CMPS 11
Intermediate Programming
Lab Assignment 4

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 unix 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!");
   }
}
```

Ordinarily, this is compiled by doing `javac HelloWorld.java`, then executed by `java HelloWorld` at the unix command prompt. Java provides a utility called jar (Java archive) for creating compressed archives of `.class` files. This utility can be used to create an *executable jar file* containing a java program such as HelloWorld. When a program is archived in this way, one need not type `java` at the command line, just the name of the jar file. (Note this feature is not available on Windows and Mac platforms. It is possible in Linux and most other versions of Unix, other than Mac OS X.) To create a jar file, first create a `Manifest` file that specifies the entry point for program execution, i.e. which `.class` file contains the `main()` method to be executed. (All of the java programs we've seen so far consist of a single `.class` file. More complicated programs usually consist of multiple files.) Create a file called `Manifest` containing the single line:

```
Main-class: HelloWorld
```

You can do this without opening up an editor by doing:

```
% 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 a file, in this case `Manifest`, rather than the screen. Now do

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

The first group of characters after the `jar` command 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 the name of a manifest file. Consult the `man` pages to see other options to `jar`.) Following the manifest file name is the (space separated) list of `.class` files to be archived. In our example, this consists of the single file `HelloWorld.class`. The name of the executable jar file (second argument) can be anything you like, and in particular it need not match the prefix of any `.class` file. For that matter, the manifest file need not be called `Manifest`. Before we can run the jar file `HelloWorld`, we must first make it executable (to you the user) by using the `chmod` command, which you studied in lab2:

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

Now type

```
% HelloWorld
```

to run the program. 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. The `–Xlint` option to `javac` enables all recommended warnings. You can repeat this process with any of the Java programs we've studied, or with any of your own projects. The only problem is that it's a big hassle to type all those lines. Fortunately, unix has a utility that automates this, and many other compilation processes.

Makefiles

Large programs are often distributed throughout many files that depend on each other in complex ways. Whenever one file changes, all the files that depend on it 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 (space separated) `prerequisite-list` separated by a colon. Each `construction-commands` line *must* start with a tab character, and must follow the dependency line. Start an editor and create a file called `Makefile` containing the following:

```
# 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 cmps011-pt.w15 lab1 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 that 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 typing

```
% 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. Observe that the `submit` target simply runs the submit command. You can therefore submit a project by doing

```
% make submit
```

If you were to do this right now, with the Makefile exactly as above, you would get an error message from submit telling you that lab1 is closed.

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 `GCD.java` from programming assignment 3. This jar file should itself be called `GCD`. Your `Makefile` will include targets called `clean` and `submit`, as in the above example (but altered appropriately for this assignment). Use this `Makefile` to resubmit `GCD.java`, along with the `Makefile` itself, to the assignment name `lab4`. Pay close attention to any error messages you may see, especially from submit. Note that when we grade your work on this assignment, we will not re-evaluate the operation of your GCD program. However, it is required that the `GCD.java` file that you submit for this lab *at least compile*, otherwise no executable jar file can be created.