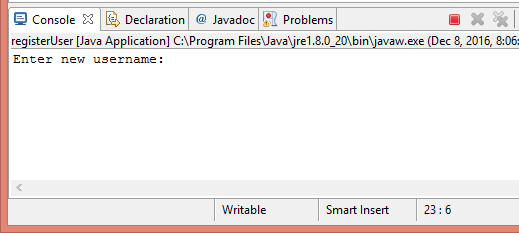
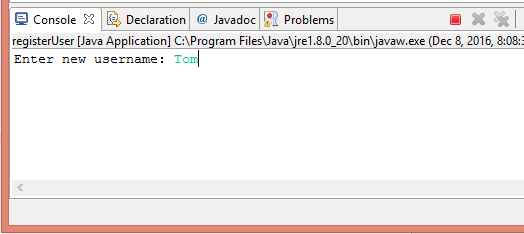
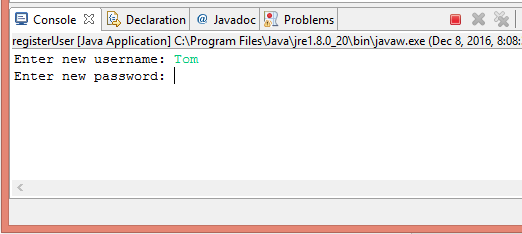
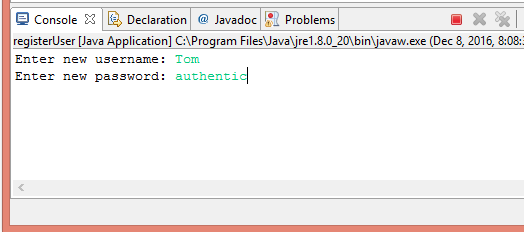
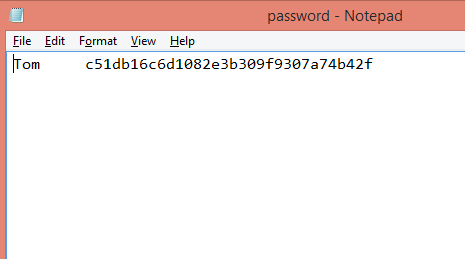
The Java class we wrote for part 1 is called, *registerUser.java*. It uses a simple FileWriter and PrintWriter objects to write to an output file called *password.txt* after reading the inputs from the user with Scanner. *registerUser.java* registers one user at a single time when it is run and writes the entry to the *password.txt* file one line at a time. When the program is ran again, it appends to the *password.txt* file onto the next line to match the output screenshot given by the prompt. Also if *password.txt* does not exist, it will create it on the directory. It will first prompt the user to enter a new username. The user types a user name and presses enter.

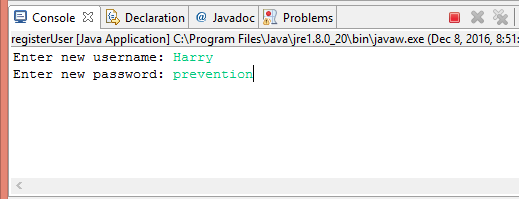
 

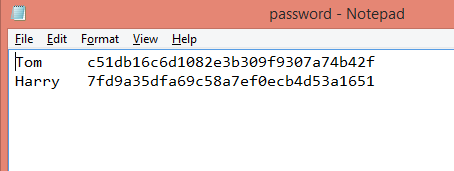
Pressing enter brings up the second input prompt to enter the password.

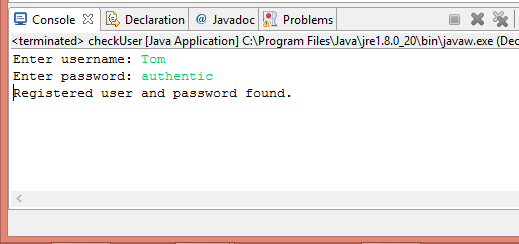
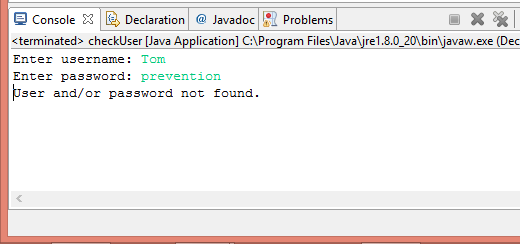
 

After pressing enter after typing the password, the program ends, and the first user is registered on *password.txt*. The given MD5 message digest takes the string input of the password and converts it to a string of the hash. The generated output on the text file is separated by a \t in the print statement between the name of the username and the hash of the password.

Restart the program and repeat the same procedure to append Harry in the text file.



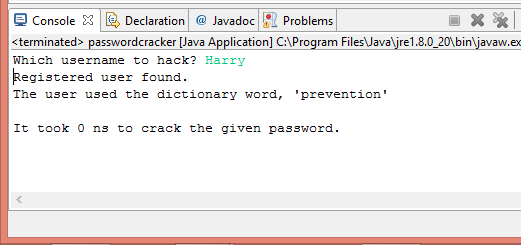
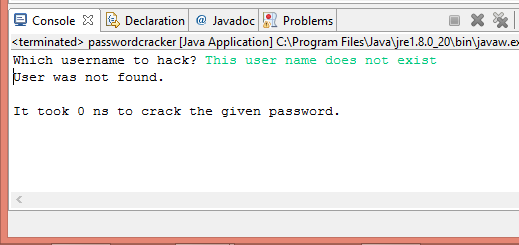
The next Java class is called *checkUser.java*. This program reads the *password.txt* file with a Scanner object line by line using a while loop. After prompting the user the username and password with another Scanner object, the two strings are stored in the variables, ‘username’ and ‘password’. These will be compared to the line being read by the Scanner for *password.txt*. The program takes the entire line and puts the line into a List of strings inside the while loop to read the whole file line by line. We chose to use a list to dynamically add each line into the List for each index. Note that *password.txt* contains information for a single user per line. The if statement checks if the variables username and password is contained within the list in the exact format of the line being read. Because the password is converted to hash in the text file, the MD5 message digest is needed once again to take the password input, and convert it to find a match. When true, the program will output that both the user and password was found. Otherwise, if either the username or the password does not match up, it will print the failure message.

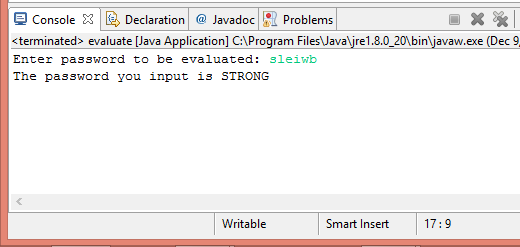
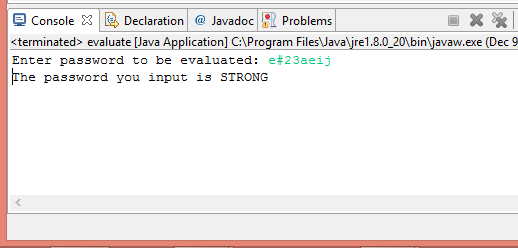
Part 2’s uses the program, *passwordcracker.java*. It takes the given *dictionary.txt* file and the *password.txt* file and reads them with separate Scanner objects.

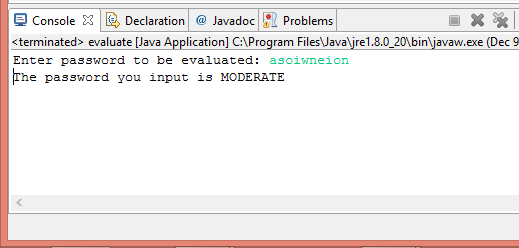
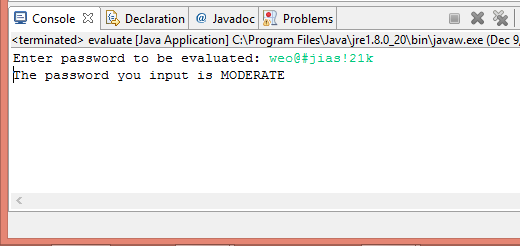
For type 1 passwords, We decided to use the Java Map object to store and add dynamically the words in *dictionary.txt* as it gets read line by line with a while loop, as well as the registered users in *password.txt* in key and value format. All the words being read line by line from *dictionary.txt* are stored as the key for the Map called ‘dictionary’. These words are also fed into the MD5 function to get the hashed string for each word and stored as the value for the Map ‘dictionary’. When *password.txt* gets read line by line, we decided to split the line into two tokens by using \t separator. The first token gets put into index 0 of a string Array called ‘tokens’ and the second token into index 1. Then we put the index 0 as the key for the other Map called ‘registered’ and the index 1 as the value for ‘registered’. The ‘dictionary’ Map has {key:[dictionary word], value:[hash of the dictionary word]}. The ‘registered’ Map has {key:[username], value:[hash of the password]}. Once our Maps are established, the program finally asks for the username as input.

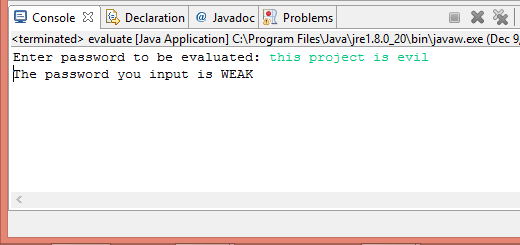
Maps are great because it has a function called .containsKey and .containsValue. After the user submits the username input, if the username is found as one of the key in ‘dictionary’, the if statement will be true and we can print out that the user was found. Our logic was to match the value of ‘dictionary’ to the value of ‘registered’ because they are both the hash. We wrote a neat little method called getKeyFromValue(). After comparing all of the values for ‘dictionary’ and ‘registered’, the program gives the key of ‘dictionary’ corresponding to the value containing the hash using the method we wrote.

If there was no match found, the program prints out that the user did not use a dictionary word. If the username was not found to begin with in ‘registered’ the program will print out that the user was not found.

For part 3, the class we wrote is *evaluate.java*. *dictionary.txt* is read by a Scanner object and we use another Scanner for the user input. The words from the text file are put into an ArrayList because ArrayLists’ elements can be dynamically added (compared to regular Arrays that you need to declare a fixed size). Just in case, we also converted every word being read to lowercase for comparing later. The user input is stored into a string variable called, ‘password’ which is also converted to lowercase, then fed through a method we created called strength(). strength() take the ArrayList called ‘words’ and the password string as parameters. The method iterates through ‘words’ using a for loop to compare each word with the password string. If the password is found inside ‘words’ entirely, then it is a weak type 1 password. If the dictionary word is a substring of the password, it is a moderate strength password. And else, it is a strong password.

  IMPORTANT NOTE: For the longest time while making this program, for some reason we would not get it was a strong password when we input a string such as ‘asoiwneion’. This password string seemingly looks like a strong password, because ‘asoiwneion’ is NOT and should NOT be dictionary word. Yet, the output we received for our keyboard mash inputs seemingly gave us the result that these were moderate strength passwords.

We thought something was wrong with the logic of the if statements or the for loop was wrong. When we finally wrote out a print statement inside our logic statements to debug, we found out something that made us waste an insane number of hours pondering. There were TWO letter words in *dictionary.txt*. So in the example of seemingly strong but moderate strength passwords above, ‘asoiwneion’ and ‘weo@#jias!21k’, the substring that was being read was actually the dictionary word, ‘as’. Our program had been working the whole time, and we just wasted lots of hours and why it seemed to not work during our project demo on Monday. Lastly, a weak password as an example to the right.