CHAPTER 9
Java

Many Java developers like Integrated Development Environments (IDEs) such as Eclipse. Given such well-known alternatives as Java IDEs and `Ant`, readers could well ask why they should even think of using `make` on Java projects. This chapter explores the value of `make` in these situations; in particular, it presents a generalized *makefile* that can be dropped into just about any Java project with minimal modification and carry out all the standard rebuilding tasks.

Using `make` with Java raises several issues and introduces some opportunities. This is primarily due to three factors: the Java compiler, `javac`, is extremely fast; the standard Java compiler supports the `@filename` syntax for reading "command-line parameters" from a file; and if a Java package is specified, the Java language specifies a path to the *.class* file.

Standard Java compilers are very fast. This is primarily due to the way the `import` directive works. Similar to a `#include` in C, this directive is used to allow access to externally defined symbols. However, rather than rereading source code, which then needs to be reparsed and analyzed, Java reads the class files directly. Because the symbols in a class file cannot change during the compilation process, the class files are cached by the compiler. In even medium-sized projects, this means the Java compiler can avoid rereading, parsing, and analyzing literally millions of lines of code compared with C. A more modest performance improvement is due to the bare minimum of optimization performed by most Java compilers. Instead, Java relies on sophisticated just-in-time (JIT) optimizations performed by the Java virtual machine (JVM) itself.

Most large Java projects make extensive use of Java's *package* feature. A class is declared to be encapsulated in a `package` that forms a scope around the symbols defined by the file. Package names are hierarchical and implicitly define a file structure. For instance, the package `a.b.c` would implicitly define a directory structure *a/b/c*. Code declared to be within the `a.b.c` package would be compiled to class files in the *a/b/c* directory. This means that `make`'s normal algorithm for associating a binary file with its source fails. But it also means that there is no need to specify a `-o` option to indicate where output files should be placed. Indicating the root of the output tree, which is the same for all files, is sufficient. This, in turn, means that source files from different directories can be compiled with the same command-line invocation.

The standard Java compilers all support the `@filename` syntax that allows command-line parameters to be read from a file. This is significant in conjunction with the package feature because it means that the entire Java source for a project can be compiled with a single execution of the Java compiler. This is a major performance improvement because the time it takes to load and execute the compiler is a major contributor to build times.

In summary, by composing the proper command line, compiling 400,000 lines of Java takes about three minutes on a 2.5-GHz Pentium 4 processor. Compiling an equivalent C++ application would require hours.

Alternatives to make

As previously mentioned, the Java developer community enthusiastically adopts new technologies. Let's see how two of these, `Ant` and IDEs, relate to `make`.

Ant

The Java community is very active, producing new tools and APIs at an impressive rate. One of these new tools is `Ant`, a build tool intended to replace `make` in the Java development process. Like `make`, `Ant` uses a description file to indicate the targets and prerequisites of a project. Unlike `make`, `Ant` is written in Java and `Ant` build files are written in XML.

To give you a feel for the XML build file, here is an excerpt from the `Ant` build file:

```
<target name="build"
        depends="prepare, check_for_optional_packages"
        description="--> compiles the source code">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes}"/>
  <mkdir dir="${build.lib}"/>

  <javac srcdir="${java.dir}"
         destdir="${build.classes}"
         debug="${debug}"
         deprecation="${deprecation}"
         target="${javac.target}"
         optimize="${optimize}" >
    <classpath refid="classpath"/>
  </javac>

  …

  <copy todir="${build.classes}">
    <fileset dir="${java.dir}">
```

```
        <include name="**/*.properties"/>
        <include name="**/*.dtd"/>
      </fileset>
    </copy>
  </target>
```

As you can see, a target is introduced with an XML `<target>` tag. Each target has a name and dependency list specified with `<name>` and `<depends>` attributes, respectively. Actions are performed by `Ant` *tasks*. A task is written in Java and bound to an XML tag. For instance, the task of creating a directory is specified with the `<mkdir>` tag and triggers the execution of the Java method `Mkdir.execute`, which eventually calls `File.mkdir`. As far as possible, all tasks are implemented using the Java API.

An equivalent build file using `make` syntax would be:

```
# compiles the source code
build: $(all_javas) prepare check_for_optional_packages
        $(MKDIR) -p $(build.dir) $(build.classes) $(build.lib)
        $(JAVAC) -sourcepath $(java.dir)                        \
                 -d $(build.classes)                            \
                 $(debug)                                       \
                 $(deprecation)                                 \
                 -target $(javac.target)                        \
                 $(optimize)                                    \
                 -classpath $(classpath)                        \
                 @$<
…
        $(FIND) . \( -name '*.properties' -o -name '*.dtd' \) | \
        $(TAR) -c -f - -T - | $(TAR) -C $(build.classes) -x -f -
```

This snippet of `make` uses techniques that this book hasn't discussed yet. Suffice to say that the prerequisite *all.javas* contains a list of all `java` files to be compiled. The `Ant` tasks `<mkdir>`, `<javac>`, and `<copy>` also perform dependency checking. That is, if the directory already exists, `mkdir` is not executed. Likewise, if the Java class files are newer than the source files, the source files are not compiled. Nevertheless, the `make` command script performs essentially the same functions. `Ant` includes a generic task, called `<exec>`, to run a local program.

`Ant` is a clever and fresh approach to build tools; however, it presents some issues worth considering:

- Although `Ant` has found wide acceptance in the Java community, it is still relatively unknown elsewhere. Also, it seems doubtful that its popularity will spread much beyond Java (for the reasons listed here). `make`, on the other hand, has consistently been applied to a broad range of fields including software development, document processing and typesetting, and web site and workstation maintenance, to name a few. Understanding `make` is important for anyone who needs to work on a variety of software systems.
- The choice of XML as the description language is appropriate for a Java-based tool. But XML is not particularly pleasant to write or to read (for many). Good

XML editors can be difficult to find and often do not integrate well with existing tools (either my integrated development environment includes a good XML editor or I must leave my IDE and find a separate tool). As you can see from the previous example, XML and the `Ant` dialect, in particular, are verbose compared with `make` and shell syntax. And the XML is filled with its own idiosyncrasies.

- When writing `Ant` build files you must contend with another layer of indirection. The `Ant` `<mkdir>` task does not invoke the underlying `mkdir` program for your system. Instead, it executes the Java `mkdir( )` method of the `java.io.File` class. This may or may not do what you expect. Essentially, any knowledge a programmer brings to `Ant` about the behavior of common tools is suspect and must be checked against the `Ant` documentation, Java documentation, or the `Ant` source. In addition, to invoke the Java compiler, for instance, I may be forced to navigate through a dozen or more unfamiliar XML attributes, such as `<srcdir>`, `<debug>`, etc., that are not documented in the compiler manual. In contrast, the `make` script is completely transparent, that is, I can typically type the commands directly into a shell to see how they behave.
- Although `Ant` is certainly portable, so is `make`. As shown in Chapter 7, writing portable *makefile*s, like writing portable `Ant` files, requires experience and knowledge. Programmers have been writing portable *makefile*s for two decades. Furthermore, the `Ant` documentation notes that there are portability issues with symbolic links on Unix and long filenames on Windows, that MacOS X is the only supported Apple operating system, and that support for other platforms is not guaranteed. Also, basic operations like setting the execution bit on a file cannot be performed from the Java API. An external program must be used. Portability is never easy or complete.
- The `Ant` tool does not explain precisely what it is doing. Since `Ant` tasks are not generally implemented by executing shell commands, the `Ant` tool has a difficult time displaying its actions. Typically, the display consists of natural language prose from `print` statements added by the task author. These print statements cannot be executed by a user from a shell. In contrast, the lines echoed by `make` are usually command lines that a user can copy and paste into a shell for reexecution. This means the `Ant` build is less useful to developers trying to understand the build process and tools. Also, it is not possible for a developer to reuse parts of a task, impromptu, at the keyboard.
- Last and most importantly, `Ant` shifts the build paradigm from a scripted to a nonscripted programming language. `Ant` tasks are written in Java. If a task does not exist or does not do what you want, you must either write your own task in Java or use the `<exec>` task. (Of course, if you use the `<exec>` task often, you would do far better to simply use `make` with its macros, functions, and more compact syntax.)

> Scripting languages, on the other hand, were invented and flourish precisely to address this type of issue. make has existed for nearly 30 years and can be used in the most complex situations without extending its implementation. Of course, there have been a handful of extensions in those 30 years. Many of them conceived and implemented in GNU make.

Ant is a marvelous tool that is widely accepted in the Java community. However, before embarking on a new project, consider carefully if Ant is appropriate for your development environment. This chapter will hopefully prove to you that make can powerfully meet your Java build needs.

IDEs

Many Java developers use Integrated Development Environments (IDEs) that bundle an editor, compiler, debugger, and code browser in a single (typically) graphical environment. Examples include the open source Eclipse (*http://www.eclipse.org*) and Emacs JDEE (*http://jdee.sunsite.dk*), and, from commercial vendors, Sun Java Studio (*http://wwws.sun.com/software/sundev/jde*) and JBuilder (*http://www.borland.com/jbuilder*). These environments typically have the notion of a project-build process that compiles the necessary files and enables the application execution.

If the IDEs support all this, why should we consider using make? The most obvious reason is portability. If there is ever a need to build the project on another platform, the build may fail when ported to the new target. Although Java itself is portable across platforms, the support tools are often not. For instance, if the configuration files for your project include Unix- or Windows-style paths, these may generate errors when the build is run on the other operating system. A second reason to use make is to support unattended builds. Some IDEs support batch building and some do not. The quality of support for this feature also varies. Finally, the build support included is often limited. If you hope to implement customized release directory structures, integrate help files from external applications, support automated testing, and handle branching and parallel lines of development, you may find the integrated build support inadequate.

In my own experience, I have found the IDEs to be fine for small scale or localized development, but production builds require the more comprehensive support that make can provide. I typically use an IDE to write and debug code, and write a *makefile* for production builds and releases. During development I use the IDE to compile the project to a state suitable for debugging. But if I change many files or modify files that are input to code generators, then I run the *makefile*. The IDEs I've used do not have good support for external source code generation tools. Usually the result of an IDE build is not suitable for release to internal or external customers. For that task I use make.

A Generic Java Makefile

Example 9-1 shows a generic *makefile* for Java; I'll explain each of its parts later in the chapter.

Example 9-1. Generic makefile for Java

```
# A generic makefile for a Java project.

VERSION_NUMBER := 1.0

# Location of trees.
SOURCE_DIR  := src
OUTPUT_DIR  := classes

# Unix tools
AWK         := awk
FIND        := /bin/find
MKDIR       := mkdir -p
RM          := rm -rf
SHELL       := /bin/bash

# Path to support tools
JAVA_HOME   := /opt/j2sdk1.4.2_03
AXIS_HOME   := /opt/axis-1_1
TOMCAT_HOME := /opt/jakarta-tomcat-5.0.18
XERCES_HOME := /opt/xerces-1_4_4
JUNIT_HOME  := /opt/junit3.8.1

# Java tools
JAVA        := $(JAVA_HOME)/bin/java
JAVAC       := $(JAVA_HOME)/bin/javac

JFLAGS      := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -source 1.4

JVMFLAGS    := -ea                              \
               -esa                             \
               -Xfuture

JVM         := $(JAVA) $(JVMFLAGS)

JAR         := $(JAVA_HOME)/bin/jar
JARFLAGS    := cf

JAVADOC     := $(JAVA_HOME)/bin/javadoc
JDFLAGS     := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -link http://java.sun.com/products/jdk/1.4/docs/api

# Jars
COMMONS_LOGGING_JAR   := $(AXIS_HOME)/lib/commons-logging.jar
```

Example 9-1. Generic makefile for Java (continued)

```
LOG4J_JAR            := $(AXIS_HOME)/lib/log4j-1.2.8.jar
XERCES_JAR           := $(XERCES_HOME)/xerces.jar
JUNIT_JAR            := $(JUNIT_HOME)/junit.jar

# Set the Java classpath
class_path := OUTPUT_DIR               \
              XERCES_JAR               \
              COMMONS_LOGGING_JAR      \
              LOG4J_JAR                \
              JUNIT_JAR

# space - A blank space
space := $(empty) $(empty)

# $(call build-classpath, variable-list)
define build-classpath
$(strip                                         \
  $(patsubst :%,%,                              \
    $(subst : ,:,                               \
      $(strip                                   \
        $(foreach j,$1,$(call get-file,$j):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                       \
    $($1)                                       \
    $(if $(call file-exists-eval,$1),,          \
      $(warning The file referenced by variable \
               '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                       \
    $(if $($1),,$(warning '$1' has no value))   \
    $(wildcard $($1)))

# $(call brief-help, makefile)
define brief-help
  $(AWK) '$$1 ~ /^[^.][-A-Za-z0-9]*:/                   \
         { print substr($$1, 1, length($$1)-1) }' $1 |  \
  sort |                                                \
  pr -T -w 80 -4
endef

# $(call file-exists, wildcard-pattern)
file-exists = $(wildcard $1)

# $(call check-file, file-list)
define check-file
  $(foreach f, $1,                              \
```

Example 9-1. Generic makefile for Java (continued)

```
    $(if $(call file-exists, $($f)),,           \
      $(warning $f ($($f)) is missing)))
endef

# #(call make-temp-dir, root-opt)
define make-temp-dir
  mktemp -t $(if $1,$1,make).XXXXXXXXXX
endef

# MANIFEST_TEMPLATE - Manifest input to m4 macro processor
MANIFEST_TEMPLATE := src/manifest/manifest.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                       \
     --define=IMPL_VERSION=$(VERSION_NUMBER)            \
     --define=SPEC_VERSION=$(VERSION_NUMBER)            \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                   \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef

# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef

# Set the CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))

# make-directories - Ensure output directory exists.
make-directories := $(shell $(MKDIR) $(OUTPUT_DIR))

# help - The default goal
.PHONY: help
help:
        @$(call brief-help, $(CURDIR)/Makefile)

# all - Perform all tasks for a complete build
.PHONY: all
all: compile jars javadoc
```

Example 9-1. Generic makefile for Java (continued)

```
# all_javas - Temp file for holding source file list
all_javas := $(OUTPUT_DIR)/all.javas

# compile - Compile the source
.PHONY: compile
compile: $(all_javas)
	$(JAVAC) $(JFLAGS) @$<

# all_javas - Gather source file list
.INTERMEDIATE: $(all_javas)
$(all_javas):
	$(FIND) $(SOURCE_DIR) -name '*.java' > $@

# jar_list - List of all jars to create
jar_list := server_jar ui_jar

# jars - Create all jars
.PHONY: jars
jars: $(jar_list)

# server_jar - Create the $(server_jar)
server_jar_name     := $(OUTPUT_DIR)/lib/a.jar
server_jar_manifest := src/com/company/manifest/foo.mf
server_jar_packages := com/company/m com/company/n

# ui_jar - create the $(ui_jar)
ui_jar_name     := $(OUTPUT_DIR)/lib/b.jar
ui_jar_manifest := src/com/company/manifest/bar.mf
ui_jar_packages := com/company/o com/company/p

# Create an explicit rule for each jar
# $(foreach j, $(jar_list), $(eval $(call make-jar,$j)))
$(eval $(call make-jar,server_jar))
$(eval $(call make-jar,ui_jar))

# javadoc - Generate the Java doc from sources
.PHONY: javadoc
javadoc: $(all_javas)
	$(JAVADOC) $(JDFLAGS) @$<

.PHONY: clean
clean:
	$(RM) $(OUTPUT_DIR)

.PHONY: classpath
classpath:
	@echo CLASSPATH='$(CLASSPATH)'

.PHONY: check-config
check-config:
	@echo Checking configuration...
	$(call check-file, $(class_path) JAVA_HOME)
```

Example 9-1. Generic makefile for Java (continued)

```
.PHONY: print
print:
	$(foreach v, $(V), \
	  $(warning $v = $($v)))
```

Compiling Java

Java can be compiled with make in two ways: the traditional approach, one javac execution per source file; or the fast approach outlined previously using the @filename syntax.

The Fast Approach: All-in-One Compile

Let's start with the fast approach. As you can see in the generic *makefile*:

```
# all_javas - Temp file for holding source file list
all_javas := $(OUTPUT_DIR)/all.javas

# compile - Compile the source
.PHONY: compile
compile: $(all_javas)
	$(JAVAC) $(JFLAGS) @$<

# all_javas - Gather source file list
.INTERMEDIATE: $(all_javas)
$(all_javas):
	$(FIND) $(SOURCE_DIR) -name '*.java' > $@
```

The phony target compile invokes javac once to compile all the source of the project.

The $(all_javas) prerequisite is a file, *all.javas*, containing a list of Java files, one filename per line. It is not necessary for each file to be on its own line, but this way it is much easier to filter files with grep -v if the need ever arises. The rule to create *all.javas* is marked .INTERMEDIATE so that make will remove the file after each run and thus create a new one before each compile. The command script to create the file is straightforward. For maximum maintainability we use the find command to retrieve all the java files in the source tree. This command can be a bit slow, but is guaranteed to work correctly with virtually no modification as the source tree changes.

If you have a list of source directories readily available in the *makefile*, you can use faster command scripts to build *all.javas*. If the list of source directories is of medium length so that the length of the command line does not exceed the operating system's limits, this simple script will do:

```
$(all_javas):
	shopt -s nullglob; \
	printf "%s\n" $(addsuffix /*.java,$(PACKAGE_DIRS)) > $@
```

This script uses shell wildcards to determine the list of Java files in each directory. If, however, a directory contains no Java files, we want the wildcard to yield the empty string, not the original globbing pattern (the default behavior of many shells). To achieve this effect, we use the `bash` option `shopt -s nullglob`. Most other shells have similar options. Finally, we use globbing and `printf` rather than `ls -1` because these are built-in to `bash`, so our command script executes only a single program regardless of the number of package directories.

Alternately, we can avoid shell globbing by using `wildcard`:

```
$(all_javas):
	print "%s\n" $(wildcard \
	               $(addsuffix /*.java,$(PACKAGE_DIRS))) > $@
```

If you have very many source directories (or very long paths), the above script may exceed the command-line length limit of the operating system. In that case, the following script may be preferable:

```
.INTERMEDIATE: $(all_javas)
$(all_javas):
	shopt -s nullglob;          \
	for f in $(PACKAGE_DIRS);   \
	do                          \
	  printf "%s\n" $$f/*.java; \
	done > $@
```

Notice that the `compile` target and the supporting rule follow the nonrecursive `make` approach. No matter how many subdirectories there are, we still have one *makefile* and one execution of the compiler. If you want to compile all of the source, this is as fast as it gets.

Also, we completely discarded all dependency information. With these rules, `make` neither knows nor cares about which file is newer than which. It simply compiles everything on every invocation. As an added benefit, we can execute the *makefile* from the source tree, instead of the binary tree. This may seem like a silly way to organize the *makefile* considering `make`'s abilities to manage dependencies, but consider this:

- The alternative (which we will explore shortly) uses the standard dependency approach. This invokes a new `javac` process for each file, adding a lot of overhead. But, if the project is small, compiling all the source files will not take significantly longer than compiling a few files because the `javac` compiler is so fast and process creation is typically slow. Any build that takes less than 15 seconds is basically equivalent regardless of how much work it does. For instance, compiling approximately 500 source files (from the Ant distribution) takes 14 seconds on my 1.8-GHz Pentium 4 with 512 MB of RAM. Compiling one file takes five seconds.
- Most developers will be using some kind of development environment that provides fast compilation for individual files. The *makefile* will most likely be used

when changes are more extensive, complete rebuilds are required, or unattended builds are necessary.

- As we shall see, the effort involved in implementing and maintaining dependencies is equal to the separate source and binary tree builds for C/C++ (described in Chapter 8). Not a task to be underestimated.

As we will see in later examples, the PACKAGE_DIRS variable has uses other than simply building the *all.javas* file. But maintaining this variables can be a labor-intensive, and potentially difficult, step. For smaller projects, the list of directories can be maintained by hand in the *makefile*, but when the number grows beyond a hundred directories, hand editing becomes error-prone and irksome. At this point, it might be prudent to use find to scan for these directories:

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                          \
  $(patsubst %/,%,                               \
    $(sort                                       \
      $(dir                                      \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

The find command returns a list of files, dir discards the file leaving only the directory, sort removes duplicates from the list, and patsubst strips the trailing slash. Notice that find-compilation-dirs finds the list of files to compile, only to discard the filenames, then the *all.javas* rule uses wildcards to restore the filenames. This seems wasteful, but I have often found that a list of the packages containing source code is very useful in other parts of the build, for instance to scan for EJB configuration files. If your situation does not require a list of packages, then by all means use one of the simpler methods previously mentioned to build *all.javas*.

Compiling with Dependencies

To compile with full dependency checking, you first need a tool to extract dependency information from the Java source files, something similar to cc -M. Jikes (*http://www.ibm.com/developerworks/opensource/jikes*) is an open source Java compiler that supports this feature with the -makefile or +M option. Jikes is not ideal for separate source and binary compilation because it always writes the dependency file in the same directory as the source file, but it is freely available and it works. On the plus side, it generates the dependency file while compiling, avoiding a separate pass.

Here is a dependency processing function and a rule to use it:

```
%.class: %.java
        $(JAVAC) $(JFLAGS) +M $<
        $(call java-process-depend,$<,$@)

# $(call java-process-depend, source-file, object-file)
define java-process-depend
```

```
    $(SED) -e 's/^.*\.class *:/$2 $(subst .class,.d,$2):/'   \
           $(subst .java,.u,$1) > $(subst .class,.tmp,$2)
    $(SED) -e 's/#.*//'                                      \
           -e 's/^[^:]*: *//'                                \
           -e 's/ *\\$$$$//'                                 \
           -e '/^$$$$/ d'                                    \
           -e 's/$$$$/ :/' $(subst .class,.tmp,$2)           \
           >>  $(subst .class,.tmp,$2)
    $(MV) $(subst .class,.tmp,$2).tmp  $(subst .class,.d,$2)
  endef
```

This requires that the *makefile* be executed from the binary tree and that the `vpath` be set to find the source. If you want to use the Jikes compiler only for dependency generation, resorting to a different compiler for actual code generation, you can use the `+B` option to prevent Jikes from generating bytecodes.

In a simple timing test compiling 223 Java files, the single line compile described previously as the fast approach required 9.9 seconds on my machine. The same 223 files compiled with individual compilation lines required 411.6 seconds or 41.5 times longer. Furthermore, with separate compilation, any build that required compiling more than four files was slower than compiling all the source files with a single compile line. If the dependency generation and compilation were performed by separate programs, the discrepancy would increase.

Of course, development environments vary, but it is important to carefully consider your goals. Minimizing the number of files compiled will not always minimize the time it takes to build a system. For Java in particular, full dependency checking and minimizing the number of files compiled does not appear to be necessary for normal program development.

Setting CLASSPATH

One of the most important issues when developing software with Java is setting the `CLASSPATH` variable correctly. This variable determines which code is loaded when a class reference is resolved. To compile a Java application correctly, the *makefile* must include the proper `CLASSPATH`. The `CLASSPATH` can quickly become long and complex as Java packages, APIs, and support tools are added to a system. If the `CLASSPATH` can be difficult to set properly, it makes sense to set it in one place.

A technique I've found useful is to use the *makefile* to set the `CLASSPATH` for itself and other programs. For instance, a target `classpath` can return the `CLASSPATH` to the shell invoking the *makefile*:

```
.PHONY: classpath
classpath:
        @echo "export CLASSPATH='$(CLASSPATH)'"
```

Developers can set their `CLASSPATH` with this (if they use `bash`):

```
$ eval $(make classpath)
```

The CLASSPATH in the Windows environment can be set with this invocation:

```
.PHONY: windows_classpath
windows_classpath:
        regtool set /user/Environment/CLASSPATH "$(subst /,\\,$(CLASSPATH))"
        control sysdm.cpl,@1,3 &
        @echo "Now click Environment Variables, then OK, then OK again."
```

The program regtool is a utility in the Cygwin development system that manipulates the Windows Registry. Simply setting the Registry doesn't cause the new values to be read by Windows, however. One way to do this is to visit the Environment Variable dialog box and simply exit by clicking OK.

The second line of the command script causes Windows to display the System Properties dialog box with the Advanced tab active. Unfortunately, the command cannot display the Environment Variables dialog box or activate the OK button, so the last line prompts the user to complete the task.

Exporting the CLASSPATH to other programs, such as Emacs JDEE or JBuilder project files, is not difficult.

Setting the CLASSPATH itself can also be managed by make. It is certainly reasonable to set the CLASSPATH variable in the obvious way with:

```
CLASSPATH = /third_party/toplink-2.5/TopLink.jar:/third_party/…
```

For maintainability, using variables is preferred:

```
CLASSPATH = $(TOPLINK_25_JAR):$(TOPLINKX_25_JAR):…
```

But we can do better than this. As you can see in the generic *makefile*, we can build the CLASSPATH in two stages: first list the elements in the path as make variables, then transform those variables into the string value of the environment variable:

```
# Set the Java classpath
class_path := OUTPUT_DIR                \
              XERCES_JAR                \
              COMMONS_LOGGING_JAR       \
              LOG4J_JAR                 \
              JUNIT_JAR
…
# Set the CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))
```

(The CLASSPATH in Example 9-1 is meant to be more illustrative than useful.) A well-written build-classpath function solves several irritating problems:

- It is very easy to compose a CLASSPATH in pieces. For instance, if different applications servers are used, the CLASSPATH might need to change. The different versions of the CLASSPATH could then be enclosed in ifdef sections and selected by setting a make variable.
- Casual maintainers of the *makefile* do not have to worry about embedded blanks, newlines, or line continuation, because the build-classpath function handles them.

- The path separator can be selected automatically by the build-classpath function. Thus, it is correct whether run on Unix or Windows.
- The validity of path elements can be verified by the build-classpath function. In particular, one irritating problem with make is that undefined variables collapse to the empty string without an error. In most cases this is very useful, but occasionally it gets in the way. In this case, it quietly yields a bogus value for the CLASSPATH variable.* We can solve this problem by having the build-classpath function check for the empty valued elements and warn us. The function can also check that each file or directory exists.
- Finally, having a hook to process the CLASSPATH can be useful for more advanced features, such as help accommodating embedded spaces in path names and search paths.

Here is an implementation of build-classpath that handles the first three issues:

```
# $(call build-classpath, variable-list)
define build-classpath
$(strip                                  \
  $(patsubst %:,%,                       \
    $(subst : ,:,                        \
      $(strip                            \
        $(foreach c,$1,$(call get-file,$c):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                               \
    $($1)                               \
    $(if $(call file-exists-eval,$1),,  \
      $(warning The file referenced by variable \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                \
    $(if $($1),,$(warning '$1' has no value))    \
    $(wildcard $($1)))
endef
```

The build-classpath function iterates through the words in its argument, verifying each element and concatenating them with the path separator (: in this case). Selecting the path separator automatically is easy now. The function then strips spaces added by the get-file function and foreach loop. Next, it strips the final separator

* We could try using the --warn-undefined-variables option to identify this situation, but this also flags many other empty variables that are desirable.

added by the foreach loop. Finally, the whole thing is wrapped in a strip so errant spaces introduced by line continuation are removed.

The get-file function returns its filename argument, then tests whether the variable refers to an existing file. If it does not, it generates a warning. It returns the value of the variable regardless of the existence of the file because the value may be useful to the caller. On occasion, get-file may be used with a file that will be generated, but does not yet exist.

The last function, file-exists-eval, accepts a variable name containing a file reference. If the variable is empty, a warning is issued; otherwise, the wildcard function is used to resolve the value into a file (or a list of files for that matter).

When the build-classpath function is used with some suitable bogus values, we see these errors:

```
Makefile:37: The file referenced by variable 'TOPLINKX_25_JAR'
             (/usr/java/toplink-2.5/TopLinkX.jar) cannot be found
...
Makefile:37: 'XERCES_142_JAR' has no value
Makefile:37: The file referenced by variable
             'XERCES_142_JAR' ( ) cannot be found
```

This represents a great improvement over the silence we would get from the simple approach.

The existence of the get-file function suggests that we could generalize the search for input files.

```
# $(call get-jar, variable-name)
define get-jar
  $(strip                                                   \
    $(if $($1),,$(warning '$1' is empty))                   \
    $(if $(JAR_PATH),,$(warning JAR_PATH is empty))         \
    $(foreach d, $(dir $($1)) $(JAR_PATH),                  \
      $(if $(wildcard $d/$(notdir $($1))),                  \
        $(if $(get-jar-return),,                            \
          $(eval get-jar-return := $d/$(notdir $($1))))))    \
    $(if $(get-jar-return),                                 \
      $(get-jar-return)                                     \
      $(eval get-jar-return :=),                            \
      $($1)                                                 \
      $(warning get-jar: File not found '$1' in $(JAR_PATH))))
endef
```

Here we define the variable JAR_PATH to contain a search path for files. The first file found is returned. The parameter to the function is a variable name containing the path to a jar. We want to look for the jar file first in the path given by the variable, then in the JAR_PATH. To accomplish this, the directory list in the foreach loop is composed of the directory from the variable, followed by the JAR_PATH. The two other uses of the parameter are enclosed in notdir calls so the jar name can be composed from a path from this list. Notice that we cannot exit from a foreach loop.

Instead, therefore, we use eval to set a variable, get-jar-return, to remember the first file we found. After the loop, we return the value of our temporary variable or issue a warning if nothing was found. We must remember to reset our return value variable before terminating the macro.

This is essentially reimplementing the vpath feature in the context of setting the CLASSPATH. To understand this, recall that the vpath is a search path used implicitly by make to find prerequisites that cannot be found from the current directory by a relative path. In these cases, make searches the vpath for the prerequisite file and inserts the completed path into the $^, $?, and $+ automatic variables. To set the CLASSPATH, we want make to search a path for each jar file and insert the completed path into the CLASSPATH variable. Since make has no built-in support for this, we've added our own. You could, of course, simply expand the jar path variable with the appropriate jar filenames and let Java do the searching, but CLASSPATHs already get long quickly. On some operating systems, environment variable space is limited and long CLASSPATHs are in danger of being truncated. On Windows XP, there is a limit of 1023 characters for a single environment variable. In addition, even if the CLASSPATH is not truncated, the Java virtual machine must search the CLASSPATH when loading classes, thus slowing down the application.

Managing Jars

Building and managing jars in Java presents different issues from C/C++ libraries. There are three reasons for this. First, the members of a jar include a relative path, so the precise filenames passed to the jar program must be carefully controlled. Second, in Java there is a tendency to merge jars so that a single jar can be released to represent a program. Finally, jars include other files than classes, such as manifests, property files, and XML.

The basic command to create a jar in GNU make is:

```
JAR      := jar
JARFLAGS := -cf

$(FOO_JAR): prerequisites…
	$(JAR) $(JARFLAGS) $@ $^
```

The jar program can accept directories instead of filenames, in which case, all the files in the directory trees are included in the jar. This can be very convenient, especially when used with the -C option for changing directories:

```
JAR      := jar
JARFLAGS := -cf

.PHONY: $(FOO_JAR)
$(FOO_JAR):
	$(JAR) $(JARFLAGS) $@ -C $(OUTPUT_DIR) com
```

Here the jar itself is declared `.PHONY`. Otherwise subsequent runs of the *makefile* would not recreate the file, because it has no prerequisites. As with the `ar` command described in an earlier chapter, there seems little point in using the update flag, `-u`, since it takes the same amount of time or longer as recreating the jar from scratch, at least for most updates.

A jar often includes a manifest that identifies the vendor, API and version number the jar implements. A simple manifest might look like:

```
Name: JAR_NAME
Specification-Title: SPEC_NAME
Implementation-Version: IMPL_VERSION
Specification-Vendor: Generic Innovative Company, Inc.
```

This manifest includes three placeholders, `JAR_NAME`, `SPEC_NAME`, and `IMPL_VERSION`, that can be replaced at jar creation time by `make` using `sed`, `m4`, or your favorite stream editor. Here is a function to process a manifest:

```
MANIFEST_TEMPLATE := src/manifests/default.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                      \
     --define=IMPL_VERSION=$(VERSION_NUMBER)           \
     --define=SPEC_VERSION=$(VERSION_NUMBER)           \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                  \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef
```

The `add-manifest` function operates on a manifest file similar to the one shown previously. The function first creates a temporary directory, then expands the sample manifest. Next, it updates the jar, and finally deletes the temporary directory. Notice that the last parameter to the function is optional. If the manifest file path is empty, the function uses the value from `MANIFEST_TEMPLATE`.

The generic *makefile* bundles these operations into a generic function to write an explicit rule for creating a jar:

```
# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef
```

It accepts a single argument, the prefix of a make variable, that identifies a set of variables describing four jar parameters: the target name, the jar name, the packages in the jar, and the jar's manifest file. For example, for a jar named *ui.jar*, we would write:

```
ui_jar_name     := $(OUTPUT_DIR)/lib/ui.jar
ui_jar_manifest := src/com/company/ui/manifest.mf
ui_jar_packages := src/com/company/ui \
                   src/com/company/lib

$(eval $(call make-jar,ui_jar))
```

By using variable name composition, we can shorten the calling sequence of our function and allow for a very flexible implementation of the function.

If we have many jar files to create, we can automate this further by placing the jar names in a variable:

```
jar_list := server_jar ui_jar

.PHONY: jars $(jar_list)
jars: $(jar_list)

$(foreach j, $(jar_list),\
  $(eval $(call make-jar,$j)))
```

Occasionally, we need to expand a jar file into a temporary directory. Here is a simple function to do that:

```
# $(call burst-jar, jar-file, target-directory)
define burst-jar
  $(call make-dir,$2)
  cd $2; $(JAR) -xf $1
endef
```

Reference Trees and Third-Party Jars

To use a single, shared reference tree to support partial source trees for developers, simply have the nightly build create jars for the project and include those jars in the CLASSPATH of the Java compiler. The developer can check out the parts of the source tree he needs and run the compile (assuming the source file list is dynamically created by something like find). When the Java compiler requires symbols from a missing source file, it will search the CLASSPATH and discover the *.class* file in the jar.

Selecting third-party jars from a reference tree is also simple. Just place the path to the jar in the CLASSPATH. The *makefile* can be a valuable tool for managing this process as previously noted. Of course, the get-file function can be used to automatically select beta or stable, local or remote jars by simply setting the JAR_PATH variable.

Enterprise JavaBeans

Enterprise JavaBeans™ is a powerful technique to encapsulate and reuse business logic in the framework of remote method invocation. EJB sets up Java classes used to implement server APIs that are ultimately used by remote clients. These objects and services are configured using XML-based control files. Once the Java classes and XML control files are written, they must be bundled together in a jar. Then a special EJB compiler builds stubs and ties to implement the RPC support code.

The following code can be plugged into Example 9-1 to provide generic EJB support:

```
EJB_TMP_JAR = $(EJB_TMP_DIR)/temp.jar
META_INF    = $(EJB_TMP_DIR)/META-INF

# $(call compile-bean, jar-name,
#                       bean-files-wildcard, manifest-name-opt)
define compile-bean
  $(eval EJB_TMP_DIR := $(shell mktemp -d $(TMPDIR)/compile-bean.XXXXXXXX))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml, $2),cp $(filter %.xml, $2) $(META_INF))
  cd $(OUTPUT_DIR) &&                          \
  $(JAR) -cf0 $(EJB_TMP_JAR)                   \
         $(call jar-file-arg,$(META_INF))      \
         $(filter-out %.xml, $2)
  $(JVM) weblogic.ejbc $(EJB_TMP_JAR) $1
  $(call add-manifest,$(if $3,$3,$1),,)
  $(RM) $(EJB_TMP_DIR)
endef

# $(call jar-file-arg, jar-file)
jar-file-arg = -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
```

The `compile-bean` function comaccepts three parameters: the name of the jar to create, the list of files in the jar, and an optional manifest file. The function first creates a clean temporary directory using the `mktemp` program and saves the directory name in the variable `EJB_TMP_DIR`. By embedding the assignment in an `eval`, we ensure that `EJB_TMP_DIR` is reset to a new temporary directory once for each expansion of `compile-bean`. Since `compile-bean` is used in the command script part of a rule, the function is expanded only when the command script is executed. Next, it copies any XML files in the bean file list into the *META-INF* directory. This is where EJB configuration files live. Then, the function builds a temporary jar that is used as input to the EJB compiler. The `jar-file-arg` function converts filenames of the form *dir1/dir2/dir3* into `-C dir1/dir2 dir3` so the relative path to the file in the jar is correct. This is the appropriate format for indicating the *META-INF* directory to the jar command. The bean file list contains *.xml* files that have already been placed in the *META-INF* directory, so we filter these files out. After building the temporary jar, the WebLogic EJB compiler is invoked, generating the output jar. A manifest is then added to the compiled jar. Finally, our temporary directory is removed.

Using the new function is straightforward:

```
bean_files = com/company/bean/FooInterface.class       \
             com/company/bean/FooHome.class            \
             src/com/company/bean/ejb-jar.xml          \
             src/com/company/bean/weblogic-ejb-jar.xml

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

The `bean_files` list is a little confusing. The *.class* files it references will be accessed relative to the *classes* directory, while the *.xml* files will be accessed relative to the directory of the *makefile*.

This is fine, but what if you have lots of bean files in your bean jar. Can we build the file list automatically? Certainly:

```
src_dirs := $(SOURCE_DIR)/com/company/...

bean_files =                                          \
  $(patsubst $(SOURCE_DIR)/%,%,                       \
    $(addsuffix /*.class,                             \
      $(sort                                          \
        $(dir                                         \
          $(wildcard                                  \
            $(addsuffix /*Home.java,$(src_dirs)))))))

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

This assumes that all the directories with EJB source are contained in the `src_dirs` variable (there can also be directories that do not contain EJB source) and that any file ending in *Home.java* identifies a package containing EJB code. The expression for setting the `bean_files` variable first adds the wildcard suffix to the directories, then invokes `wildcard` to gather the list of *Home.java* files. The filenames are discarded to leave the directories, which are sorted to remove duplicates. The wildcard `/*.class` suffix is added so that the shell will expand the list to the actual class files. Finally, the source directory prefix (which is not valid in the *classes* tree) is removed. Shell wildcard expansion is used instead of `make`'s `wildcard` because we can't rely on `make` to perform its expansion after the class files have been compiled. If `make` evaluated the `wildcard` function too early it would find no files and directory caching would prevent it from ever looking again. The `wildcard` in the source tree is perfectly safe because (we assume) no source files will be added while `make` is running.

The above code works when we have a small number of bean jars. Another style of development places each EJB in its own jar. Large projects may have dozens of jars. To handle this case automatically, we need to generate an explicit rule for each EJB jar. In this example, EJB source code is self-contained: each EJB is located in a single directory with its associated XML files. EJB directories can be identified by files that end with *Session.java*.

The basic approach is to search the source tree for EJBs, then build an explicit rule to create each EJB and write these rules into a file. The EJB rules file is then included in our *makefile*. The creation of the EJB rules file is triggered by make's own dependency handling of include files.

```
# session_jars - The EJB jars with their relative source path.
session_jars =
  $(subst .java,.jar,                          \
    $(wildcard                                 \
      $(addsuffix /*Session.java, $(COMPILATION_DIRS))))

# EJBS - A list of all EJB jars we need to build.
EJBS = $(addprefix $(TMP_DIR)/,$(notdir $(session_jars)))

# ejbs - Create all EJB jar files.
.PHONY: ejbs
ejbs: $(EJBS)
$(EJBS):
        $(call compile-bean,$@,$^,)
```

We find the *Session.java* files by calling a `wildcard` on all the compilation directories. In this example, the jar file is the name of the Session file with the *.jar* suffix. The jars themselves will be placed in a temporary binary directory. The `EJBS` variable contains the list of jars with their binary directory path. These EJB jars are the targets we want to update. The actual command script is our `compile-bean` function. The tricky part is that the file list is recorded in the prerequisites for each jar file. Let's see how they are created.

```
-include $(OUTPUT_DIR)/ejb.d

# $(call ejb-rule, ejb-name)
ejb-rule = $(TMP_DIR)/$(notdir $1):          \
            $(addprefix $(OUTPUT_DIR)/,      \
              $(subst .java,.class,          \
                $(wildcard $(dir $1)*.java))) \
            $(wildcard $(dir $1)*.xml)

# ejb.d - EJB dependencies file.
$(OUTPUT_DIR)/ejb.d: Makefile
        @echo Computing ejb dependencies...
        @for f in $(session_jars);           \
        do                                   \
          echo "\$$(call ejb-rule,$$f)";     \
        done > $@
```

The dependencies for each EJB jar are recorded in a separate file, *ejb.d*, that is included by the *makefile*. The first time make looks for this include file it does not exist. So make invokes the rule for updating the include file. This rule writes one line for each EJB, something like:

```
$(call ejb-rule,src/com/company/foo/FooSession.jar)
```

The function ejb-rule will expand to the target jar and its list of prerequisites, something like:

```
classes/lib/FooSession.jar: classes/com/company/foo/FooHome.jar \
            classes/com/company/foo/FooInterface.jar           \
            classes/com/company/foo/FooSession.jar             \
            src/com/company/foo/ejb-jar.xml                    \
            src/com/company/foo/ejb-weblogic-jar.xml
```

In this way, a large number of jars can be managed in make without incurring the overhead of maintaining a set of explicit rules by hand.