**Basic Compilation: javac, gcc, g++**

# 1 Compiling Java Programs

[The discussion in this section applies to Java 1.2 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.

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.

## 1.1 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.io.String`: the class named `String` that resides in the subpackage named `io` 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`).

**The Java interpreter.** 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 §1.4 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. Our standard class setup has '.' and the directory containing the `ucb` package (with our own special classes, lovingly concocted just for you). If you print its value with, for example,

```
echo $CLASSPATH
```

you'll see something like

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

(the colon is used in place of comma (for some reason) to separate directory names). It is also possible to set the class path (overriding the CLASSPATH environment variable) for a single program execution with

```
java -classpath PATH ...
```

but I really don't recommend this.

**The Java compiler.** 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`, or if it does find $A$`.class` but notices that it is older (less recently modified) than the corresponding source file $A$`.java`, it will automatically (re)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.

## 1.2 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.

### 1.3 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. You can ask `javac` to compile a whole bunch of files at once, simply by listing all of them:

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

and so on.

### 1.4 Archive files

For the purposes of this class, 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 previously described 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*

## 2 Compiling C and C++

`Gcc` is a publicly available optimizing compiler (translator) for C, C++, Ada 95, and Objective C that currently runs under various implementations of Unix (plus VMS as well as OS/2 and perhaps other PC systems) on a variety of processors too numerous to mention. You can find full documentation on-line under Emacs (use `C-h i` and select the "GCC" menu option). You don't need to know much about it for our purposes. This document is a brief summary.

### 2.1 Running the compiler

You can use `gcc` both to compile programs into object modules and to link these object modules together into a single program. It looks at the names of the files you give it to determine what language they are in and what to do with them. Files of the form *name*`.cc` (or *name*`.C`) are assumed to be C++ files and files matching *name*`.o` are assumed to be object (i.e., machine-language) files. For compiling C++ programs,

you should use `g++`, which is basically an alias for `gcc` that automatically includes certain libraries that are used in C++, but not C.

To translate a C++ source file, *F*`.cc`, into a corresponding object file, *F*`.o`, use the command

`g++ -c` *compile-options* *F*`.cc`

To link one or more object files—$F_1$`.o`, $F_2$`.o`, ...—produced from C++ files into a single executable file called *F*, use the command.

`g++ -o` *F* *link-options* $F_1$`.o` $F_2$`.o ...` *libraries*

(The *options* and *libraries* clauses are described below.)

You can bunch these two steps—compilation and linking—into one with the following command.

`g++ -o` *F* *compile-and-link-options* $F_1$`.cc ...` *other-libraries*

After linking has produced an executable file called *F*, it becomes, in effect, a new Unix command, which you can run with

`./`*F* `arguments`

where *arguments* denotes any command-line arguments to the program.

## 2.2 Libraries

A *library* is a collection of object files that has been grouped together into a single file and indexed. When the linking command encounters a library in its list of object files to link, it looks to see if preceding object files contained calls to functions not yet defined that are defined in one of the library's object files. When it finds such a function, it then links in the appropriate object file from the library. One library gets added to the list of libraries automatically, and provides a number of standard functions common to C++ and C.

Libraries are usually designated with an argument of the form `-l`*library-name*. In particular, `-lm` denotes a library containing various mathematical routines (sine, cosine, arctan, square root, etc.) They must be listed *after* the object or source files that contain calls to their functions.

## 2.3 Options

The following compile- and link-options will be of particular interest to us.

**-c** (Compilation option)
: Compile only. Produces `.o` files from source files without doing any linking.

**-D***name*=*value* (Compilation option)
: In the program being compiled, define *name* as if there were a line

  `#define` *name* *value*

  at the beginning of the program. The '=*value*' part may be left off, in which case *value* defaults to 1.

**-o** *file-name* (Link option, usually)
: Use *file-name* as the name of the file produced by `g++` (usually, this is an executable file).

**-l***library-name* (Link option)
: Link in the specified library. See above. (Link option).

**-g** (Compilation and link option)

Put debugging information for `gdb` into the object or executable file. Should be specified for *both* compilation and linking.

**-MM** (Compilation option)

Print the header files (other than standard headers) used by each source file in a format acceptable to `make`. Don't produce a `.o` file or an executable.

**-pg** (Compilation and link option)

Put profiling instructions for generating profiling information for `gprof` into the object or executable file. Should be specified for *both* compilation or linking. *Profiling* is the process of measuring how long various portions of your program take to execute. When you specify `-pg`, the resulting executable program, when run, will produce a file of statistics. A program called `gprof` will then produce a listing from that file telling how much time was spent executing each function.

**-Wall** (Compilation option)

Produce warning messages about a number of things that are legal but dubious. I strongly suggest that you *always* specify this and that you treat every warning as an error to be fixed.