile target. This target will make use of the `javac` task, which is a wrapper to the `javac` command. In Listing 3-6, notice that before the `javac` task is invoked, all files under the `src-java` directory that match the `patternset non.source.set` are copied to the `classes` directory. This is done so that any resources such as Java properties files, images, and others are available to the compiled code under the `classes` directory. This is a common practice for many IDEs.

Listing 3-6. Compile Target

```

<!-- ===== -->
<!-- Target: compile -->
<!-- Compiles all classes -->
<!-- MUST use JDK 1.5 compiler -->
<!-- ===== -->

<target
  name="compile"
  depends="compile-init"
  description="Compiles all classes (JDK1.5)">
  <javac
    destdir="${classes}"
    classpathref="class.path"
    debug="on"
    deprecation="on"
    optimize="off"
  >
    <src>
      <path refid="all.source.path" />
    </src>
  </javac>
</target>

<target name="compile-init">
  <target-banner target="compile"/>
  <mkdir dir="${classes}"/>
  <copy todir="${classes}">
    <fileset dir="${src-java}">
      <patternset refid="non.source.set" />
    </fileset>

```

```

        <fileset dir="dd">
            <include name="*.properties"/>
        </fileset>
    </copy>
</target>

<target name="compile-clean">
    <delete includeemptydirs="true">
        <fileset dir="${classes}" includes="**/*"/>
    </delete>
</target>

```

Notice that we've added two more targets other than compile. These are compile-init and compile-clean. The compile-init target simply creates the classes directory by making use of the mkdir task. The compile-clean target uses the delete task to remove the directory and all of its contents.

Best Practice For each main target in the buildfile, add a target-init and a target-clean, where target is the name of the main target. This makes it fairly straightforward to determine the resources needed and created by a target and also makes it easier to maintain large buildfiles. For simple buildfiles a single clean target will usually suffice.

Buildfile Reuse with Macros

If you paid close attention to the compile-init target shown previously, you've notice that the first line is:

```
<target-banner target="compile"/>
```

The element target-banner is not a standard Ant task or a third-party task; it is a macro definition contained in a separate XML named macros.xml. Macro definitions, a feature introduced in Ant 1.6, help you avoid the tedious copy-paster reuse and enable you to modularize your builds. Macros can be invoked anywhere in the buildfile, and the macro definitions can be parameterized. In order to enable our build to use the macros, we use the import task as shown next:

```

<!-- ===== -->
<!-- Imports -->
<!-- ===== -->
<import file="ant/macros.xml"/>

```

Let's take a look at the file macros.xml, which is located in the ant directory at the root of the TechConf project and shown in Listing 3-7.

Listing 3-7. *Ant Macros File*

```

<?xml version="1.0"?>
<project name="techconf-ant-macros" default="test-macros" basedir="..">

    <!-- ===== -->
    <!-- Prints a banner for the target being executed -->
    <!-- ===== -->
    <macrodef name="target-banner">
        <attribute name="target"/>
        <attribute name="message" default="" />
        <sequential>
            <echo>=====</echo>
            <echo>Executing Target @<{target}</echo>
            <echo>@<{message}</echo>
            <echo>=====</echo>
        </sequential>
    </macrodef>

    <!-- ===== -->
    <!-- Test the macros -->
    <!-- ===== -->
    <target name="test-macros">
        <target-banner target="Compile"/>
        <target-banner target="Testing" message="This is a sample message"/>
    </target>

    ...
</project>

```

In `macros.xml` we define the `target-banner` macrodef. Ant macrodefs can take attributes; in this case there are two attributes, `target` and `message`. As you can guess from the snippet shown, the `target-banner` macrodef uses the `echo` task to print a banner to the console that informs the user of the current target being executed and also prints an optional message. The attributes are defined using the `attribute` element. The `target` attribute is required, but the `message` attribute is optional since it has a default value. Macro attributes are mutable and are expanded via `@<{attrname}</code>.`

Note Macro attributes `@<{attr}</code> are expanded before Ant properties $(property)</code>. This is important if you are using properties in your macros.`

On its own, the `macros.xml` file behaves just like any other Ant buildfile. If we execute the `macros.xml` file using Ant as follows:

```
ant -f macros.xml
```

The default target `test-macros` will execute, producing the following output:

```
Buildfile: macros.xml

test-macros:
  [echo] =====
  [echo] Executing Target Compile
  [echo] =====

  [echo] =====
  [echo] Executing Target Testing
  [echo] This is a sample message
  [echo] =====

BUILD SUCCESSFUL
Total time: 1 second
```

As you can see, `macrodef` in combination with the `import` task can help you create reusable, modularized Ant functionality that will help you keep your buildfiles simple. For the `TechConf` project we will use the `macros.xml` file to house most of the tasks peripheral to the build process. In the remainder of this chapter we will continue to enhance both the `build.xml` file and the `macros.xml` file to create a J2EE build system that’s modular and reusable.

Javadoc Generation

For proper team communication and for enabling code reuse you must have a consistent, up-to-date set of API documentation. The Javadoc tool has existed for as long as Java has been around, and all developers are well acquainted with it. The problem has been that developers feel that they can run Javadoc only after they are finished with the code (which might be never). Running Javadoc at the end of a project provides very little help to others in the team and moves documentation to the end of process, when it isn’t as helpful (waterfall).

With Ant you can ensure that Javadoc is generated as part of the daily build and that you don’t hide the documentation process until the “end” of the development phase. The Ant Javadoc task provides a convenient way to generate Javadoc from within Ant.

To incorporate Javadoc generation into the `TechConf` build we will enhance the `macros.xml` file with a generic `macrodef` that defaults most of the common settings used with the Javadoc task. There are four required attributes—`source.path`, `class.path`, `dest`, `year`—and the optional `company` attribute, as shown in Listing 3-8.

Listing 3-8. Javadoc Macrodef

```

<!-- ===== -->
<!-- JavaDocs -->
<!-- ===== -->

<macrodef name="generate-javadoc" description="Generate JavaDocs.">
  <attribute name="company" default="Integrallis Software, LLC."/>
  <attribute name="source.path" />
  <attribute name="class.path" />
  <attribute name="year" />
  <attribute name="dest" />
  <sequential>
    <javadoc
      destdir="@{dest}"
      author="true"
      version="true"
      use="true"
      windowtitle="\${ant.project.name}"
      sourcepathref="@{source.path}"
      classpathref="@{class.path}"
      packagenames="*.*)"
      Verbose="false">
      <doctitle><![CDATA[<h1>\${ant.project.name}</h1>]]></doctitle>
      <bottom>
      <![CDATA[<i>Copyright &#169; @{year} @{company} All Rights Reserved.</i>]]>
      </bottom>
      <tag name="todo" scope="all" description="To do:" />
    </javadoc>
  </sequential>
</macrodef>

```

Notice that the doctitle and the bottom nested elements make use of the XML character data (CDATA) section in order to be able to use HTML markup and not have it interfere with the markup of the buildfile.

To use the generate-javadoc macrodef in the TechConf buildfile we can create a target in our build.xml as shown in Listing 3-9.

Listing 3-9. Generate-docs Target

```

<!-- ===== -->
<!-- Target: docs -->
<!-- Generates documentation artifacts -->
<!-- ===== -->

```

```

<target name="generate-docs" description="Generates all documentation">
  <target-banner target="generate-docs"/>
  <generate-javadoc
    class.path="class.path"
    dest="${docs-api}"
    source.path="all.source.path"
    year="2005"
  />
</target>

<target name="generate-docs-clean">
  <delete dir="${docs}" />
</target>

```

It is easy to see how much cleaner your main buildfile can become by using macrodefs effectively. For the rest of the chapter we will use the same technique to continue enhancing the build with other functionality.

Checking Code Conventions with Checkstyle

Even if you're using a formatting tool either at build time or with your favorite IDE, there are still style checks beyond the realm of formatting. Checkstyle is a tool that enables code to be checked against a convention. Checkstyle supports the Sun convention by default, although it can check for more than just simple formatting. For example, it can check for illegal regular expressions in the code, inline conditionals, double-checked locking, and other idioms or patterns that might be considered unsafe or problematic.

You can download Checkstyle from <http://checkstyle.sourceforge.net>. At the root of the Checkstyle distribution you'll find the `checkstyle-all-4.0.jar` file. Place this file in a directory named `checkstyle` under the `lib` directory of the TechConf project directory. The file containing the XML configuration representing the Sun convention is named `sun_checks.xml`, and it's located under the `docs` directory of the distribution directory. Copy this file to the `lib/checkstyle` directory also.

Checkstyle writes its output to the standard out by default or to a file in plain text or XML format. The Checkstyle distribution also provides several Extensible Stylesheet Language (XSL) stylesheets that can be used to convert the XML reports to HTML format for easier viewing. You can find these stylesheets in the Checkstyle distribution under the `contrib` directory. Copy the `checkstyle-noframes-sorted.xsl` file to the `lib/checkstyle` directory.

To use Checkstyle from within Ant, you first need to load the checkstyle task. As with the Javadoc task we will incorporate the checkstyle task in a macrodef contained in the `macros.xml` file. First we need to make the checkstyle task available to the `macros.xml` file by defining a `taskdef` for it:

```

<path id="checkstyle.class.path">
  <fileset dir="lib/checkstyle">
    <include name="*.jar"/>
  </fileset>
</path>

```

```
<taskdef
  resource="checkstyle.task.properties"
  classpathref="checkstyle.class.path"
/>
```

The macrodef `generate-checkstyle` takes two required attributes: `src` to determine the directory containing the source files to check and `checkstyle-reports` for the location to place the generated reports. The rest of the attributes—`checkstyle-checks-file`, `checkstyle-xml-report-file`, `checkstyle-html-report-file` and `checkstyle-stylesheet`—are all optional. Notice that some of the default values for the optional attributes are generated from the values of the required attributes.

The `checkstyle` macrodef uses the `checkstyle` task to check the code under the `@{src}` directory against the conventions specified by the file `@{checkstyle-checks-file}` and uses a formatter of type XML to generate the report referred to in `@{checkstyle-xml-report-file}`. The `failureProperty` attribute is the property that's set if there are any errors encountered during the checking process. You can use this value to determine if any action is to be taken in the case of an error, such as emailing the report. The second part of the target uses the `style` task to transform the generated XML into an HTML report. The `generate-checkstyle` macrodef is shown in Listing 3-10.

Listing 3-10. *Generate-checkstyle Macrodef*

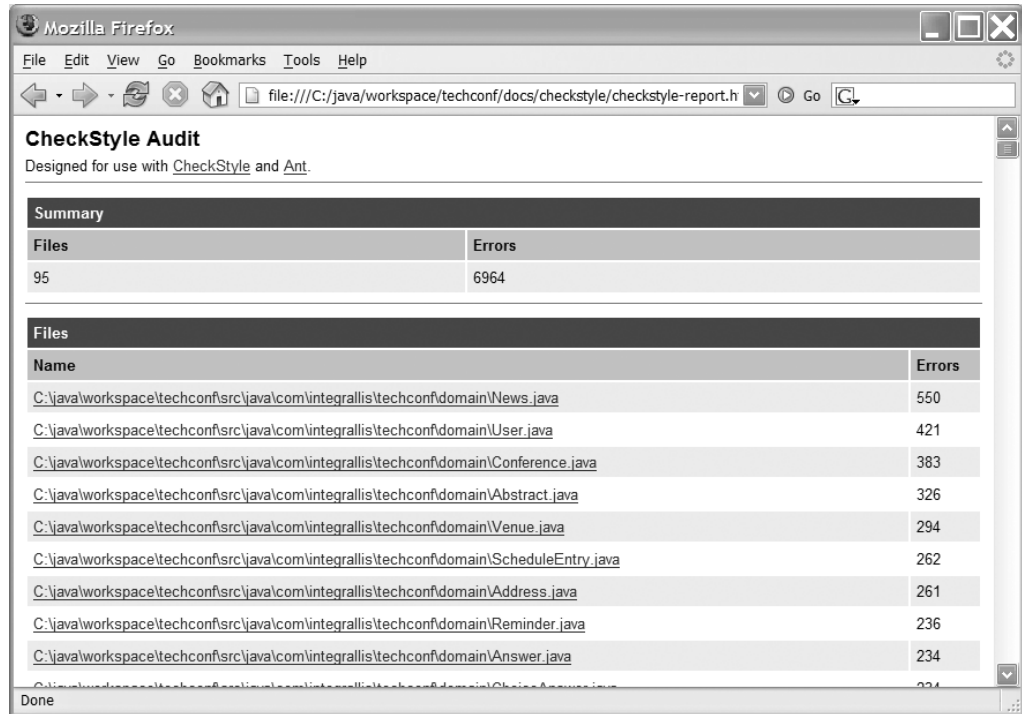
```
<!-- ===== -->
<!-- CheckStyle -->
<!-- ===== -->
<macrodef name="generate-checkstyle"
  description="Generates Code Convention Violations Report.">
  <attribute name="src" />
  <attribute name="checkstyle-reports" />
  <attribute name="checkstyle-checks-file"
    default="lib/checkstyle/sun_checks.xml"/>
  <attribute name="checkstyle-xml-report-file"
    default="@{checkstyle-reports}/checkstyle-report.xml"/>
  <attribute name="checkstyle-html-report-file"
    default="@{checkstyle-reports}/checkstyle-report.html"/>
  <attribute name="checkstyle-stylesheet"
    default="lib/checkstyle/checkstyle-noframes-sorted.xsl"/>
  <sequential>
    <mkdir dir="@{checkstyle-reports}" />
    <checkstyle
      config="@{checkstyle-checks-file}"
      failureProperty="checkstyle.failure"
      failOnViolation="false"
    >
      <formatter type="xml" tofile="@{checkstyle-xml-report-file}"/>
      <fileset dir="@{src}" includes="**/*.java"/>
    </checkstyle>
```

```

<style
  in="@{checkstyle-xml-report-file}"
  out="@{checkstyle-html-report-file}"
  style="@{checkstyle-stylesheet}"
/>
</sequential>
</macrodef>

```

A sample Checkstyle report is shown in Figure 3-6.



CheckStyle Audit
Designed for use with [CheckStyle](#) and [Ant](#).

Summary	
Files	Errors
95	6964

Files	
Name	Errors
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\News.java	550
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\User.java	421
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Conference.java	383
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Abstract.java	326
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Venue.java	294
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\ScheduleEntry.java	262
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Address.java	261
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Reminder.java	236
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\Answer.java	234
C:\java\workspace\techconf\src\java\com\integrallis\techconfdomain\ChoiceAnswer.java	234

Figure 3-6. Checkstyle HTML report

Generating Source-Code Metrics

Although I don't advocate counting code lines, classes, or methods as a measure of a project's success, static code analysis can help you pinpoint some areas of unnecessary complexity that can lead to the discovery of potential bugs or high-maintenance code.

JavaNCSS is a simple source-measurement tool for Java that provides that following basic types of analysis:

- **NCSS:** Noncommenting source statements provide counts of many features of the code such as lines of code, declarations, methods, statements, constructors, and so on.
- **CCN:** Cyclomatic complexity number (McCabe metric). McCabe's cyclomatic complexity metric looks at a program's control flow graph as a measure of its complexity.

You can download JavaNCSS from www.kclee.de/clemens/java/javancss/ as a simple ZIP file that includes an Ant task. Place all JAR files located under the distribution's lib directory in a directory named javancss under the lib directory of the TechConf project. Next, create a directory named xslt under the lib/javancss and copy the contents of the xslt directory under the JavaNCSS distribution directory.

To make the JavaNCSS Ant task available in the macros.xml file we add the following path and taskdef definitions:

```
<path id="javancss.class.path">
  <fileset dir="lib/javancss">
    <include name="*.jar"/>
  </fileset>
</path>

<!-- Javancss - kclee.com/clemens/java/javancss -->
<taskdef
  name="javancss"
  classname="javancss.JavancssAntTask"
  classpathref="javancss.class.path"
/>
```

The Ant task can generate a report in plain text of the XML format. Similar to the checkstyle macrodef, you'll use the style task to transform the reports to HTML, as shown in Listing 3-11.

Listing 3-11. *Generate-metrics Macrodef*

```
<!-- ===== -->
<!-- Metrics -->
<!-- ===== -->
<macrodef name="generate-metrics">
  <attribute name="src" />
  <attribute name="report-name" />
  <attribute name="report-dir" default="." />
  <attribute name="xml-report" default="@{report-dir}/@{report-name}.xml" />
  <attribute name="html-report" default="@{report-dir}/@{report-name}.html" />
  <attribute name="stylesheet" default="lib/javancss/xslt/javancss2html.xsl" />
  <sequential>
    <mkdir dir="@{report-dir}" />
    <javancss
      srcdir="@{src}"
      includes="**/*.java"
      generateReport="true"
      outputfile="@{xml-report}"
      format="xml"
      functionMetrics="false"
    />
```

```

<style
  in="@{xml-report}"
  out="@{html-report}"
  style="@{stylesheet}"
/>
</sequential>
</macrodef>

```

The generated HTML reports look like the one shown in Figure 3-7.

JavaNCSS Analysis

Designed for use with [JavaNCSS](#) and [Ant](#).

Packages

Nr.	Classes	Functions	NCSS	Javadocs	Package
1	24	314	1342	250	com.integrallis.techconf.domain
2	4	5	17	4	com.integrallis.techconf.dto
3	3	8	17	3	com.integrallis.techconf.service
4	1	4	50	0	com.integrallis.techconf.web
	32	331	1426	257	Total

Packages	Classes	Functions	NCSS	Javadocs	per
4.00	32.00	331.00	1,426.00	257.00	Project
	8.00	82.75	356.50	64.25	Package
		10.34	44.56	8.03	Class
			4.31	0.78	Function

Objects

Nr.	NCSS	Functions	Classes	Javadocs	Class
1	36	4	0	0	com.integrallis.techconf.web.ConferenceSummaryServlet
2	84	22	0	19	com.integrallis.techconf.domain.Abstract
3	43	10	0	7	com.integrallis.techconf.domain.AbstractStatus
4	54	17	0	14	com.integrallis.techconf.domain.Address
5	64	16	0	13	com.integrallis.techconf.domain.Answer
6	26	5	0	2	com.integrallis.techconf.domain.Attendee
7	16	6	0	1	com.integrallis.techconf.domain.BasePrincingRule
8	55	14	0	11	com.integrallis.techconf.domain.Booth
9	64	16	0	13	com.integrallis.techconf.domain.ChoiceAnswer
10	50	12	0	9	com.integrallis.techconf.domain.GroupPrincingRule
11	144	22	1	31	com.integrallis.techconf.domain.News
12	49	12	0	9	com.integrallis.techconf.domain.PresentationLevel
13	49	12	0	9	com.integrallis.techconf.domain.PresentationTopic

Done

Figure 3-7. A JavaNCSS HTML report

Generating Browsable Source Code

One useful feature for sharing knowledge about a project is the ability to generate a browsable version of the code for viewing online. Many open source projects use this as a way to allow others to view the source to a particular class without having to download a source distribution

or having to use CVS. Java2Html is a tool that enables you to take a Java class or a snippet of Java code and generate a syntax-highlighted HTML version of the code.

The Java2Html tool can be obtained from www.java2html.de as a single ZIP file that contains one JAR file (`java2html.jar`). As with the other third-party Ant tasks, place the JAR file in a directory named `java2html` under the `TechConf lib` directory.

As mentioned previously, you should load the task using the `taskdef` task. First we add the path and `taskdef` to the `macros.xml` file as shown here:

```
<path id="java2html.class.path">
  <fileset dir="lib/java2html">
    <include name="*.jar"/>
  </fileset>
</path>

<!-- Java2Html - java2html.de -->
<taskdef
  name="java2html"
  classname="de.java2html.anttasks.Java2HtmlTask"
  classpathref="java2html.class.path"
/>
```

The generated HTML source will be placed under the location pointed to by the property `${browseable-source}`, as shown in Listing 3-12.

Listing 3-12. *Generate-html Macrodef*

```
<!-- ===== -->
<!-- Generates browsable source code in HTML format -->
<!-- ===== -->
<macrodef name="generate-html"
  description="Generates browsable HTML version of the source code." >
  <attribute name="src"/>
  <attribute name="dest"/>
  <sequential>
    <mkdir dir="@{dest}" />
    <java2html
      srcdir="@{src}"
      destdir="@{dest}"
      includes="**/*.java"
      outputFormat="html"
      tabs="4"
      style="eclipse"
      showLineNumbers="true"
      showFileName="true"
      showTableBorder="true"
```



```
        includeDocumentHeader="true"  
        includeDocumentFooter="true"  
        addLineAnchors="true"  
        lineAnchorPrefix="fff"  
    />  
</sequential>  
</macrodef>
```

Figure 3-8 shows an example of an HTML page generated by Java2Html.

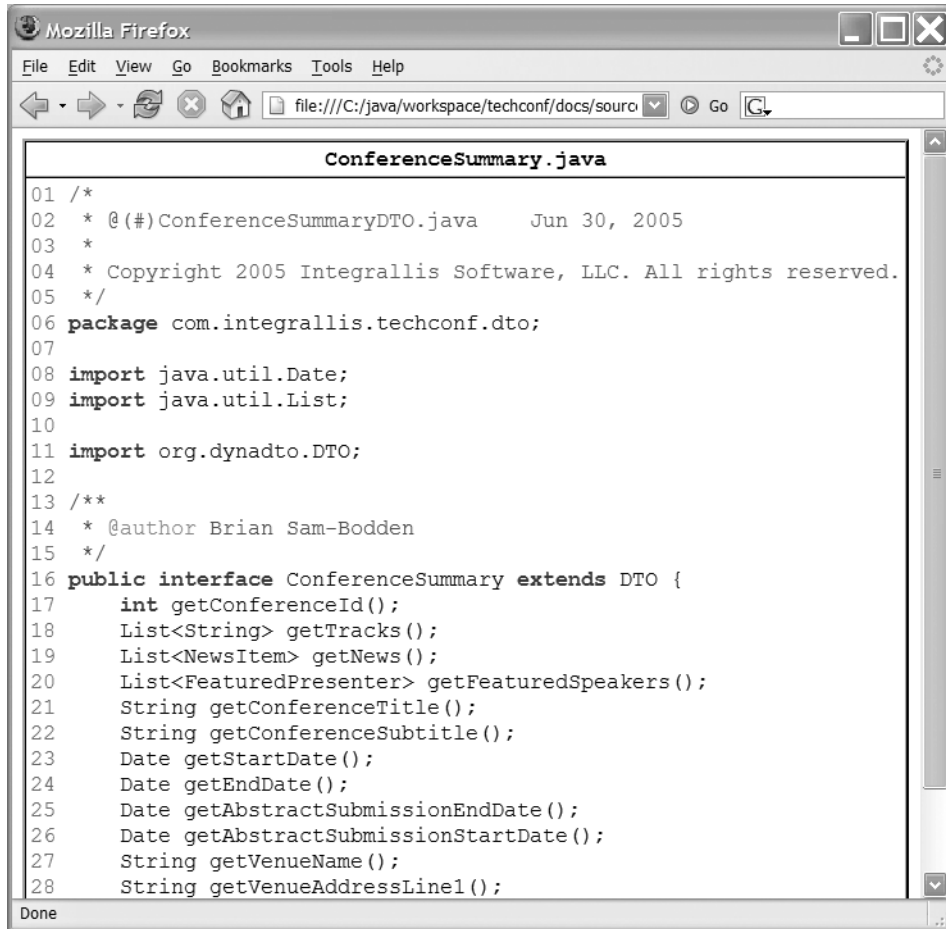


Figure 3-8. An HTML page generated by Java2Html

Document Generation

Finally we can group all of the document-generation tasks under one single target in the build.xml file as shown in Listing 3-13.

Listing 3-13. *Generate-docs Target*

```

<!-- ===== -->
<!-- Target: docs -->
<!-- Generates documentation artifacts: Javadoc, Browsable HTML, etc. -->
<!-- ===== -->

<target name="generate-docs"
  description="Generates all documentation">
  <target-banner target="generate-docs"/>
  <generate-javadoc
    class.path="class.path"
    dest="{docs-api}"
    source.path="all.source.path"
    year="2005"
  />
  <generate-html
    src="{src}"
    dest="{docs-html-source}"
  />
  <generate-checkstyle
    src="{src}"
    checkstyle-reports="{checkstyle-reports}"
  />
  <generate-metrics
    src="{src}"
    report-dir="{metrics-reports}"
    report-name="{ant.project.name}"
  />
</target>

```

This single document-generation target makes the buildfile simpler. The use of macrodefs makes the main buildfile less verbose. I decided to group all document generation-tasks so that users of the build have to deal with only a single, simple target for all document-generation tasks.

Cleaning Up

The build process produces many files and directories. Getting the project directory to the same state as when the source was checked out of a repository is important for determining what has changed. Many Ant users recommend having a “clean” target that can remove all the products of the build process.

The problem with this approach is that for large builds it’s easy to accidentally delete files that are needed, and it’s also easy to miss files or directories that need to be deleted. For this reason you should include a clean sub target for each main target in the buildfile. By doing this you’ll easily be able to determine what needs to be clean at the target level. Then for the global clean target you can simply invoke all individual clean sub targets by invoking them using the antcall task (or by listing them as dependencies), as shown in Listing 3-14.

Listing 3-14. *Clean-all Target*

```

<!-- ===== -->
<!-- Target: clean-all -->
<!-- Removes all build artifacts -->
<!-- ===== -->
<target name="clean-all" description="Removes all build artifacts">
    <antcall target="compile-clean" />
    <antcall target="generate-docs-clean" />
    <antcall target="test-clean" />
    ...
</target>

```

The All Target

Finally, it's a common practice to make the buildfile default target a target named "all", which has in its dependencies a list of the targets that represented a full build of the system. If your build process has any non-critical targets that take a fair amount of time to generate, you can create new targets that will do whatever the all target does in addition to any extra work. For example a target that does "all" and also generates documentation can be called "all-with-docs". The point is that you want to minimize the amount of time that it takes to build the application so that developers don't have noticeable interruptions in the flow of their work. A typical all target looks like that shown in Listing 3-15.

Listing 3-15. *The All Target*

```

<!-- ===== -->
<!-- Does it all -->
<!-- ===== -->

<target
    name="all"
    depends="compile,..."
    description="Generates, compiles, packages and deploys."
/>

```

Eclipse Integration

In Chapter 2 we learned how to get the Eclipse IDE installed and configured. Now that we have an Ant buildfile it will be ideal if we can achieve harmony between the command line and the IDE. Luckily for us, Eclipse ships with powerful Ant integration. Eclipse provides a great Ant XML editor with syntax highlighting, code completion, flyover evaluation of Ant elements, as well as immediate visual feedback about the validity of your buildfile.

For the TechConf application to work seamlessly, you can create an Ant Builder, which is a facility provided by Eclipse's external tools framework for Ant integration. An Ant Builder can be configured to run at specific times. For example, in my environment I configured the Ant build to run when a manual build is invoked or when the project "Clean" option is selected.

To create an Ant builder, select Project ► Properties from the Eclipse menu (alternatively you can right-click and select Properties or press Alt+Enter on the project's top node in the Navigator or Package Explorer). Next, select the Builders node as shown in Figure 3-9.

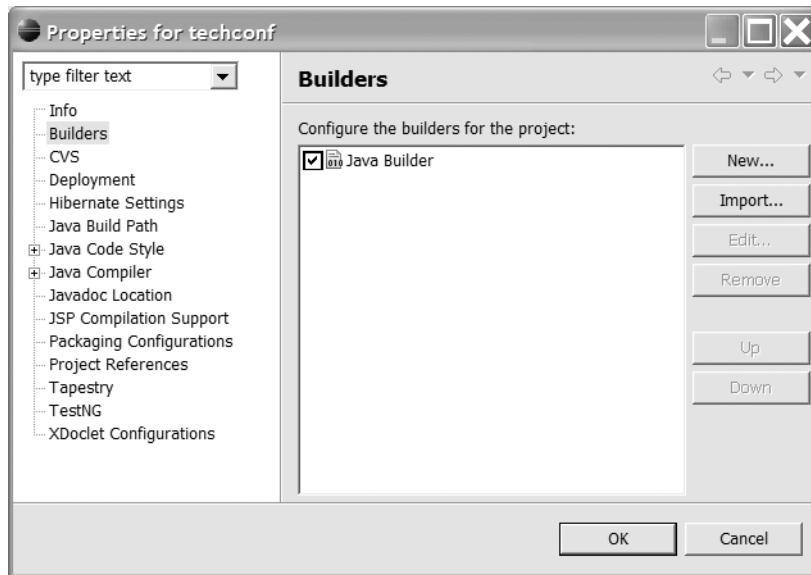


Figure 3-9. *TechConf Eclipse project properties dialog*

To create a new builder, click New in the Builders property dialog. Another dialog will appear asking to select the type of builder to create, as shown in Figure 3-10.

Select the Ant Build option and click OK. You should now be presented with the Ant builder property dialog as shown in Figure 3-11. Enter a suitable name for the builder in the Name field such as “TechConf Ant Builder”. The dialog consists of several tabs of options. In the Main tab you can select the Ant buildfile to be used by the builder. Under the Buildfile field click Browse Workspace and find the build.xml file at the root of the techconf project.

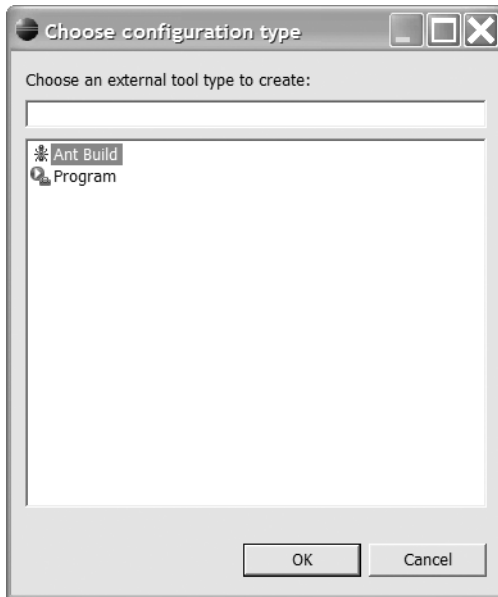


Figure 3-10. External tool builder type selection

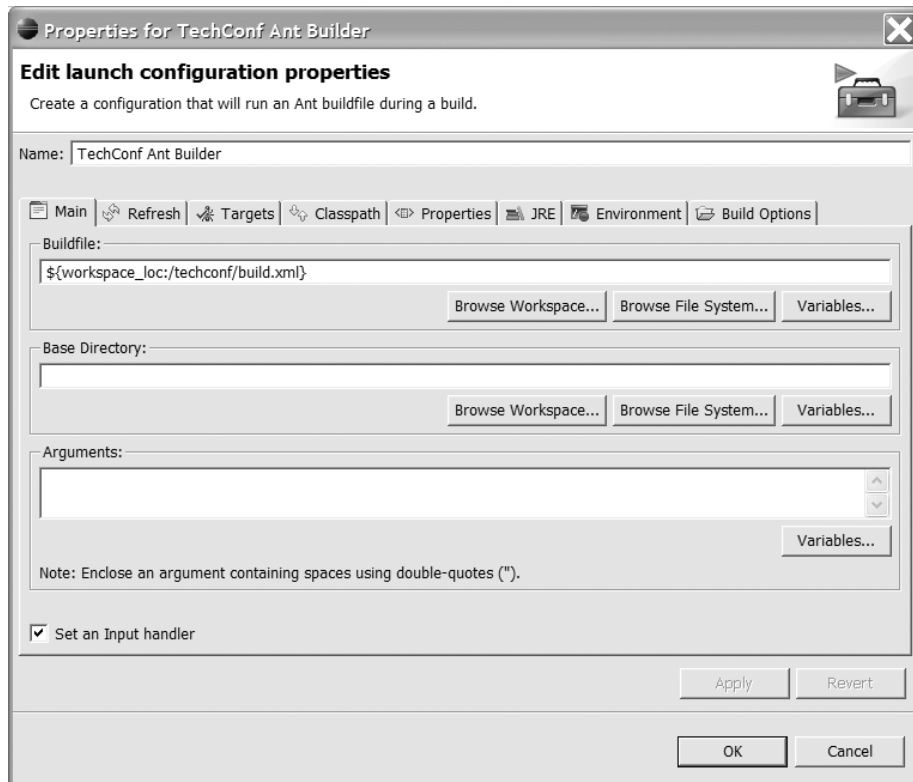


Figure 3-11. Eclipse Ant builder properties dialog

Under the Refresh tab, check the “Refresh resources upon completion” box, choose the radio button labeled “The project containing the selected resource”, and check “Recursively include sub-folders” as shown in Figure 3-12.

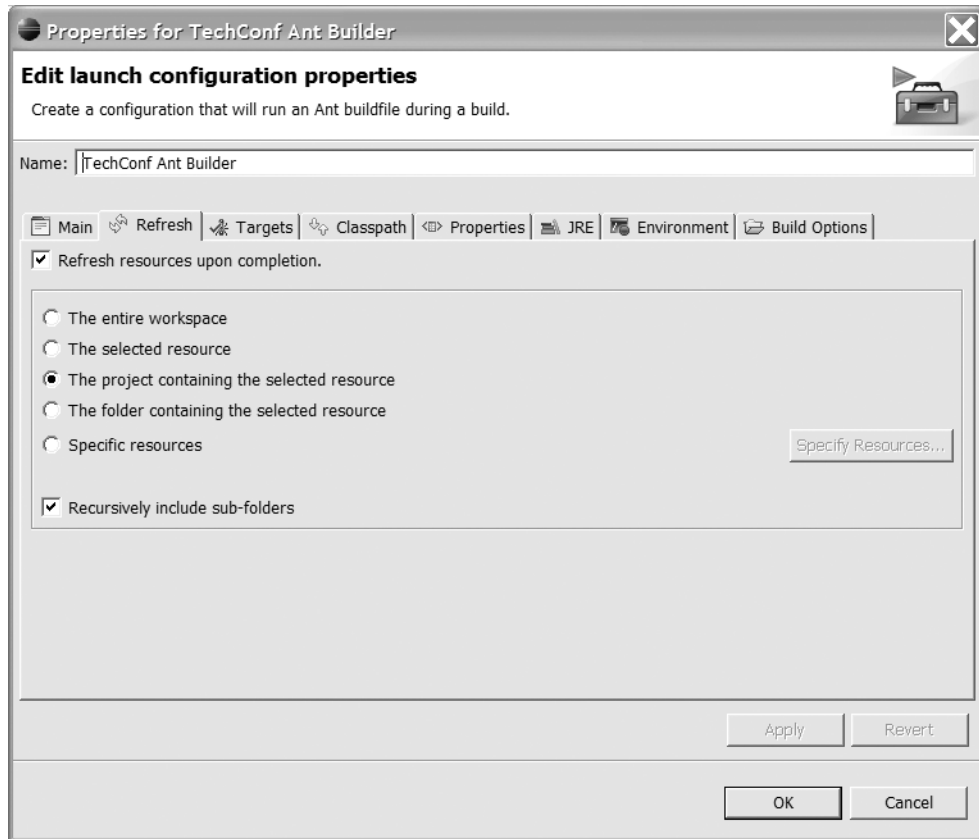


Figure 3-12. Eclipse Ant builder properties Refresh tab

Finally, under the Targets tab, you want the default target of the Ant build to be executed after a “Clean” and during a “Manual Build” as shown in Figure 3-13.

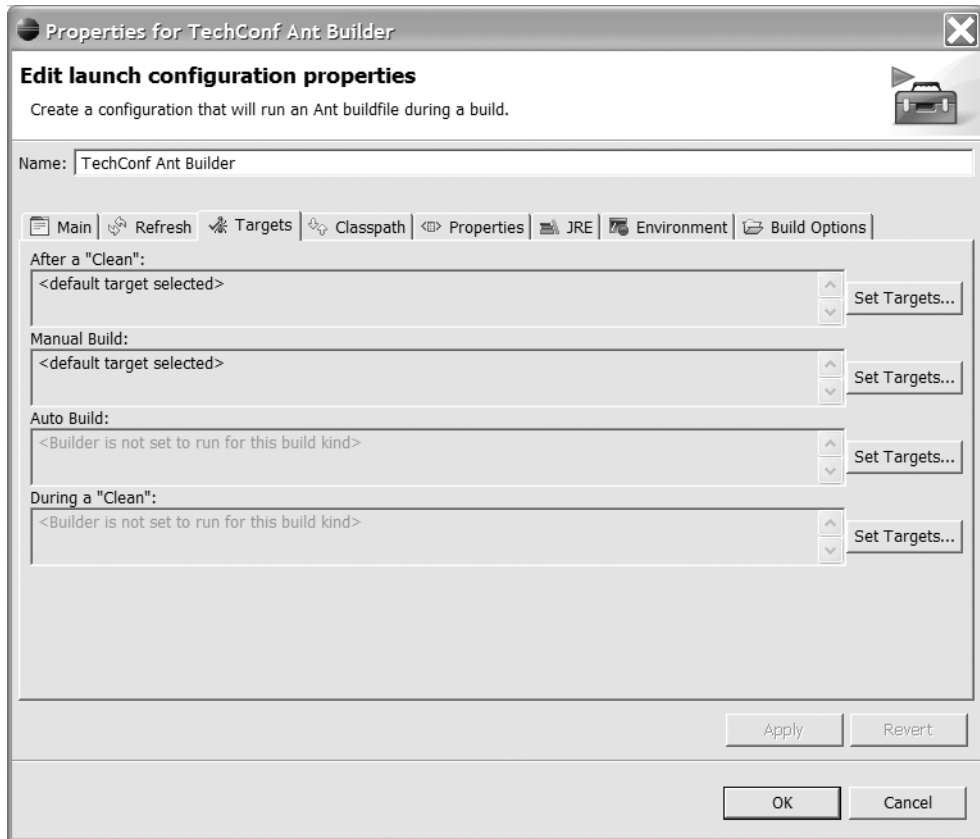


Figure 3-13. Eclipse Ant builder properties Targets tab

Summary

In this chapter you have learned the importance of having a solid build system in place and the basics of the Ant build tool. We crafted a reusable Ant build to automate the building process of the TechConf system by using several open source Ant tasks. The resulting base build reflects my experience building many Java and J2EE applications. You can apply most of the ideas used in this system to your existing and future projects. In the rest of the book we will continue to enhance the build to create the J2EE components and artifacts that compose the sample application.

The power of a well-crafted build will become even more apparent when we combine the building blocks learned in this chapter with the power of unit testing (Chapter 8) and continuous integration (Chapter 9).

