CMPS 12M

Introduction to Data Structures

Lab Assignment 2

Objective

The purpose of this assignment is to learn the absolute basics of version control with git to avoid losing your work, do some input and output using Java, and make a new makefile.

Quick Introduction to Git

Git is a popular version control tool designed for the distributed development needs of the Linux kernel community. Although it has many advanced features and alternate interfaces, git can be useful even for just keeping track of a few files in a project locally. In fact, I recommend that you use git to version control every file that you create. Without version control you will probably experience the sinking feeling of making just one more small edit to your program right before the deadline causing the whole thing to fail, and then not remembering what you changed and having no idea how to get back to the working state that you had just a few seconds ago.

First set up git with the following commands:

```
git config --global user.name "Your Name"
```

```
git config --global user.email "your_email@whatever.com"
```

Next, create a new directory lab2 for this lab within the existing directory structure you created in the last lab. Navigate into this new directory using the cd command. Initialize a new git repository:

```
git init
```

Git works by keeping a staging area of changed files that are going to be included in the next commit. The general workflow is to add changed files or new files to the staging area, then commit all the changes in one chunk with a message describing what the change does.

Create a file named Makefile and put some text in it (the contents aren't critical). Add this file to the staging area:

```
git add Makefile
```

Now commit all the changes in the staging area with a message:

```
git commit -m "Initial commit of a blank makefile"
```

If you are ever confused about the current status of the repository you can do:

```
git status
```

The reply will indicate if there are changes in the directory and whether or not they are in the staging area ready to be committed.

Make a change to Makefile and save the changed version back to the same filename. Try "git status" again to see that git knows something has changed. Tell git to stage the changes, then look at the status again:

```
git add Makefile
```

```
git status
```

Now commit the changes:

```
git commit -m "Add variables to makefile following lab1"
```

You can also edit or add multiple files at once before doing a commit. This groups all the changes into one logical unit. Create two more blank files named greet.java and greetings.java. Now add them and commit them:

```
git add greet.java greetings.java
```

```
git commit -m "Initial blank versions of java source files"
```

Take a look at the history of what has happened in your repository using the log command:

```
git log
```

In this lab you shouldn't need any more git commands. If you accidentally commit an incorrect change, the simplest solution is to just edit the files to what they should be and commit a new change with a message explaining what happened. This will make the history show the incorrect change followed by a correcting change.

Continue the rest of the lab using git to version control your files. Group your changes into logical units that have a specific purpose. Make sure your messages explain the meaning behind the change, not just what changed. One change should be adding one feature, fixing one bug, or something less than 1/2 hour of development work.

Standard Input and Output in Java

In Unix, every program that is started is given a standard input stream and a standard output stream. By default, the standard input stream reads from the keyboard and the standard output displays to the screen. In the previous lab you used System.out.println to write strings to the standard output stream.

To read input from the standard input stream in Java you need to use some of the objects in the java.io package. Here is code that constructs a suitable input object:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

The code creates a variable named "in" that is a new object ready to read from the standard input stream. We're constructing it this way because the default System.in object only has very low level input functions. InputStreamReader objects read from input streams and provide higher-level methods. Finally, we wrap the InputStreamReader object inside a BufferedReader to provide I/O buffering. It is

good practice to always buffer I/O for efficiency. Once you have the BufferedReader, you can read a line from the input stream using in.readLine() which returns a string.

One slight complication is that input operations can fail. When reading from a file, you might try to open a nonexistent file, or you may not have permission to read the file. To accommodate these types of problems, I/O operations may throw IOExceptions. In Java, if you ever call a method that might throw an exception you must either catch the exception and handle it, or declare that your method may also throw the exception.

Below is an example program that asks the user for their name, then greets them by name. In the example we choose to catch any IO exceptions and display a stack trace to help with debugging.

```
// greet.java
// Asks user to type their name, then says hello.

import static java.lang.System.*;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

class greet {
    public static void main( String[] args ) {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        try {
            System.out.println("What is your name?");
            String name = in.readLine();
            System.out.println("Hello, " + name + ".");
        } catch(IOException io) {
            io.printStackTrace();
        }
    }
}
```

Save the code into greet.java and set up a makefile following lab 1. Once that is working, modify the code to loop forever, repeatedly greeting the user. You should be able to terminate the program by typing Control-C.

Reading from a File

Reading from a file is similar to reading from System.in. Use code similar to the following:

```
BufferedReader in = new BufferedReader(new FileReader("list.txt"));
```

Once you have the BufferedReader object associated with the file you can again call the readLine() method to read a line at a time.

Dealing with files means we have to worry about more exceptions. The call to the FileReader constructor may throw an IOException if the file does not exist or if there is some other problem opening the file. There is also the issue of knowing when we have reached the end of the file. When using readline(), the return value will be null if no more characters remain in the file.

Below is an example program that greets everyone in the people.txt file. In the example we do not catch any exceptions. Instead we declare that main may throw the IOException. During program execution if an exception occurs that is not caught, the JRE will halt and display information about the exception.

```
// greetings.java
// Greet everyone listed in people.txt

import static java.lang.System.*;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileReader;

class greetings {
    public static void main( String[] args ) throws IOException {
        BufferedReader in = new BufferedReader(
            new FileReader("people.txt"));
        while(true) {
            String name = in.readLine();
            if (name == null) break;
            System.out.println("Hello, " + name + ".");
        }
        in.close();
    }
}
```

Pasing Lines

Once you are reading lines from a file you might want to parse them. For example, you might want to split the line wherever there is a field separator character such as a tab, colon, or other character. Luckily there is a method of the String class called split that can do this for you.

```
public String[] split(String regex, limit)

public String[] split(String regex)
```

Here is an example code snippet using the function:

```
String txt = "Nathan,(408)5551212";

String[] matches = txt.split(",");

System.out.println("Hello, " + matches[0]);
```

Modify the greetings.java code so that it reads full names from a file called directory.txt. Each line in directory.txt contains a name, then a comma, then a phone number. Your updated version of greetings should parse each line and greet the person by name. The phone number should never be printed.

What to Turn In

All files you turn in for every assignment and lab should begin with a comment block that includes your name, CruzID, class, date, filename, short description of the file's role in the assignment, and any special instructions related to the file. Also create a file called README. The README file should have the normal comment block, then list all the files being submitted (including itself) along with any special notes to the graders.

For this lab, submit the following files:

```
README

Makefile

greet.java

greetings.java

git_history.txt
```

To submit, use the submit command.

```
submit cmps12b-nojw.f14 lab2 README Makefile greet.java greetings.java
git_history.txt
```

All your files should be checked into a local git repository. As you edit and develop your files you should commit changes in logical units to your repository with meaningful messages. Include a file named "git_history.txt" that is the result of the following command executed right before submission:

```
git log --graph --oneline > git_history.txt
```

You should have a small Java program in greet.java with an associated makefile. There should be a phony target "runGreet" so that you can do "make runGreet" and it will start the greet program and build it if necessary. The greet program should repeatedly ask for the user's name and greet the user by name until the program is forcibly terminated.

You should have a small Java program in greetings.java that can also be built by the same makefile. There should be a phony target "runGreetings" that runs the greetings application and builds it if necessary. The greetings program should read from the file "directory.txt" and greet everyone in the file. The file contents contain one record per line, with a name, comma, then phone number on each line.

Note that in this lab you will have to modify the makefile from the previous lab to build more than one class from the same makefile. You will also have to add more phony targets and enter the dependencies between targets properly. In this lab it is not necessary to create executable jar files; it is enough to make the phony targets runGreet and runGreetings work (perhaps by using the java command).