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. If you are using a Mac and have a properly installed jdk, you can do this on your own machine, otherwise login to the unix server to compete this lab.

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

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

One may compile this program in the usual manner by typing `javac HelloWorld.java` at the Unix command prompt, and then run it by doing `java HelloWorld`. 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 manner, 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 works on 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. As before the percent symbol `%` represents the Unix command prompt and you do not type it. 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, 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:

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

The parameter `u+x` says give the **U**ser the permission to e**X**ecute this file. You can learn more about `chmod` by typing `man chmod`. 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 tasks.

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 Unix `make` utility automates this process. The `make` command 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. The `make` command normally stops if it encounters an error during the construction process. Each dependency line has the following format.

```
target: prerequisite-list
<tab--->construction-command(s)
```

The dependency line is composed of the `target` and the (space separated) `prerequisite-list` separated by a colon. Each `construction-command` 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
```

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. 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.

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.

1. Make a directory named lab5.
2. Put a copy of the program `TwentyOnePickup.java` from lab4 in the new lab5 directory.
3. Write a `Makefile` that creates an executable jar file for the program `TwentyOnePickup.java`. This jar file should itself be called `TwentyOnePickup`.
4. Run make to build the jar file.
5. Confirm the jar file is correct which can be done by typing the command:
   ```
   java -jar TwentyOnePickup
   ```
6. Your `Makefile` should include a target called `clean`, as in the above example (but altered appropriately for this assignment).
7. Create a zip file, `lab5.zip`, of your `lab5` folder which should at this point contain `Makefile`, `TwentyOnePickup.java`, `TwentyOnePickup.class`, and the jar file `TwentyOnePickup`. Both OS X and Windows have direct support in their respective file browsers for creating zip files. If you need help, ask a TA or a friend. Other similar compressed archive formats are not acceptable. If you completed the above steps on the unix server, you can create a zip file using the following steps:
   a. Type: cd .. which will move you into the parent directory of lab5.
   b. Type: zip lab5.zip lab5/* which will create a zip file named lab5zip containing everything in the directory lab5. Note: typing zip lab5.zip lab without the "*" will create a zip file containing an EMPTY lab5 folder. This is NOT what you want.
   c. Transfer lab5.zip to your local machine using your preferred file transfer program (see lab 2) and submit it. Note you can confirm you created it correctly (that it isn't empty) by unzipping it on your local machine and seeing that all of the files are there in the lab5 folder as expected.

    d. As a final note, if you do this from the lab machines, it is possible to locate the remove file directly without transferring it. Ask a TA for help.

8. Submit this `lab5.zip` file under lab5 in Canvas.