UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2019

Basic Compilation: javac and make

[The discussion in this section applies to Java 1.5 tools from Sun Microsystems. Tools from other manufacturers and earlier tools from Sun differ in various details.]

Programming languages do not exist in a vacuum; any actual programming done in any language one does within a *programming environment* that comprises the various programs, libraries, editors, debuggers, and other tools needed to actually convert program text into action. This document discusses the tools that translate programs into executable form and then execute them.

1 Compilation and Interpretation

The Scheme environment that you used in CS61A was particularly simple. It provided a component called the *reader,* which read in Scheme-program text from files or command lines and converted it into internal Scheme data structures. Then a component called the *interpreter* operated on these translated programs or statements, performing the actions they denoted. You probably weren't much aware of the reader; it doesn't amount to much because of Scheme's very simple syntax.

Java's more complex syntax and its static type structure (as discussed in lecture) require that you be a bit more aware of the reader—or *compiler,* as it is called in the context of Java and most other "production" programming languages. The Java compiler supplied by Sun Microsystems is a program called `javac` on our systems. You first prepare programs in files (called *source files*) using any appropriate text editor (Emacs, for example), giving them names that end in '`.java`'. Next you compile them with the java compiler to create new, translated files, called *class files*, one for each class, with names ending in '`.class`'. Once programs are translated into class files, there is a variety of tools for actually executing them, including Sun's java interpreter (called '`java`' on our systems), and interpreters built into products such as Netscape or Internet Explorer. The same class file format works (or is *supposed* to) on all of these.

In the simplest case, if the class containing your main program or applet is called C, then you should store it in a file called C`.java`, and you can compile it with the command

```
javac C.java
```

This will produce `.class` files for C and for any other classes that had to be compiled because they were mentioned (directly or indirectly) in class C. For homework problems, this is often all you need to know, and you can stop reading. However, things rapidly get complicated when a program consists of multiple classes, especially when they occur in multiple packages. In this document, we'll try to deal with the more straightforward of these complications.

2 Where 'java' and 'javac' find classes

Every Java class resides in a *package* (a collection of classes and subpackages). For example, the standard class `String` is actually `java.lang.String`: the class named `String` that resides in the subpackage named `lang` that resides in the outer-level package named `java`. You use a `package` declaration at the beginning of a `.java` source file to indicate what package it is supposed to be in. In the absence of such a declaration, the classes produced from the source file go into the *anonymous package,* which you can think of as holding all the outer-level packages (such as `java`).

2.1 The interpreter's classes

When the `java` program (the interpreter) runs the main procedure in a class, and that main procedure uses some other classes, let's say `A` and `p.B`, the interpreter looks for files `A.class` and `B.class` in places that are dictated by things called *class paths*. Essentially, a class path is a list of directories and *archives* (see §5 below for information on archives). If the interpreter's class path contains, let's say, the directories D_1 and D_2, then upon encountering a mention of class `A`, `java` will look for a file named D_1`/A.class` or D_2`/A.class`. Upon encountering a mention of `p.B`, it will look for D_1`/p/B.class` or D_2`/p/B.class`.

The class path is cobbled together from several sources. All Sun's java tools automatically supply a *bootstrap class path,* containing the standard libraries and such stuff. If you take no other steps, the only other item on the class path will be the directory '.' (the current directory). Otherwise, if the environment variable `CLASSPATH` is set, it gets added to the bootstrap class path. In past years, our standard class setup had '.' and the directory containing the `ucb` package (with our own special classes, lovingly concocted just for you), which we set up with the command

```
setenv CLASSPATH .:/home/ff/cs61b/lib/java/classes
```

(the colon is used in place of comma (for some reason) to separate directory names). The interpreter and compiler would then find the definition of a class such as `ucb.io.StdIO` in

```
/home/ff/cs61b/lib/java/classes/ucb/io/StdIO.class
```

These days, we use archive files instead, as described below in §5.

2.2 The compiler's classes

The compiler looks in the same places for `.class` files, but its life is more complicated, because it also has to find source files. By default, when it needs to find the definition of a class A, it looks for file A`.java` in the same directories it looks for A`.class`. This is the easiest case to deal with. If it does not find A`.class`, it will automatically compile A`.java`. To use this default behavior, simply make sure that the current directory ('.') is in your class path (as it is in our default setup) and put the source for a class `A` (in the anonymous package) in `A.java` in the current directory, or for a class `p.B` in `p/B.java`, etc., using the commands

```
javac A.java
javac p/A.java
```

respectively, to compile them.

It is also possible to put source files, input class files, and output class files (i.e., those created by the compiler) in three different directories, if you really want to (I don't think we'll need this). See the `-sourcepath` and `-d` options in the on-line documentation for `javac`, if you are curious.

3 Multiple classes in one source file

In general, you should try to put a class named *A* in a file named *A*`.java` (in the appropriate directory). For one thing, this makes it possible for the compiler to find the class's definition. On the other hand, although public classes must go into files named in this way, other classes don't really need to. If you have a non-public class that really is used *only* by class *A*, then you can put it, too, into *A*`.java`. The compiler will still generate a separate `.class` file for it. This semester, we will avoid this practice (it violates our automated style guidelines). It is probably more appropriate to *nest* such classes in class *A* instead.

4 Compiling multiple files

Java source files depend on each other; that is, the text of one file will refer to definitions in other files. As I said earlier, if you put these source files in the right places, the compiler often will automatically compile all that are needed even if it is only actually asked to compile one "root" class (the one containing the main program or main applet). However, it is possible for the compiler to get confused when (a) some `.java` files have *already* been compiled into `.class` files, and then (b) subsequently changed. *Sometimes* the compiler will recompile all the necessary files (that is, the ones whose source files have changed or that use classes whose source files have changed), but it is a bit dangerous to rely on this for the Sun compiler. The compiler also can't find class definitions if you "hide" them by putting, say, several classes into one file. The compiler guesses that class `A.B` is in file `A/B.java`. If it isn't, then it gives up. Fortunately, we won't be doing any such hiding this semester. Still, you can avoid both of these problems by asking listing all the necessary files for `javac` explicitly:

```
javac A.java p/B.java root.java
```

Since this is tedious to write, it is best to rely on a makefile to do it for you, as described below in §6.

5 Archive files

For the purposes of this course, it will be sufficient to have separate `.class` files in appropriate directories, as I have been describing. However in real life, when one's application consists of large numbers of `.class` files scattered throughout a bunch of directories, it becomes awkward to ship it elsewhere (say to someone attempting to run your Web applet remotely). Therefore, it is also possible to bundle together a bunch of `.class` files into a single file called a *Java archive* (or *jar file*). You can put the name of a jar file as one member of a class path (instead of a directory), and all its member classes will be available just as if they were unpacked into the directory structure described in previous sections.

The utility program '`jar`', provided by Sun, can create or examine jar files. Typical usage: to form a jar file `stuff.jar` out of all the classes in package `myPackage`, plus the files `A.class and B.class`, use the command

```
jar cvf stuff.jar A.class B.class myPackage
```

This assumes that myPackage is a subdirectory containing just `.class` files in package `myPackage`. To use this bundle of classes, you might set your class path like this:

`setenv CLASSPATH .:stuff.jar:`*other directories and archives*

6 The make utility

Even relatively small software systems can require rather involved, or at least tedious, sequences of instructions to translate them from source to executable forms. Furthermore, since translation takes time (more than it should) and systems generally come in separately translatable parts, it is desirable to save time by updating only those portions whose source has changed since the last compilation. However, keeping track of and using such information is itself a tedious and error-prone task, if done by hand. Therefore, most programming environments provide some kind of *project* or *compilation-control* facility. The UNIX `make` utility is a conceptually simple and general example. It accepts as input a description of the interdependencies of a set of source files and the commands necessary to compile them, known as a *makefile*; it examines the ages of the appropriate files; and it executes whatever commands are necessary, according to the description. For further convenience, it will supply certain standard actions and dependencies by default, making it unnecessary to state them explicitly.

There are numerous dialects of `make`, both among UNIX installations and (under other names) in programming environments for personal computers. In this course, I will use a version known as `make`[1]. Though conceptually simple, the `make` utility has accreted features with age and use, and is rather imposing in the glory of its full definition. This document describes only the simple use of `make`.

In addition to compilation (or re-compilation) control, there are other uses for `make`. It is useful in cases where one needs some kind of *preprocessing:* where the Java source files themselves result from applying some program to other inputs. The makefiles themselves also serve as a useful repository for scripts that perform numerous tasks incidental to compilation. I use them to build course material and copy it to where others can get to it.

6.1 Basic Operation and Syntax

Figure 1 is a sample makefile for compiling a simple editor program, `edit`, from eight `.java` files.

This file consists five *rules.* A rule consists of a line containing two lists of names separated by a colon, followed by one or more lines beginning with tab characters. Any line may be continued, as illustrated, by putting a backslash at the very end, which essentially acts like a space, combining the line with its successor. The '#' character indicates the start of a comment that goes to the end of that line.

The names preceding the colons are known as *targets*; they are most often the names of files that are to be produced. The names following the colons are known as *dependencies* of the targets. They usually denote other files (possibly, other targets) that must be present and up-to-date before the target can be processed. The lines starting with tabs[2] that follow the first line of a rule are called *actions.* They are shell commands (that is, commands that you could type in response to the Unix prompt) that get executed in order to create or bring up to date the target of the rule (we'll use the generic term *update* for the process of determining whether action is necessary on a particular target and then (if needed) building or rebuilding it).

Each rule says, in effect, that to update the targets, each of the dependencies must first be updated (recursively). Next, if a target does not exist (that is, if no file by that name exists) or if it does exist but is older than one of its dependencies (so that one of its dependencies was changed after the target was last updated), the actions of the rule are executed to create or update that target. The

[1]For "GNU `make`," GNU being an acronym for "GNU's Not Unix." `make` is "copylefted" (it has a license that *requires* free use of any product containing it). It is also more powerful than the standard `make` utility.

[2]Tabs, not blanks. Yes, I know: this is a really irritating design, because if you ever make the mistake of substituting blanks for the tab, you get errors (with very unhelpful messages). The `make` utility is of rather ancient lineage, and the file format has been this way since before most of you were born (literally).

```
# Makefile for a simple editor

# The jar file contains the entire collection of classes
# constituting the editor.
edit.jar: edit.class commands.class display.class files.class
        jar cf edit.jar edit.class commands.class display.class \
                        files.class

edit.class: edit.java commands.java display.java files.java
        javac -g edit.java commands.java display.java \
                 files.java

commands.class: edit.java commands.java display.java files.java
        javac -g edit.java commands.java display.java \
                 files.java

display.class: edit.java commands.java display.java files.java
        javac -g edit.java commands.java display.java \
                 files.java

files.class: edit.java commands.java display.java files.java
        javac -g edit.java commands.java display.java \
                 files.java
```

Figure 1: Sample makefile for an editor program. Adapted from "GNU Make: A Program for Directing Recompilation" by Richard Stallman and Roland McGrath, 1988.

program will complain if any dependency does not exist and there is no rule for creating it. To start the process off, the user who executes the `make` utility specifies one or more targets to be updated. The first target of the first rule in the file is the default.

In Figure 1, `edit.jar` is the default target. The first step in updating it is to update all the files listed as dependencies (a bunch of `.class` files). The remaining rules tell how to update each of these `.class` files. As you can see, they all look pretty much the same, and say that to update X`.class`:

> First update all the source files `edit.java`, `command.java`, etc. Next, if X`.class` is missing or is older than any of these source files, then execute `javac` on all the source files.

We chose to compile all the source files together like this because otherwise it is possible for the compiler to get confused by old `.class` files that are still lying around.

Updating the source (`.java`) files is easy. There are no rules for any of them, so `make` simply insists that they all exist in order to be considered up to date.

Now `edit.class`, for example, is up to date if it is younger (was created more recently) than all the files `edit.java`, `command.java`, and so forth. If instead it is older, `make` assumes that that one of those source files has been changed since the last compilation that produced `edit.class` and must be "rebuilt." Of course, if `edit.class` does not exist, then `make` also knows it has to be rebuilt. If rebuilding is necessary, `make` executes the action "`javac -g edit.java commands.java ...`", producing new `.class` files. In our particular case, if any one of the `.class` files needs to be rebuilt, they are *all* rebuilt. If two of them need to be rebuilt (let's say `edit.class` and `files.class`), then

`make` will execute the action for one of them and then, when it checks the other `.class` file, will discover that it has already been updated. Thus, you needn't worry that the `javac` command will be executed more than once.

Once all the `.class` files are up-to-date, `make` will check to see if any of them are younger than `edit.jar` (or if `edit.jar` does not exist). If any of the class files had to be rebuilt, then of course it will be younger, and `make` will execute the indicated action: "`jar cf edit.jar...`."

To invoke `make` for this example, one issues the command

`make -f` *makefile-name target-names*

where the *target-names* are the targets that you wish to update and the *makefile-name* given in the `-f` switch is the name of the makefile. By default, the target is that of the first rule in the file. Furthermore, you may (and usually do) leave off `-f` *makefile-name*, in which case it defaults to either `Makefile`, `makefile`, or (in the case of `make` only) `GNUmakefile`, whichever exists. It is typical to arrange that each directory contains the source code for a single principal program. By adopting the convention that the rule with that program as its target goes first, and that the makefile for the directory is named `Makefile`, you can arrange that, by convention, issuing the command `make` with no arguments in any directory will update the principal program of that directory.

It is possible to have more than one rule with the same target, as long as no more than one rule for each target has an action. Thus, I can also write the latter part of the example above as follows:

```
edit.class:
        javac -g edit.java commands.java display.java files.java

commands.class:
        javac -g edit.java commands.java display.java files.java

display.class:
        javac -g edit.java commands.java display.java files.java

files.class:
        javac -g

edit.class: edit.java commands.java display.java files.java
commands.class: edit.java commands.java display.java files.java
display.class: edit.java commands.java display.java files.java
files.class: edit.java commands.java display.java files.java
```

The order in which these rules are written is irrelevant. Which order or grouping you choose is largely a matter of taste, aside from which is the first (default) target.

Next, you can combine rules with the same dependencies and action. For example:

```
edit.class commands.class display.class files.class
        javac -g edit.java commands.java display.java files.java

edit.class commands.class display.class files.class: edit.java \
     commands.java display.java files.java
```

or just

```
edit.class commands.class display.class files.class: edit.java \
     commands.java display.java files.java
        javac -g edit.java commands.java display.java files.java
```

The example of this section illustrates the concepts underlying `make`. The rest of `make`'s features exist mostly to enhance the convenience of using it.

6.2 Variables

You can clarify the example from §6.1 considerably and eliminate redundancy by defining *variables* to contain the names of the files.

```
# Makefile for simple editor

JFLAGS = -g

JAVA_SRCS = edit.java \
       commands.java \
       display.java \
       files.java

CLASSES = edit.class commands.class display.class files.class

edit.jar : $(CLASSES)
        jar cf edit.jar $(CLASSES)

$(CLASSES): $(JAVA_SRCS)
        javac $(JFLAGS) $(JAVA_SRCS)
```

The (continued) line beginning "`JAVA_SRCS =`" defines the variable `JAVA_SRCS`, which can later be referenced as "`$(JAVA_SRCS)`". These later references cause the definition of `JAVA_SRCS` to be substituted verbatim before the rule is processed. It is somewhat unfortunate that both `make` and the shell use '$' to prefix variable references; `make` defines '$$' to be simply '$', thus allowing you to send '$'s to the shell in actions, where needed.

You will sometimes find that you need a value that is just like that of some variable, with a certain systematic substitution. For example, given a variable listing the names of all source files, you might want to get the names of all resulting `.class` files. You can rewrite the definition of CLASSES above to get this.

```
CLASSES = $(JAVA_SRCS:.java=.class)
```

The substitution suffix '`:.java=.class`' specifies the desired substitution. I now have variables for both the names of all sources and the names of all class files without having to repeat a lot of file names (and possibly make a mistake). (I have assumed here that each source file contains a single class whose name is derived from the source file. You can't use this trick if that isn't so.)

Variables may also be set in the command line that invokes `make`. For example, the makefile above contains what might look like an unnecessary definition of JFLAGS. However, defining it like that allows one to write:

```
make JFLAGS="-g -deprecation" ...
```

which passes an extra flag to `javac` (this one happens to give a fuller explanation of certain warning messages). Variable definitions in the command lines override those in the makefile, which allows the makefile to supply defaults.

6.3 Phony targets

It is often useful to have targets for which there are never any corresponding files. If the actions for a target do not create a file by that name, it follows from the definition of how `make` works that the actions for that target will be executed each time `make` is applied to that target (because it will think the target is missing). A common use is to put a standard "clean-up" operation into each of your makefiles, specifying how to get rid of files that can be reconstructed, if necessary. For example, you will often see a rule like this in a makefile.

```
.PHONY: clean

clean:
        rm -f *.class *~
```

Every time you issue the shell command "`make clean`," this action will execute, removing all `.class` files and Emacs old-version files.

The special `.PHONY` target tells `make` that `clean` is not a file, and is instead just the name of a target that is *always* out of date. Therefore, when you make the "`clean`" target, `make` will always execute the `rm` command, regardless of what files happen to be lying around. In effect, `.PHONY` tells `make` to treat `clean` as a command.

Another possible use is to provide a standard way to run a set of tests on your program—what are typically known as *regression tests*—to see that it is working and has not "regressed" as a result of some change you've made. For example, to cause the command

```
make check
```

to feed a test file through our editor program and check that it produces the right result, use:

```
.PHONY: check

check: edit
    rm -f test-file1
    java edit < test-commands1
    diff test-file1 expected-test-file1
```

where the test input file `test-commands1` presumably contains editor commands that are supposed to produce a file `test-file1`, and the file `expected-test-file1` contains what is supposed to be in `test-file1` after executing those commands. The first action line of the rule clears away any old copy of `test-file1`; the second runs the editor and feeds in `test-commands1` through the standard input, and the third compares the resulting file with its expected contents. If either the second or third action fails, `make` will report that it encountered an error.

6.4 Details of actions

By default, each action line specified in a rule is executed by the Bourne shell (as opposed to the C shell, which, most unfortunately, is more commonly used here). For the simple makefiles we are likely to use, this will make little difference, but be prepared for surprises if you get ambitious.

The `make` program usually prints each action as it is executed, but there are times when this is not desirable. Therefore, a '@' character at the beginning of an action suppresses the default printing. Here is an example of a common use.

```
edit.jar : $(CLASSES)
        @echo Creating edit.jar ...
        @jar cf edit.jar $(CLASSES)
        @echo Done
```

The result of these actions is that when `make` executes this final step for the `edit` program, the only thing you'll see printed is a line reading "`Creating edit.jar ...`" and, at the end of the step, a line reading "`Done`".

When `make` encounters an action that returns a non-zero exit code, the UNIX convention for indicating an error, its standard response is to end processing and exit. The error codes of action lines that begin with a '-' sign (possibly preceded by a '@') are ignored. Also, the `-k` switch to `make` will cause it to abandon processing only of the current rule (and any that depend on its target) upon encountering an error, allowing processing of "sibling" rules to proceed.

6.5 Including makefiles

A good way to create makefiles is to have a template that you include in your particular makefile—something like the example in Figure 2. We've prepared one like this already, so that in the very simplest case, your makefile can contain just:

```
JAVA_SRCS = edit.java commands.java display.java files.java

include $(MASTERDIR)/lib/java.Makefile.std
```

As you can probably guess, the `include` line is a special command that essentially gets replaced by the contents of the named file.

Figure 2 illustrates what such a template file might look like. It uses one obscure new feature that makes it possible to partially define an action, and allow others to add to it. The definition of the phony target `clean` uses two colons rather than one. This is a signal that there may be other "double-colon" rules for `clean`, complete with actions. They will all get used (in the order encountered). For example, if you include this particular template in a place where you want to define additional clean-up actions besides the ones defined in the template, you can write:

```
JAVA_SRCS = edit.java commands.java display.java files.java

include $(MASTERDIR)/lib/java.Makefile.std

clean::
        rm -rf test-output
```

which will cause `make clean` to remove the directory `test-output` as well as the class files and Emacs-generated files removed in the template.

```
# Standard definitions for make utility: Java version.

# Assumes that this file is included from a Makefile that defines
# JAVA_SRCS to be a list of Java source files to be compiled.
# It may optionally define OTHER_CLASSES to contain names of classes
# that aren't derivable from the names of the JAVA_SRCS files.
# The including Makefile may subsequently override JFLAGS (flags to
# the Java compiler), and JAVAC (the Java compiler's name), by putting
# these definitions after the "include".

# Targets defined:
#    default:Default entry.  Compiles classes from all source files.
#    clean:: Remove back-up files and files that make can reconstruct.
#            You can add additional clean-up actions by adding more
#            'clean::' targets (note the double colon) to your makefile.
#    check:  Look in the subdirectory tests for all files whose name ends
#            in '.sh'.  Each of these should be an executable shell script
#            (a file of commands such as you could enter at the command
#            prompt) that performs some test of the program.  Run each
#            and report all that fail (return a non-zero exit code).
#

JAVAC = javac

JFLAGS = -g

CLASSES = $(JAVA_SRCS:.java=.class) $(OTHER_CLASSES)

.PHONY: clean check default

# Default entry
default: $(CLASSES)

$(CLASSES): $(JAVA_SRCS)
        $(JAVAC) $(JFLAGS) $(JAVA_SRCS)

clean::
        /bin/rm -f $(CLASSES) *~

check:  $(CLASSES)
        cd tests; for test in *.sh; do \
            if ./$${test}; then \
                echo "$${tests}: OK."; \
            else \
                echo "$${tests}: FAILED."; \
            fi; \
        done
```

Figure 2: An example of a file of standard makefile definitions that can be included from a specific makefile to compile many simple collections of Java programs.