#### CMPS 12M

**Data Structures Lab**

**Lab Assignment 1**

The purpose of this assignment is manyfold: (1) get setup to use git, a revision control system, for all assignments (2) learn how to create an executable jar file containing a Java program, and (3) learn to automate compilation and other tasks using Makefiles. In addition, you can also optionally learn about the unix timeshare.

**Part 1: HELLO WORLD**

Start by creating a directory for this course. Fire up a terminal. (Let % denote the prompt.) Run:

% cd ~/Desktop

% mkdir CMPS12B

% cd CMPS12B

% pwd

(You should see which directory you’re in. You’re free to create this directory anywhere, but follow the above if you’re not sure.)

In this directory, create a file HelloWorld.java with the following text.

//-----------------------------------------------------

// HelloWorld.java

// Prints "Hello world", the first program you should always write

//-----------------------------------------------------

class HelloWorld {

static public void main(String[] args) {

System.out.println("Hello world");

}

}

Go to a terminal, and get into the directory with this file. Run

% javac HelloWorld.java

% java HelloWorld

You should see “Hello World” printed out. If not, you’ll need to install Java on your machine. Or, you can optionally work on the unix timeshare. Talk to the lab instructor for more directions.

**Part 2: GIT**

Go to “How to set up git” post on Piazza, and follow all the instructions. Create directory Lab1 where you will create the files needed for the remainder of this lab.

Throughout this quarter, when working on an assignment you should periodically "commit" your work to your repository (a local operation) and also "push" your repository to the server to provide a backup of the repository on the server. You should do this at least once an hour during a long programming session and at the end of every programming session. Once git is setup, those two commands are short and easy to type. Here they are again to remind you:

git commit –a –m "a short message about what you just changed"

git push

**Part 3: AFS**

**(If you prefer working on your local machine and are confident of compiling Java programs, you can skip this part. This is for people who want to work on the unix timeshare.)**

Logon to your ITS [unix timeshare](http://its.ucsc.edu/unix-timeshare/index.html) account at unix.ucsc.edu. If you don't know how to do this, ask for help at a lab session, or see:

<https://classes.soe.ucsc.edu/cmps012a/Winter16/lab1.pdf>

If you didn't already create your ssh keys on unix.ucsc.edu in the git setup above, copy them to the .ssh directory on unix.ucsc.edu. From Windows machines you will need to use an SFTP client such as FileZilla to transfer the files from your personal machine to the university server. On OSX machines you can use the command "scp somefile yourUserName@unix.ucsc.edu:destinationPath" to transfer a file. The destinationPath can be blank but don't leave off that colon (:). There are also applications such as Fugu if you prefer for OSX.

Clone your git repository on the unix server (see the git clone command above). From within Lab1 create a subdirectory called private, then set access permissions on the new directory so that other users cannot view its contents. Do all this by typing the lines below. The unix prompt is depicted here as %, although it may look different in your login session. Those lines without the unix prompt are the output of your typed commands. The first command will clone your git repository and put it in a directory named 12b.

% git clone git@git.soe.ucsc.edu:classes/cmps012b/winter17/UCSCNAME CMPS12B

% cd CMPS12B

% cd Lab1

% mkdir private

% fs setacl private system:authuser none

% fs listacl private

Access list for private is

Normal rights:

foobar rlidwka

Here foobar will be replaced by your own cruzid. The last line of output says that your access rights to directory private are rlidwka which means: read, list, insert, delete, write, lock, and administer. In other words you have all rights in this directory, while other users have none. If you are unfamiliar with any unix command, you can view its manual page by typing: man <command name>. (Do not type the angle brackets <>.) For instance man mkdir brings up the man pages for mkdir. Man pages can be very cryptic, especially for beginners, but it is best to get used to reading them as soon as possible.

Under AFS, fs denotes a file system command, setacl sets the access control list (ACL) for a specific user or group of users, and listacl displays the access lists for a given directory. The command

% fs setacl <some directory> <some user> <some subset of rlidwka or all or none>

sets the access rights for a directory and a user. Note that setacl can be abbreviated as sa and listacl can be abbreviated as la. For instance do la on your home directory:

% fs la ~

Access list for /afs/cats.ucsc.edu/users/a/foobar is

Normal rights:

system:authuser l

foobar rlidwka

The path /afs/cats.ucsc.edu/users/a/foobar will be replaced by the full path to your home directory, and your own username in place of foobar. Note that ~ (tilde) always refers to your home directory, . (dot) always refers to your current working directory (the directory where you are currently located) and .. (dot, dot) refers to the parent of your current working directory. The group system:authuser refers to anyone with an account on the ITS unix timeshare. Thus by default, any user on the system can list the contents of your home directory. No other permissions are set for system:authuser however, so again by default, no one else can read, insert, delete, write, lock, or administer your files.

Do fs la ~/CMPS12B and verify that the access rights are the same for the child directory CMPS12B. Create a subdirectory of private, call it anything you like, and check its access rights are the same as for its parent. Thus we see that child directories inherit their permissions from the parent directory when they are created. To get a more comprehensive list of AFS commands do fs help. For instance you will see that fs lq shows your quota and usage statistics. For more on the Andrew File System go to

<http://claymore.rfmh.org/public/computer_resources/AFSinfo.html>.

**Part 4: Jar Files**

Create the following file HelloUser.java in your Lab1 directory.

//-----------------------------------------------------------------------------

// HelloUser.java

// Prints greeting to stdout, then prints out some environment information.

//-----------------------------------------------------------------------------

class HelloUser{

public static void main( String[] args ){

String userName = System.getProperty("user.name");

String os = System.getProperty("os.name");

String osVer = System.getProperty("os.version");

String jre = System.getProperty("java.runtime.name");

String jreVer = System.getProperty("java.runtime.version");

String jvm = System.getProperty("java.vm.name");

String jvmVer = System.getProperty("java.vm.version");

String javaHome = System.getProperty("java.home");

long freemem = Runtime.getRuntime().freeMemory();

long time = System.currentTimeMillis();

System.out.println("Hello "+userName);

System.out.println("Operating system: "+os+" "+osVer);

System.out.println("Runtime environment: "+jre+" "+jreVer);

System.out.println("Virtual machine: "+jvm+" "+jvmVer);

System.out.println("Java home directory: "+javaHome);

System.out.println("Free memory: "+freemem+" bytes");

System.out.printf("Time: %tc.%n", time);

}

}

You can compile this in the normal way by doing javac HelloUser.java then run it by doing the command java HelloUser. Java provides a utility called jar for creating compressed archives of executable .class files. This utility can also be used to create an executable jar file that can easily be shared. You can run it by calling java –jar <NAME OF JAR>

To create a jar file, 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 executed. Create a text file called Manifest containing just one line:

Main-class: HelloUser

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

% echo Main-class: HelloUser > Manifest

The unix command echo prints text to stdout, and > redirects the output to a file. Now do

% jar cvfm HelloUser.jar Manifest HelloUser.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. The second argument HelloUser.jar is the name of the jar file to be created. The name of this file can be anything you like, i.e. it does not have to be the same as the name of the .class file containing function main(). Ideally, you should give it the extension .jar, though it’s not necessary. For that matter, the manifest file need not be called Manifest, but this is the convention. Following the manifest file is the list of .class files to be archived. In our example this list consists of just one file: HelloUser.class.

Now type java –jar HelloUser.jar to run the program. The whole process can be accomplished by typing five lines:

% javac –Xlint HelloUser.java

% echo Main-class: HelloUser > Manifest

% jar cvfm HelloUser.jar Manifest HelloUser.class

% rm Manifest

Notice we have removed the now unneeded manifest file. Note also that the –Xlint option to javac enables recommended warnings. The only problem with the above procedure is that it’s a big hassle to type all those lines. Fortunately there is a unix utility that can automate this and other processes.

**Part 5: Makefiles**

Large programs are often distributed throughout many files that depend on each other in complex ways. Whenever one file changes all the files depending on it must be recompiled. When working on such a program it can be difficult and tedious to keep track of all the dependencies. The Unix make utility automates this process. The command make looks at dependency lines in a file named Makefile. The dependency lines indicate relationships between source files, indicating a *target* file that depends on one or more *prerequisite* files. If a prerequisite has been modified more recently than its target, make updates the target file based on *construction commands* that follow the dependency line. make will normally stop 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 may consist of more than one line, but each line *must* start with a tab character. Start an editor and copy the following lines into a file called Makefile.

# A simple Makefile

HelloUser: HelloUser.class

echo Main-class: HelloUser > Manifest

jar cvfm HelloUser.jar Manifest HelloUser.class

rm Manifest

HelloUser.class: HelloUser.java

javac -Xlint HelloUser.java

clean:

rm -f HelloUser.jar HelloUser.class

Anything following # on a line is a comment and is ignored by make. The second line says that the target HelloUser depends on HelloUser.class. If HelloUser.class exists and is up to date, then HelloUser can be created by doing the construction commands that follow. Remember that *all* indentation is accomplished via the tab character. The next target is HelloUser.class which depends on HelloUser.java. The next target clean is what is sometimes called a *phony target* since it doesn’t depend on anything and just runs a command. Any target can be built (or perhaps executed if it is a phony target) by doing make <target name>. Just typing make creates the first target listed in the Makefile. Try this by doing make clean to get rid of all your previously compiled stuff, then do make again to see it all created again. Your output from make should look something like:

% make

javac -Xlint HelloUser.java

echo Main-class: HelloUser > Manifest

jar cvfm HelloUser Manifest HelloUser.class

added manifest

adding: HelloUser.class(in = 1577) (out= 843)(deflated 46%)

rm Manifest

The make utility allows you to create and use macros within a Makefile. The format of a macro definition is ID = list where ID is the name of the macro (by convention all caps) and list is a list of filenames. Then $(list) refers to the list of files. Move your existing Makefile to a temporary file, then start your editor and copy the following lines to a new file called Makefile.

#------------------------------------------------------------------------------

# A Makefile with macros

#------------------------------------------------------------------------------

JAVASRC = HelloUser.java

SOURCES = README Makefile $(JAVASRC)

MAINCLASS = HelloUser

CLASSES = HelloUser.class

JARFILE = HelloUser.jar

all: $(JARFILE)

$(JARFILE): $(CLASSES)

echo Main-class: $(MAINCLASS) > Manifest

jar cvfm $(JARFILE) Manifest $(CLASSES)

rm Manifest

$(CLASSES): $(JAVASRC)

javac -Xlint $(JAVASRC)

clean:

rm $(CLASSES) $(JARFILE)

Run this new Makefile and observe that it is equivalent to the previous one. The macros define text substitutions that happen before make interprets the file. Study this new Makefile until you understand exactly what substitutions are taking place. Now create your own Hello program HelloUser2.java that only prints the first line (“Hello <USER>”). Add HelloUser2.java to the JAVASRC list, add HelloUser2.class to the CLASSES list and change MAINCLASS to HelloUser2. Also change the name of JARFILE to just Hello.jar (emphasizing that the jar file can have any name.)

#------------------------------------------------------------------------------

# Another Makefile with macros

#------------------------------------------------------------------------------

JAVASRC = HelloUser.java HelloUser2.java

SOURCES = README Makefile $(JAVASRC)

MAINCLASS = HelloUser2

CLASSES = HelloUser.class HelloUser2.class

JARFILE = Hello.jar

all: $(JARFILE)

$(JARFILE): $(CLASSES)

echo Main-class: $(MAINCLASS) > Manifest

jar cvfm $(JARFILE) Manifest $(CLASSES)

rm Manifest

chmod +x $(JARFILE)

$(CLASSES): $(JAVASRC)

javac -Xlint $(JAVASRC)

clean:

rm $(CLASSES) $(JARFILE)

This new Makefile compiles both HelloUser classes (even though neither one depends on the other.) Notice however the entry point for program execution has been changed to function main() in your program HelloUser2.java. Macros make it easy to make changes like this, so learn to use them. To learn more about Makefiles follow the links posted on the class webpage.

We've discussed three Makefiles in this project. If you rename them Makefile1, Makefile2 and Makefile3 respectively (since you can't have three files with the same name), you'll find that the make command does not work since a file called Makefile no longer exists. Instead of renaming files to run the Makefile you want, you can use the -f option to the make command, and specify the name of your Makefile. For instance

% make -f Makefile2

runs Makefile2. If you want to specify something other than the first target, place it after the file name on the command line. For instance

% make –f Makefile3 clean

runs the clean target in Makefile3.

**What to turn in:**

Phew! We’re finally done. At this point, you should have created a folder CMPS12B corresponding to your git repo. In that, there should a folder Lab1. In that folder, the following files should be pushed into the repo:

HelloUser.java, HelloUser2.java, Makefile1, Makefile2, Makefile3, README

**You must follow the naming convention exactly, including the capitalization. Do not call your file “hellouser.java” or “helloUser.java”, etc.**

**All labs must be done individually.** All program source files you turn in for this and other assignments should begin with a comment block giving your name and cruzid, a short description of its role in the project, the file name, and any special instructions for compiling and/or running it. Be sure to create a README file that lists all the files being submitted (including README) along with any special notes to the grader.

Please run the check script as given on the course website (under Homework or Labs) to verify your file names. You can look at “Sesh’s README” on the course website (under Labs) to make sure you’re doing things correctly. Just to be clear, “Sesh’s README” for the README of my solution. It gives you (again) the list of files to be submitted, as well as what we expect from your solution.

You should use "git commit –a –m 'msg' " and "git push" frequently (at least once per session). It is a great way to avoid losing your work. To actually "submit" your assignment, while in the assignment directory (e.g. Lab1 for this assignment) and after having done "git push" of your latest work, type "git log".

There is a lot in this assignment so start early and ask questions in lab section or office hours if anything is unclear.

Grading:

* (10 points) Everything done correctly
* (9 points) No README, but everything else correct
* (8 points) Some error in makefiles
* (7 points) Only turning in java files through git
* (5 points) Setting up git and submitting something

**If files are named incorrectly, you get no credit.**