CMPS 12L
Introduction to Programming Lab
Winter 2008

Lab Assignment 5
Due Friday February 29, 10:00 pm

In this assignment you will learn how to create an executable jar file containing a java program, and learn how to automate compilation and other tasks using Makefiles.

Jar Files
Recall the basic program `HelloWorld.java` from lab1:

```
// HelloWorld.java
class HelloWorld{
   public static void main(String[] args){
      System.out.println("Hello, world!");
   }
}
```

You can compile this in the normal way by doing `%javac HelloWorld.java` then run with `%java HelloWorld`. (Remember that `%` represents the Unix prompt.) Java provides a utility called jar (Java archive) for creating compressed archives of `.class` files. This utility can also be used to create an executable jar file which can be run by just typing its name. In particular, there will be no need to type `java` at the command line when invoking the program. To do this you must first create a `Manifest` file that specifies the entry point for program execution, i.e. which `.class` file contains the `main()` method to be run. Create a file called `Manifest` containing just one line:

```
Main-class: HelloWorld
```

If you don't feel like opening up an editor to do this, you can just type

```
% echo Main-class: HelloWorld > Manifest
```

As you learned in some previous lab assignments, the Unix command `echo` prints text to stdout, and the output redirect operator `>` assigns stdout to go to a file, in this case `Manifest`, rather than the screen. Now do

```
% jar cvfm myHello Manifest HelloWorld.class
```

The first group of characters after `jar` are options. (`c`: create a jar file, `v`: verbose output, `f`: second argument gives the name of the jar file to be created, `m`: third argument is a manifest file.) Consult the `man` pages to see other options to `jar`. Also see

```
http://java.sun.com/developer/Books/javaprogramming/JAR/basics/
```

to learn some of the many uses of jar files. Following the manifest file is the list of `.class` files to be archived, which in our example consists of just one file. The name of the executable jar file (second argument) can be anything you like. In this example it is called `myHello` to emphasize that it need not have the same name as the corresponding `.class` file. (For that matter, the manifest file need not be called `Manifest`.) However, there is really no reason not to call the `jar` file `HelloWorld` in this case, so instead let us do

```
% jar cvfm HelloWorld Manifest HelloWorld.class
```

Before we can run the jar file `HelloWorld`, we must first make it executable (by you) by doing

```
% chmod u+x HelloWorld
```

Re-read lab2 if any of this is unfamiliar. Now type

```
 % HelloWorld
```

to run the program, and observe we did not need to type `java` at the command line. The whole process can be accomplished by typing five Unix commands:

```
% javac -Xlint HelloWorld.java
% echo Main-class: HelloWorld > Manifest
% jar cvfm HelloWorld Manifest HelloWorld.class
% rm Manifest
% chmod u+x HelloWorld
```

Notice we have removed the (now unneeded) `Manifest` file. Note also that the `-Xlint` option to javac enables all recommended warnings. You can repeat the above process with any of the Java programs we've studied so far in this course, or any of your own projects. The only problem is that it's a big hassle to type all those lines. Fortunately there is a Unix utility which can automate this, and many other compilation processes.

Makefiles
Large programs are often distributed throughout many files which depend on each other in complex ways. Whenever one file changes, all the files which depend on that file must be recompiled. This is true in Java, C, C++, and most other languages. When working on such a program, it can be difficult and tedious to keep track of all the dependency relationships. The **make** utility automates this process. Make looks at dependency lines in a file named `Makefile` stored in your current working directory. The dependency lines indicate relationships among files, specifying a *target* file that depends on one or more *prerequisite* files. If a prerequisite file has been modified more recently than its target file, make updates the target file based on *construction commands* that follow the dependency line. Make normally stops if it encounters an error during the construction process. Each dependency line has the following format.

```
target: prerequisite-list
	construction-commands
```

The dependency line is composed of the target and the prerequisite-list separated by a colon. The construction-commands line *must* start with a tab character, and must follow the dependency line. Start an editor and copy the following lines into a file called `Makefile`.

```
# A simple Makefile for the HelloWorld program
HelloWorld: HelloWorld.class
	echo Main-class: HelloWorld > Manifest
	jar cvfm HelloWorld Manifest HelloWorld.class
```

```
        rm Manifest
        chmod u+x HelloWorld

HelloWorld.class: HelloWorld.java
        javac -Xlint HelloWorld.java

clean:
        rm –f HelloWorld.class HelloWorld

submit: Makefile HelloWorld.java
        submit cmps012l-pt.w08 lab5 Makefile HelloWorld.java
```

Anything following # on a line is a comment and is ignored by make. The second line says that the target `HelloWorld` depends on `HelloWorld.class`. If `HelloWorld.class` exists, and is up to date, then `HelloWorld` can be created by doing the construction commands which follow. (Don't forget that **all** indentation is accomplished via the **tab** character.) The next target is `HelloWorld.class` which depends on `HelloWorld.java`. The next target `clean`, is an example of what is called a *phony target* since it doesn't depend on anything, but just runs a command. Likewise the target `submit` doesn't compile anything, but does have some dependencies. Any target can be built (or perhaps performed if it is a phony target) by doing simply

```
% make target-name
```

where `target-name` is any target in the `Makefile`. Just typing `% make` by itself makes the first target in the `Makefile`. Try doing `% make clean` to get rid of all your previously compiled stuff, then do

```
% make
```

again to see the compilation performed from scratch. Notice the `clean` target says to remove some files using the Unix command `rm`. The `–f` option to `rm` is used here to suppress any error messages that might arise if the files to be removed do not exist. See the `man` pages for `rm` for more options.

What to turn in

This `Makefile` can be rewritten to work on any Java program by just replacing `HelloWorld` everywhere you see it by the appropriate program name. Write a `Makefile` that creates an executable jar file for the program `lawn.java` from programming assignment 1. This jar file should itself be called `lawn`. Your `Makefile` will include targets called `clean` and `submit`, as in the above example. Use this `Makefile` to resubmit `lawn.java`, along with the `Makefile` itself, to the assignment name `lab5`. Perform the usual checks to make sure everything was submitted correctly.