1 Getting the Assignment

First, you need to make your personal copy of the assignment through GitHub classroom. You can do that at the following link:

https://classroom.github.com/a/HkrcqLUu

After you have your version of the assignment repository made, find the URL to clone with HTTPS. You will need this URL for later.

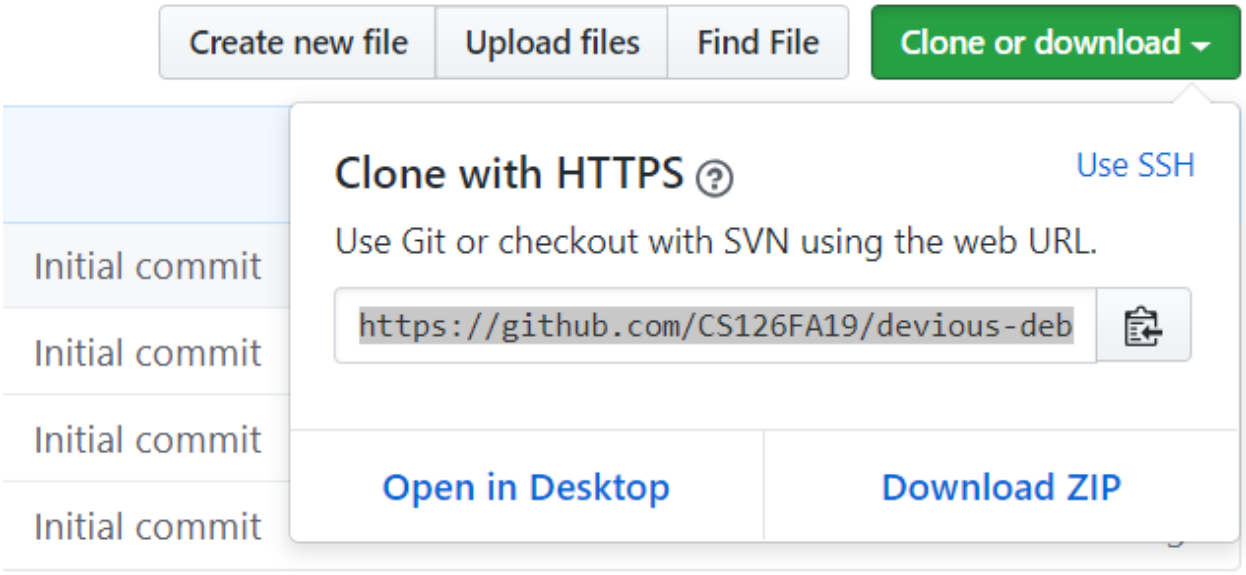

Figure 1: The HTTPS clone link

Now we need to import your assignment repository into your local IntelliJ. This tutorial uses images from Windows. If you're following along with Mac or Linux the location of the options shown may be slightly different. From any project in IntelliJ:

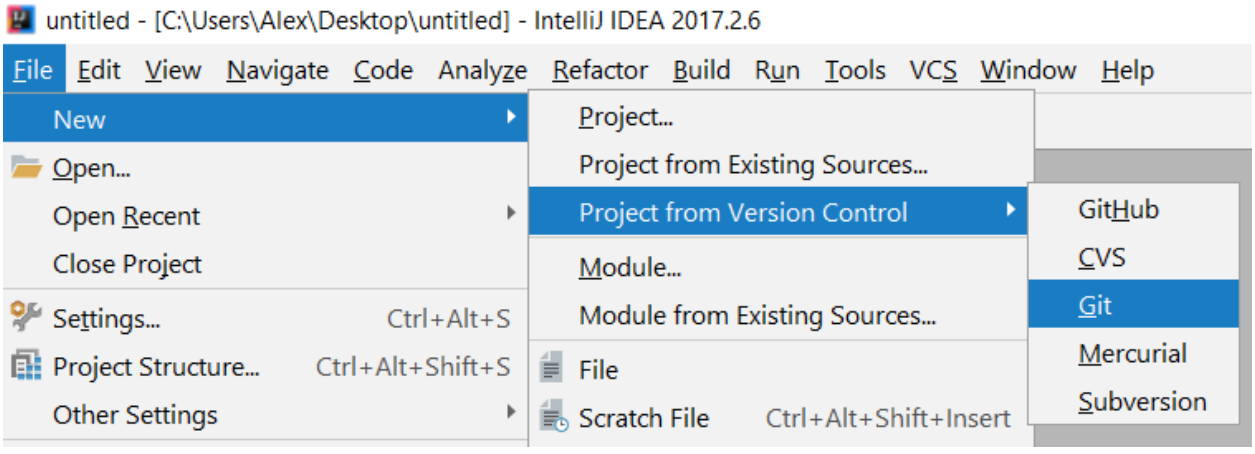

Figure 2: File -> New -> Project from Version Control -> Git

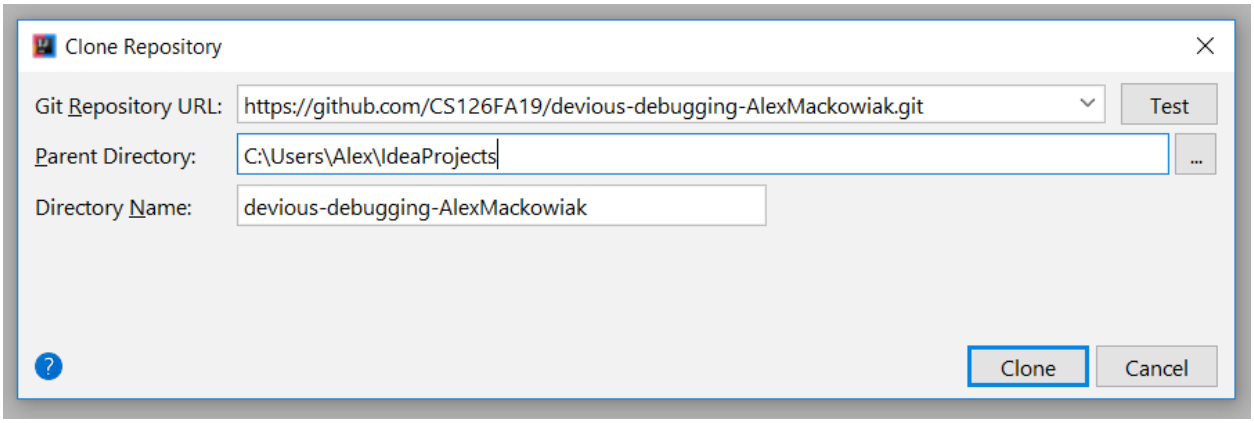

Figure 3: Clone your version of the assignment with the link from earlier as the Git Repository URL

This assignment requires Java 8+ to run. Please ensure that you are running in at least Java 8.

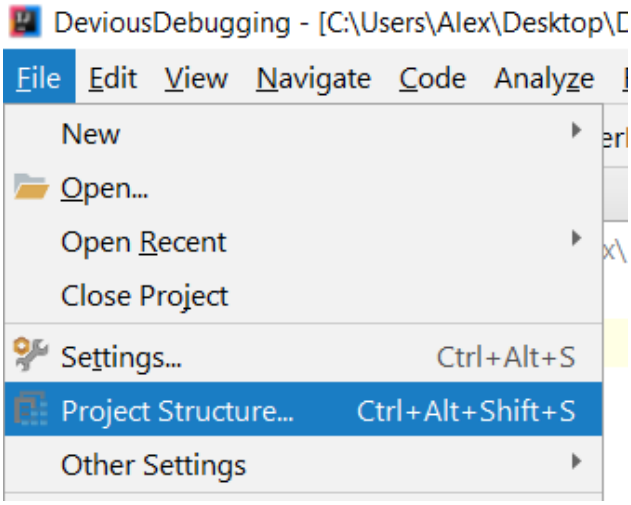

Figure 4: File → Project Structure

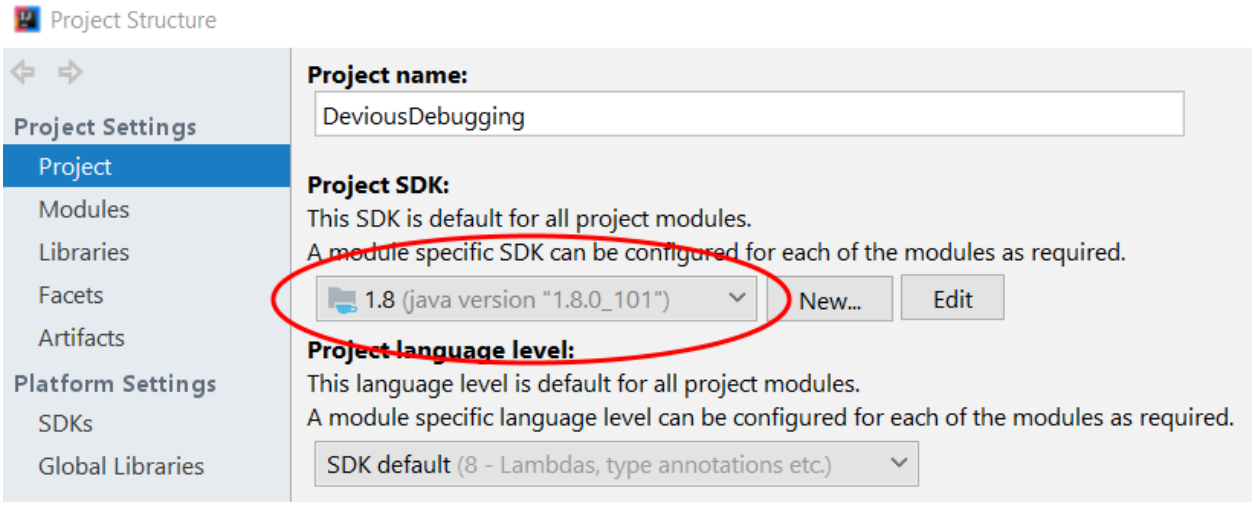

Figure 5: Ensure SDK is at least 1.8

2 The Assignment

Put your NetID in YourNetID.txt
Do NOT modify any of the code

First and foremost, you absolutely must do the following two things in order to have completed the assignment:

1. Put your NetID in the file under src/ called "YourNetID.txt". Make sure you save the file after adding your NetID. Doing this will ensure that the final output of the program is unique to you, so we know you personally completed the assignment.
2. Absolutely do not modify any of the files given to you other than "YourNetID.txt". If you change anything at all in these files, including adding whitespace, the final program output will be incorrect. Because the files are in version control, IntelliJ will make the file name blue to let you know if anything at all was changed from the version in your local git repository. The only file within the src/ directory that should be blue at any point is "YourNetID.txt" (Files in other directories such as the .idea directory are not tracked by our hash, and therefore are ok if they are changed).

This assignment is an interactive debugging tutorial designed to take 15-30 minutes. Completing this assignment, and actively using the debugging tools taught on future assignments, is likely to save you far more than 15-30 minutes of debugging time over the course of the semester.

If you encounter any problems during this tutorial, please feel free to ask on Piazza, or if it seems like a serious issue send an email to CS126SP20@gmail.com

You are tasked with using the tools in the IntelliJ debugger to work your way through 5 methods in HackerMain.java without modifying the code. Each method will exit the program if given the wrong answer. Please keep in mind that while this assignment does not ask you to actively debug problems in the code, the techniques used are widely applicable to finding and fixing bugs.

To begin, you will need to place some breakpoints. Setting a breakpoint is done by clicking directly to the right of the line number. This will leave a red circle and give the line a red tint.

```
26          long passcodeValue = (8 * Math.random()) > 4 ? 65535
27          long studentAnswer = studentInputReader.nextLong();
```

Figure 6: A line with a breakpoint

You will need to put breakpoints on the following lines:

Line 27: `long studentAnswer = studentInputReader.nextLong();`

Line 38: `System.out.println("This loop ...");`

Line 53: `randomSecQuestions.get(...).invoke(null);`

Line 78: `Collections.shuffle(randomBankAccounts);`

Line 84: `if(currentBankAccount.getId() == 126 ... )`

Line 98: `int fakeAccountId = studentInputReader.nextInt();`

Now run the program in debug mode to start the assignment. This can be done by clicking the bug icon next to the play button icon that normally runs the program.

Figure 7: The debug button

Alternatively you can click the play button next to the main method and select the Debug option.

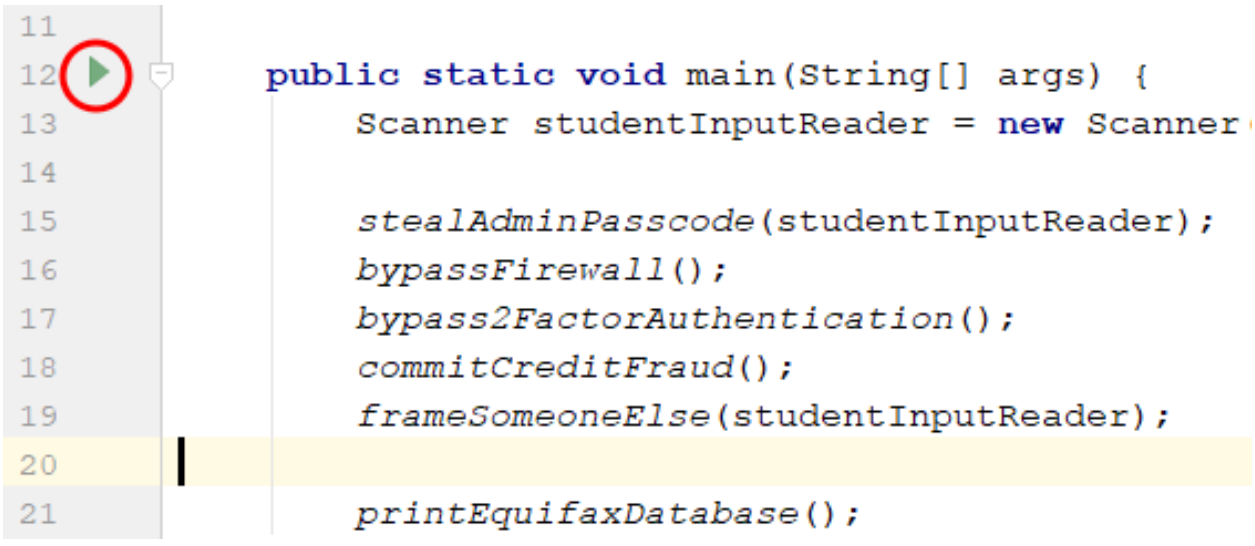

Figure 8: The alternate play button

3 Introduction to the IntelliJ Debugger

You should now be running the program in debug mode. Debug mode will run the program normally until the next line to be executed has a breakpoint. Right before executing a line with a breakpoint, the program will pause execution. While paused, you can see the values of variables, advance execution one line at a time, and make use of many other useful features.

The program should be paused right before executing the line
`long studentAnswer = studentInputReader.nextLong();` allowing you to see the debugging screen. This tutorial was written using the 2017 version of IntelliJ, so your debugging screen may look slightly different, but it should have the same functionality. If your buttons look different, hovering the mouse over a button for about 2 seconds will tell you what the button does.

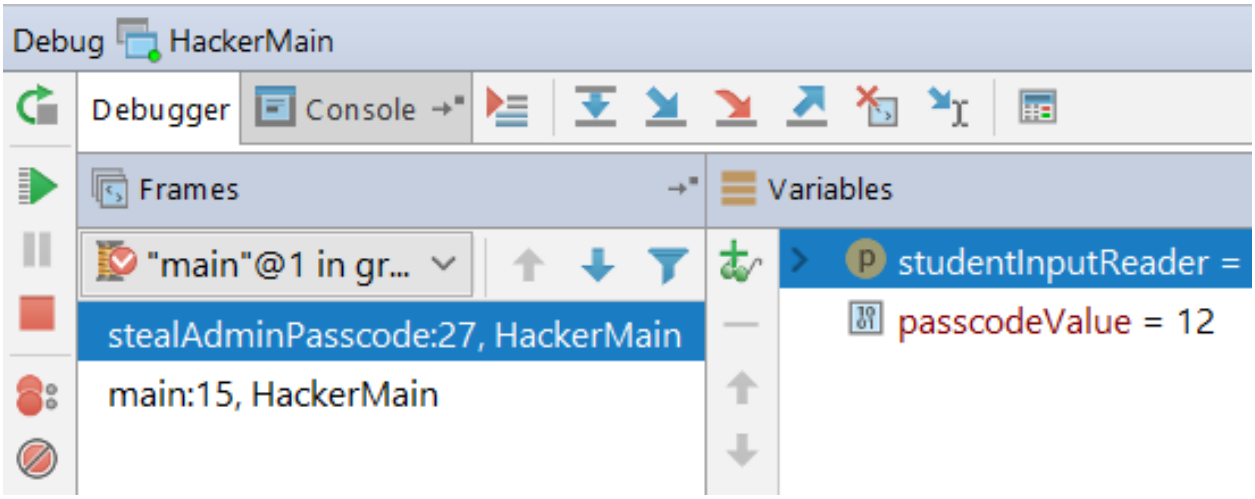

Figure 9: The debugging screen

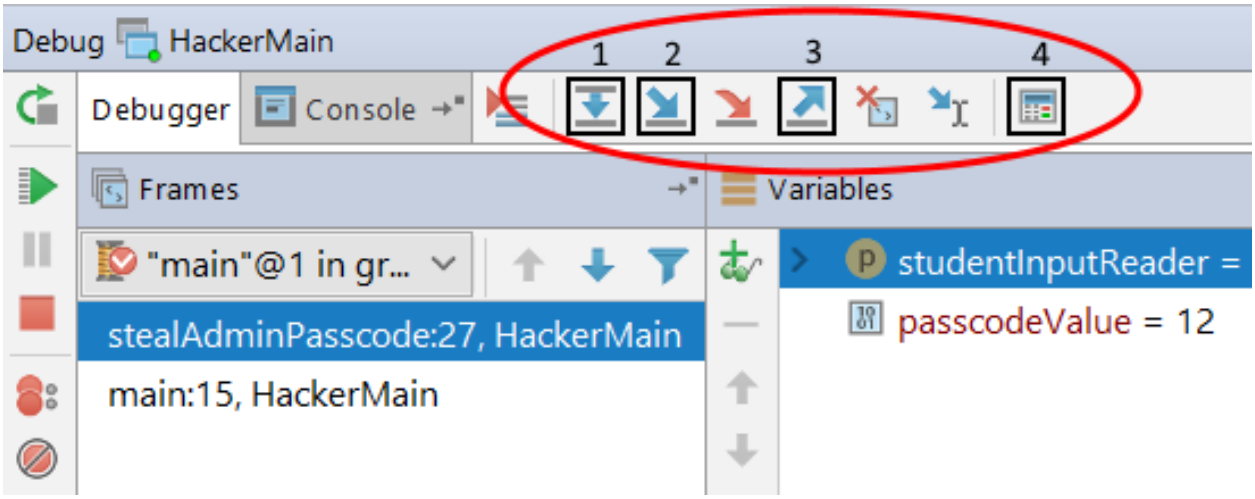

Figure 10: Useful buttons while paused

1. Step Over: Execute one more program line and pause.
2. Step Into: Similar to "Step Over". Covered in depth later in this tutorial.

3. Step Out: Execute the rest of the method and pause.
4. Evaluate Expression: Allows you to type a line of code and see its result.

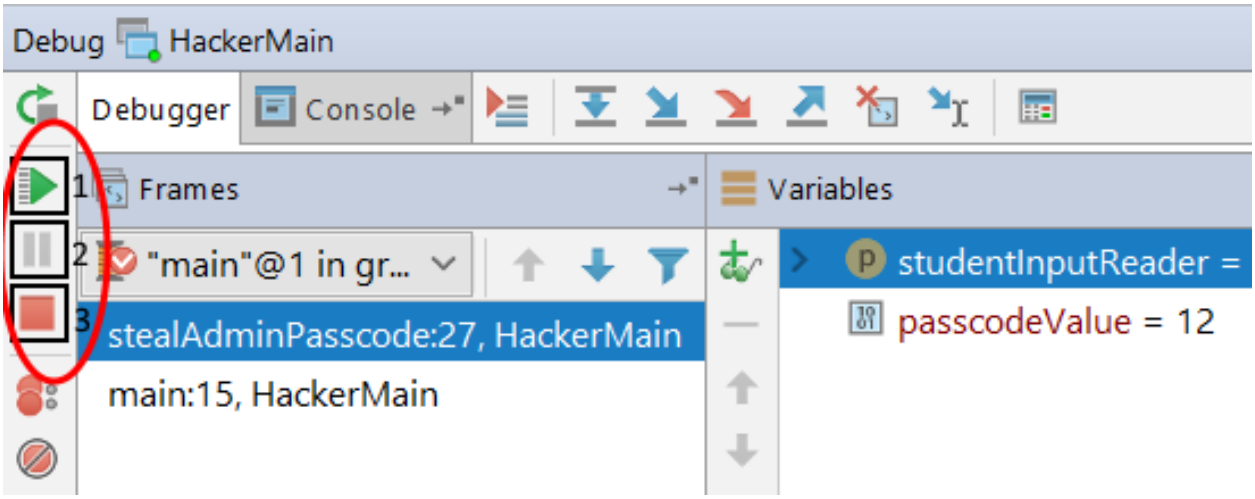

Figure 11: Program control

1. Resume: Continue executing the program until the next breakpoint.
2. Pause: Immediately pause the program without a breakpoint. Useful for finding infinite loops.
3. Stop: Immediately stop program execution.

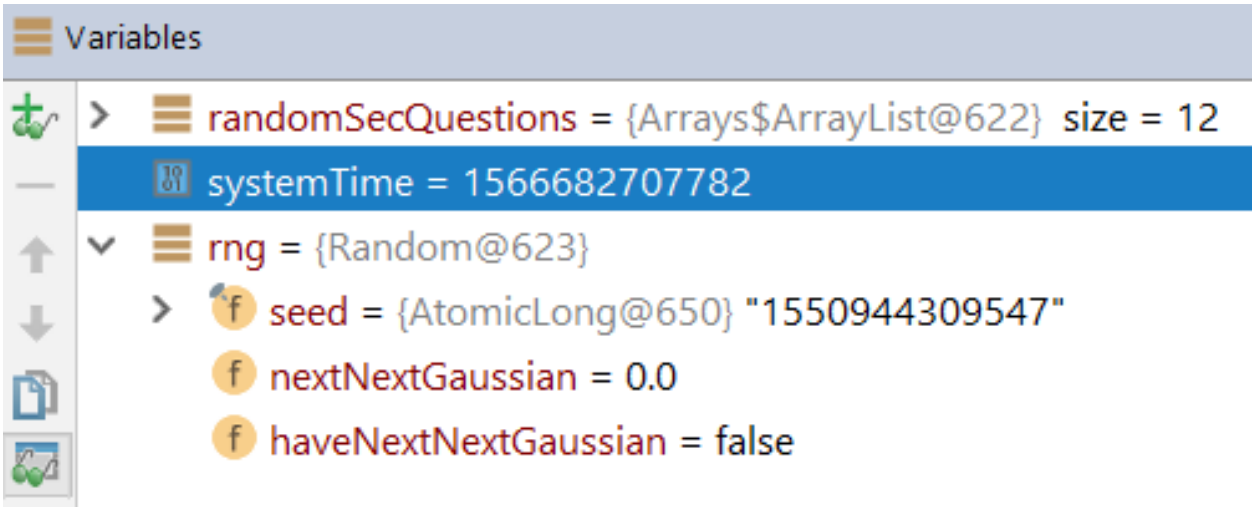

Figure 12: Viewing variables

This part of the debugging screen allows you to see the values of every variable currently in the scope of the program. Primitive types will just show you their value as you can see with `long systemTime`. For object types like `Random rng` you can click the arrow next to the variable, and it will show you the value of all fields belonging to that object.

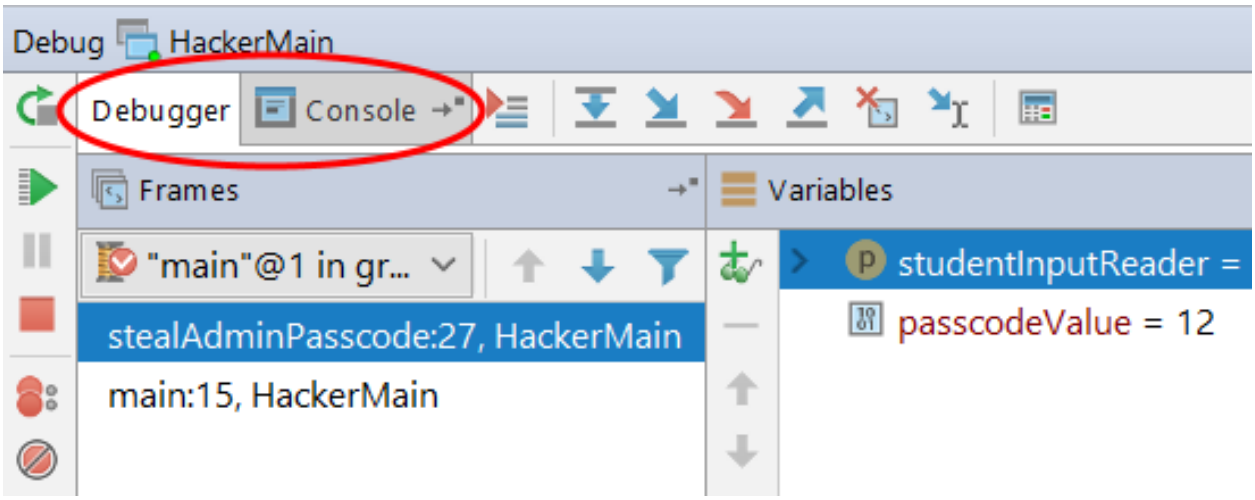

Figure 13: Switching to console

Finally, shown above is the last part of the debugging screen relevant to this assignment. You can click the "Console" button to see the standard console for viewing the result of print statements and for entering things into `System.in`. Clicking the "Debugger" button will switch you back to the debugging screen.

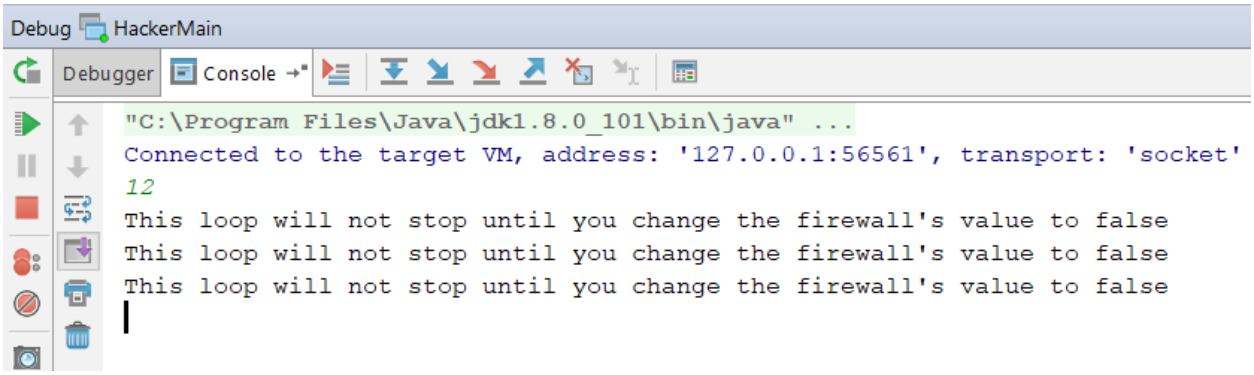

Figure 14: The console view

Now you should have everything you need to work your way through the tutorial.

4 Method 1: stealAdminPasscode()

This method teaches you how to read a variable's value while the program is currently running. There is an extremely complicated expression on the line above your breakpoint. Because your breakpoint is below this expression in the method, the expression has already been executed and its value is stored in `passcodeValue`. Trying to figure out what the expression is doing and what its value is beforehand would be quite difficult and a complete waste of time. Instead, find the value of `passcodeValue` on the debugging screen. When you have the value, type it into the console, press enter, and either hit "Step Over" or "Resume" to continue to the next method.

Seeing variable values at runtime will be an invaluable debugging tool in the future. It allows you to actually see what values are resulting in the wrong program behavior.

5 Method 2: bypassFirewall()

This method teaches you how to change a variable's value while the program is running. `bypassFirewall()` contains an infinite loop which will not stop unless the value of `boolean firewall` is set to false. Just like how you read the value of `passcodeValue` in the previous method, find the `firewall` variable on the debugging screen. When you have found this variable, right click on it, and go to "Set Value", then enter `false` as the new value.

Although modifying variable values at runtime is necessary for this assignment, keep in mind it is not generally a widely used debugging tool. This is because as a programmer you are typically interested in how the program is currently performing without being changed. This tool however can be used to force your program to go down different paths without having to change the code to do so.

6 Method 3: bypass2FactorAuthentication()

The purpose of this method is to teach the use of the "Step Into" debugging tool. "Step Into" is exactly the same as "Step Over", except when there is a method call on the line about to be executed. "Step Into" will follow the program into that method, stopping execution before the first line, where "Step Over" will entirely skip over showing the method call. The method call still happens with "Step Over", it is just not explored by the debugger.

The `bypass2FactorAuthentication()` method itself contains somewhat complicated code. Do not worry about understanding how this code works. All you need to know is that a random method from the `SecurityQuestions` class is selected and then that random method is called.

Your breakpoint should be set on the line that has the following code:
`randomSecQuestions.get(...).invoke(null)`. This line will call a random method from the `SecurityQuestions` class. Clicking "Step Into" will show you what random method is being called by pausing before executing the first line of that method.

Later versions of IntelliJ enable "Smart Step Into" by default which is not what you want here. Either disable that option, or manually select that you want to "Step Into" the `invoke()` method call and continue to "Step Into" until you reach a `SecurityQuestions` class method.

When you know what `SecurityQuestions` method has been called, remember the String return value. Use "Step Over" to leave the `SecurityQuestions` method until the line about to be executed in `bypass2FactorAuthentication()` is `rng.setSeed(systemTime)`.

When you are on this line, the `randomSecurityAnswer` variable has been declared. You need to set this value of this variable to the return value of the random method you saw. When you have done that, the same random method from before will be called again. If the random method's return value matches the value of `randomSecurityAnswer` you have completed this section.

"Step Into" is an incredibly useful tool for debugging. It allows you to easily trace through buggy code to the smallest level of detail so you can see how individual method calls are modifying the state of the program.

7 Method 4: commitCreditFraud()

What if the problem you are trying to find in your code is a rare issue? Do you have to set a breakpoint and manually trace through thousands of iterations until you spot the problem happening? Fortunately, you do not. This method serves as an introduction to conditional breakpoints: breakpoints that only stop the program's execution when a certain boolean condition is true.

Your current breakpoint should have stopped the program on the line with

`Collections.shuffle(randomBankAccounts)`. Look ahead to the loop that happens below your current breakpoint. This loop will exit the program if it finds the account with ID 126, and that account does not contain at least 1000000 money.

You should already have a breakpoint on the line with `if(currentBankAccount.getId() == 126 ... )`. Right click on that breakpoint, and a small box should pop up with a text field for "Condition". When the expression you put inside this "Condition" text field is true, only then will the breakpoint pause the program's execution. Figure out what expression you should put in this box on your own, and then press "Resume". When the program has stopped on the right condition, you should be able to modify the `BankAccount` object to have at least one million currency

Conditional breakpoints are occasionally very useful. They can be extremely helpful in the specific cases where you know a problem with the code needs a certain condition to occur, or if you are trying to see if some condition is ever true at a given point in the code.

8 Method 5: frameSomeoneElse()

This final method is designed to teach the use of the "Evaluate Expression" debugging tool. This tool can be used when the program execution is paused, and lets you see the result of executing a single expression of Java code. This expression is evaluated in the exact context of the paused program, so all variables defined in the current scope of the program may be used in the expression with the exact values these variables currently have.

Your program should be paused on the line with `int fakeAccountId = studentInputReader.nextInt()`. The random number generator variable `userAccessLog` has already been declared in the scope of the `frameSomeoneElse()` method, so you may use it in "Evaluate Expression". You need to figure out what the value returned by the next call to `userAccessLog.nextInt(241374)` will be before this exact function call is made in the if statement. To do this, click the "Evaluate Expression" tool button, and the tool window should pop up. In the text box at the top, enter the expression you want to evaluate. In this case, enter `userAccessLog.nextInt(241374)`, then click the "Evaluate" button in the bottom left corner. When you have the result of this expression, advance through the code and type it into the console to finish the last method.

You may find "Evaluate Expression" to be an occasionally useful debugging tool. It can allow you to make one-time modifications to the code at runtime if you find you have a need for that.

9 Finishing the Assignment

In order to finish the assignment you will need to push the "YourNetID.txt" and "TutorialFinished.txt" files to GitHub.

If you have never worked with git before, this section will cover basic git concepts. If you just want a walkthrough of getting your code onto GitHub and don't really care why you're doing a particular step, you can skip to the last section "Pushing to GitHub in IntelliJ"

When working with git, there are 3 things to keep in mind:

1. Your working copy: This is the code currently in IntelliJ.
2. Your local repository: This is the code git has saved on your computer. It is updated by making "commits".
3. Your remote repository: This is the code on GitHub. It is updated by "pushing" commits made to the local repository.

At this point in time "YourNetID.txt" should have a blue file name which is IntelliJ indicating that this file has been modified from the version in your local git repository. "TutorialFinished.txt" should have a red file name which is IntelliJ indicating that this file does not exist in your local git repository.

When you are the only developer working on the project, sending changes you make in your code to GitHub consists of 3 steps:

1. Add: Mark the files that you have changed to be bundled together into a commit.
2. Commit: Send the changes from all added files to your local repository. You can think of this as a save-point because you can always jump back in time to previous commits if something breaks.
3. Push: Send any new commits from your local repository to the remote repository (in this case GitHub).

When working in IntelliJ, after you have added a file for the first time IntelliJ will automatically assume you want to add the file for every commit in the future. You can uncheck the box to keep the file's changes out of a commit.

10 Pushing to GitHub in IntelliJ

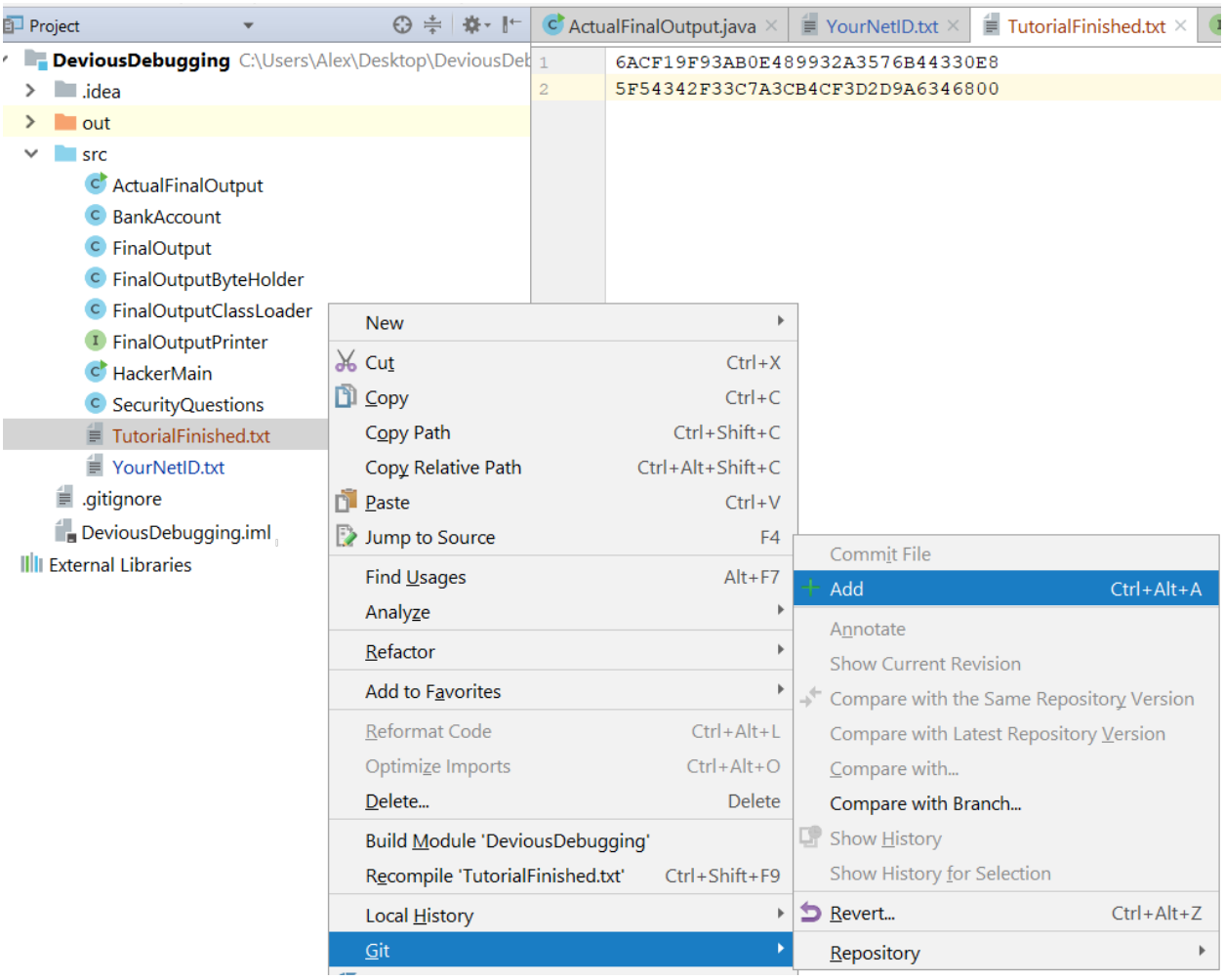

Figure 15: Add TutorialFinished.txt to git

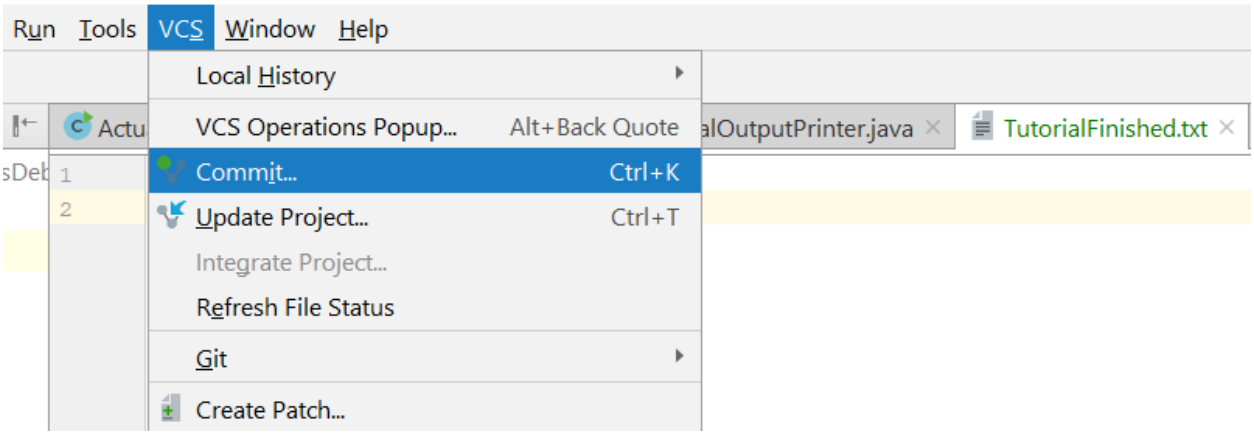

Figure 16: Go to VCS → Commit

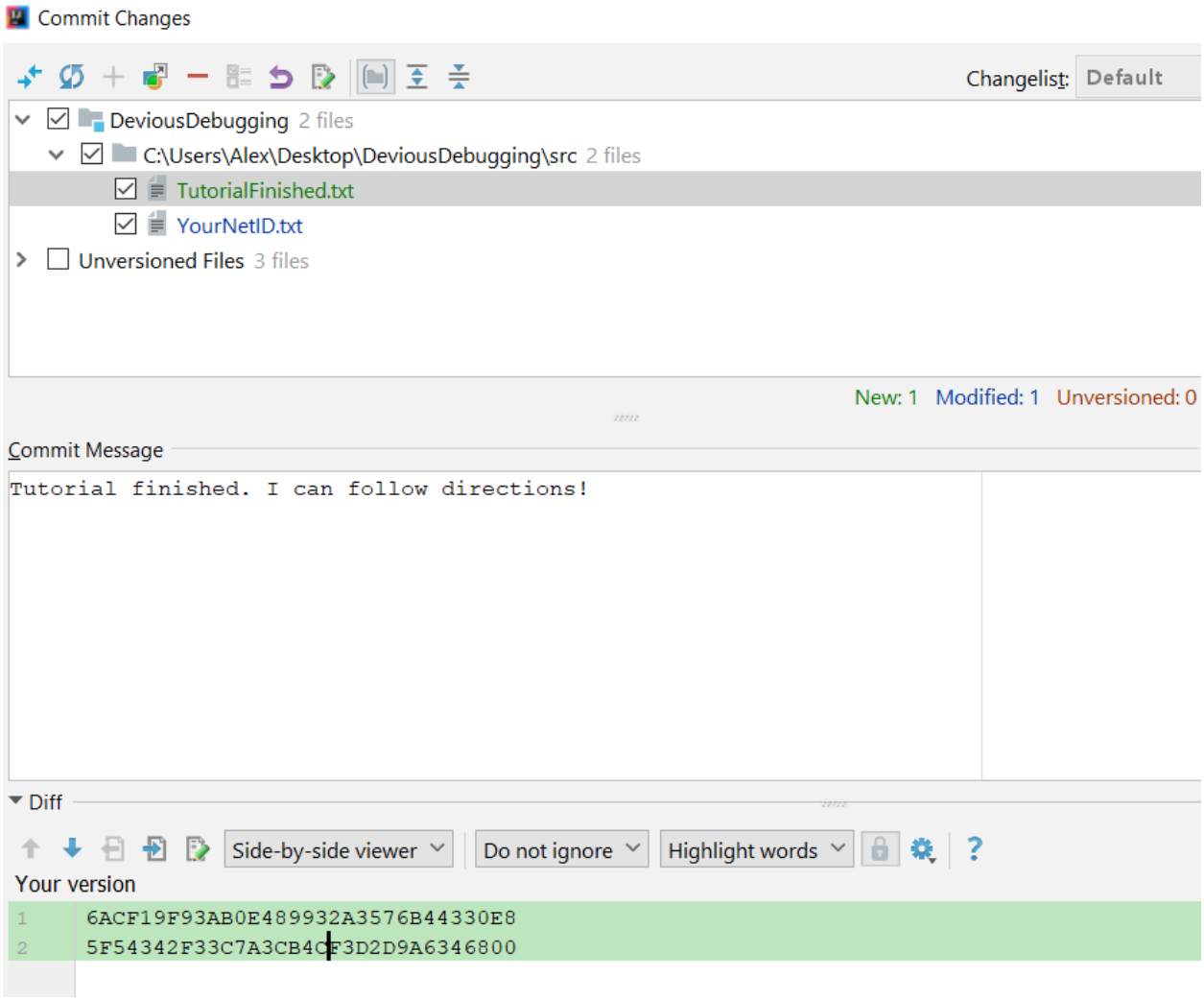

Figure 17: Both files shown should be checked to be committed

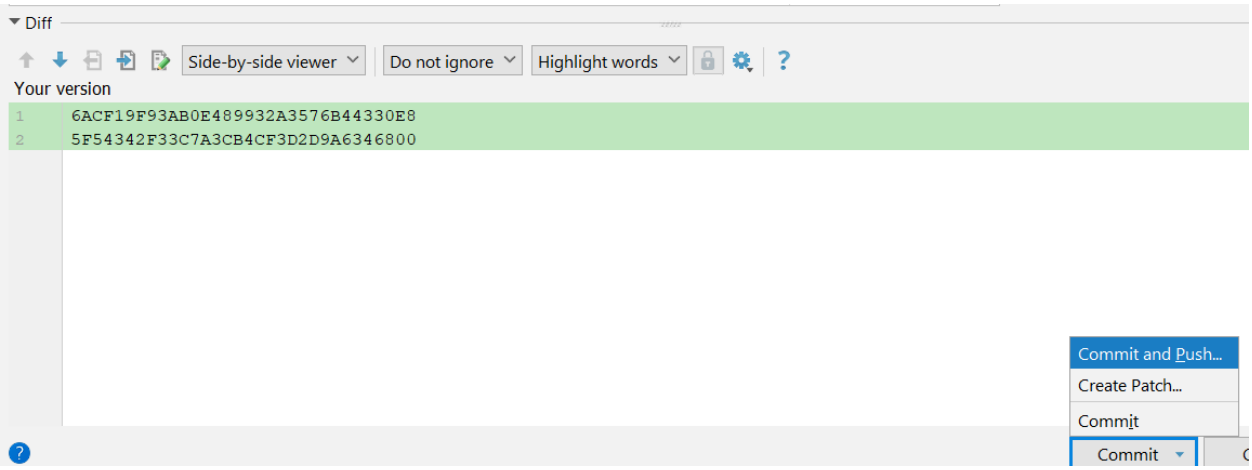

Figure 18: Commit and push in one step

Finally, double check on GitHub that the files you pushed are actually in your repository.