**Weather Data Project: Design Document**

***General Purpose and Design:*** This Weather Data program calculates the wind chill factor and cloud base altitude based on the user inputted values for the temperature (in degrees Fahrenheit), the wind speed (in miles per hour or MPH), and the dew point (in degrees Fahrenheit). The program utilizes GUIs and controls when the user inputs their data; they are used to display the data on a main interface and in a separate window, and to plot the temperature and wind chill values on a graph. The program also has a secure login system where users create a username and password to login and get access to the main interface of the program. There is input validation, exception handling, and error dialog message boxes within the program to ensure the high quality of the program and to prevent the user from encountering bugs or unintended problems in the program. All controls, data, text input, and displays are designed to be pleasant on the eyes, user-friendly, and clean.

**Milestone 1:**

The design of this Weather Data program started with the creation and development of the create account-login startup interface. The goal with this first milestone was to create a simple, easy-to-use, and clean create account/login/cancel dialog box that connects the user to the separate and specific create account and login windows/GUIs. A screenshot of the finished startup window is shown below:

Screen Capture of the Final Startup GUI:

Graphical user interface, text, application

Description automatically generated

*Section 1: Creating the Initial Dialog Box*

To begin, I created the “WeatherData.java” class to be the main body and operator of this project. Inside the “main” method in this class, I began building the frame by declaring the three options (“Cancel”, “Create Account”, and “Login”) that the user can choose from. Then, I set up a variable, named “selection” to carry an integer. This “selection” variable is initialized to the JOptionPane for the dialog box, which includes all of the components that make up the dialog box itself like the standard formatting for the window, the message inside the box, the title on the boarder of the window, the option and message types, the icon inside the window, and which of the options will be the button centered on the bottom of the dialog box. Depending on which option or button the user clicks on in the dialog box, the variable will be assigned an integer that represents that option. This implementation is important because of what follows it, which is an if-else if block that looks at the integer that is in the variable “selection” and flows through to the block of code corresponding to that integer value. This is the main flow of control point in the program. It is the part that determines if the “create account” window should appear, if the “login” window should appear, or if the program should end. The next section looks at this part of the program in detail. But at this point, when the program runs, the dialog box appears with all of the intended formatting, options, and text. It is clean and operates as I designed and intended.

Screen Capture of the Startup Code:

Text

Description automatically generated

*Section 2: The Flow of Control*

Following the creation of the dialog box, I turned my attention to setting up the flow of control for the program. Here, as I mentioned before, I implemented an “if-else if” statement to create three different branches that the program can go in. The first path is for if the user selects to create an account (making the integer assigned to the “selection” variable a 1). This branch then prints “Creating Account” to the console, and then creates a new instance of the “WeatherCreateAccount” class (discussed later), called “createAccWin”. At this point, the “Create Account” window would appear, and the user would go through the process of creating an account. Once successfully completed, I plan to have a new dialog box appear that asks the user if they want to go back to the initial startup interface or if they want to go directly to the login screen. This dialog box is obviously not created yet (as it is inherently tied to the code and work for Milestone 2), but it is one of my largest goals for the program going forward as I want to provide the user as much convenience as possible. In this case, I would be providing the users a way to go from account creation directly to the login screen if they so choose to take that route, but they are not forced back to the “main” menu, nor are they forced to the login screen either. I want users to be able to choose. The second path is very similar to the first path. The second branch is run through if the user selects to go straight to logging in (making the integer assigned to the “selection” variable a 2). The console simply displays “Login”, and then an instance of the “WeatherLogin” class is created, named “loginWin”. Like with “Create Account”, the “Login” window would be displayed, and the user would go to login. If successful, then the main GUI of the program would appear (not created yet!). Lastly, there is the last path the flow can go. If the user selects the “Cancel” button, then the console displays “Cancel” and the program ends.

Screen Capture of the “if-else if” Code Block:

Text

Description automatically generated

*Section 3: Building the “WeatherCreateAccount” and “WeatherLogin” Classes and Windows*

Next, two new classes were created to contain constructors, components, controls, frames, and action and event listeners for the “create account” and “login” windows respectively. When the user clicks on either the “create account” or “login” buttons on the main startup interface, the corresponding window is created and appears on screen.

Screen Capture of the “Create Account” Window:

Graphical user interface, text, application, email

Description automatically generated

Screen Capture of the “Login” Window:

Graphical user interface, application

Description automatically generated

At this point though, both the “create account” and “login” windows have been designed and created, but they both lack any functionality.

Both the “WeatherCreateAccount” and “WeatherLogin” classes, as well as their windows, are extremely similar in design and structure. This was not only to keep the code neat and simple but also to keep their windows in line with the rest of the program’s design, providing simplicity, comfortability, and familiarity for the user. Each class begins with importing some java.awt classes, specifically ones relating to GridBagLayout functionality. Then, the GUI frame, panel, and the Username and Password labels are created. The key difference between the two classes is that “WeatherCreateAccount” has multiple password labels so that the user can clearly know what their password must have in order to be valid. The “WeatherLogin” class has no need for that information, so it does not have multiple password labels. Next, the classes create the Username and Password textboxes, and the “confirmation” and “cancel” buttons. The “confirmation” button is the button in the window that will make the program proceed. So, in “WeatherCreateAccount”, it is a “Create Account” button, and in “WeatherLogin”, it is a “Login” button. Lastly, there are the constructors themselves, and like the rest of the classes, their code is extremely similar. To begin, each constructor sets the size of their respective window (500 width, 350 height for the create account window; 400 width, 250 height for the login window), declares a new instance of GridBagConstraints(), and sets the default padding insets for the window. A majority of the rest of the classes’ constructors consist of setting the column and row constraints and the alignment of the username label and textbox, setting the column and row constraints, the insets, and the alignment of the password label(s) and textbox, and setting the column and row constraints, the insets, and alignment of the “confirmation” (Create Account or Login) and “cancel” buttons. The last things that the constructors do, for now, are that they set an etched title border for the panel and then they add the panel to the GUI, set resizing rules (false – make users unable to resize the windows), set the initial location of the windows to be the center of the screen, and making it so the GUIs are visible. At the end of the constructor is a setDefaultCloseOperation line that is likely temporary as I would like to develop a more “controlled” way to handle the closing of the windows. But at this point, when the program runs and I select either of the create account or login buttons on the initial dialog box screen, the create account window and login window appear when their respective button is clicked with all of the intended formatting, options, and text. It is clean and operates as I designed and intended.

As I said before though, these windows and classes are currently incomplete. There are many functionalities that I am looking to add and develop into the program in the coming weeks including (but not limited to):

* Adding the action event listeners to make the “confirmation” and “cancel” buttons have their intended functionality.
* Finish the account creation and login functionality with input validation and error messages/error dialog boxes if invalid information is entered.
* Create “Save Data” functionality to store valid usernames and passwords in a file to be retrieved by the login function.
* Start building the main GUI interface of the program that displays after the user successfully logins to the program.
* Create the dialog box for after the user successfully creates an account to allow the user to choose if they want to go back to the startup interface or if they want to go directly to the login screen.
* Code to better handle the program and windows if the user clicks on the “X” or “close” button.

**Milestone 2:**

With Milestone 2 here, I began by designing and implementing the intended functionalities for the “Create Account” window. What was required here in order for the “WeatherCreateAccount” class and the “Create Account” window to be considered “complete” to me is that it needed: the two buttons (“Create Account” and “Cancel”) to be functional (and function as intended), to be able to take in input from the user for the username and password and input validate them, to store valid account information into a file, loop back and ask the user again if invalid inputs are given, have error messages and dialog boxes appear for both successful account creation and for invalid information entered, have a dialog box to ask the user if they want to go back to the startup screen of the program or if they want to go straight to the login screen.

To start, I had to import the java.awt.event package in order for the action listeners to work. Then, I created two classes that invoke action listeners to look for if either of the two buttons are clicked, and then use that to give them their intended functionalities. So, one class is listening for if the “create account” button is clicked and the other is listening for if the “cancel” button is clicked. If “cancel” is clicked, I want the program to dispose of the “create account” window and go back to the startup screen. If “create account” is clicked, then I want the program to check the inputs to see if they are valid; if so, then have the dialog box appear saying that account creation was successful, and then have a second box appear that asks the user what they want to do next: go back to the startup screen, or go directly to the login screen; if not, then I want an error message dialog box (or multiple of them) saying what was exactly was wrong with the credentials inputted to appear and then have the account creation prompts loop to allow the user a chance to enter valid information.

Final “Create Account” Window + Success and Error Message Dialog Boxes: Graphical user interface, text, application

Description automatically generatedGraphical user interface

Description automatically generated

Graphical user interface, text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generatedWhile it was not too difficult to actually write the code for these two classes, there was one problem that came up that also took a lot of thinking to problem solve around. I was stuck for a while trying to figure out how to call the “main” method of the “WeatherData.java” file from the “WeatherCreateAccount.java” file and class here. So much so that I conducted some research on the Internet to help myself figure it out, and the solution here was to have a small method within the class that would be called by the code within the “actionPerformed” method of the class. To reiterate, this method of the class simply outputs a message to the console indicating that the cancel button was clicked. Next, it disposes of the create account window, and then it calls the “callWeatherDataMain” method that is within the class and that method calls “WeatherData.main()”, bringing up the “main startup menu” of the program. Once I developed and implemented these changes to the class and it worked, the “cancel” button’s action listener class was complete.

The Code for the CancelButtonListener Class:

Graphical user interface, text, application

Description automatically generated

As for the “create account” action listener class, that class did not have any problems when it came to developing the class and figuring out the flow of control within it. For example, at the start of it (so when the button is clicked), it first prints a message to the console and then gets the username and password that the user entered from their respective text fields. However, one issue that I did run into with this here was that I initially had the text fields have default text inside of it. So, “Username” and “Password” used to be the text that would always be in the text fields when the “create account” window was opened. However, when the button listener class next called the “verifyAcc” (detailed in *Section 1* below) method, a problem would occur where if the user did not type anything, as in left the default text in the boxes, the method would take that text in and actually indicate a successful account creation. Thus, to solve the problem, I simply removed the default text and had the text field be blank or empty by default. I believe that the instructions are clear enough and spaced out correctly to where the user should not have any confusion over which text field is for the username or the password. So, there should not be any problems on that front.

Changes to Text Box Declarations:

Text

Description automatically generated

But back to the action listener class for the create account button, it contains the call to the “verify account” method, and it has an if-else block to conduct flow of control for if the “verify account” method is true or false. If this method returns true, then the account creation is successful and the “directive” dialog box will appear for the user (designed and called in the same way as the WeatherData.main() method is in the “cancel” button’s action listener class), but if not and the method returns false, then an error message will appear stating that the username and/or password is invalid and to please enter valid information.

Error Message for Invalid Username/Password:

Graphical user interface, application

Description automatically generated

The directive dialog box is a very simple, two button, small dialog box that asks the user where they want to go next: back to the “home” window, or straight to the “login” window. To make things simple and easier to build for this dialog box, I had the “create account” button listener class invoke a method that contains the dialog box’s JOptionPane set up and options and the flow of control for the user’s selection. I basically modularized it to make the flow of control easier to follow.

Directive Dialog Box and Its Method:

Graphical user interface, text, application, email

Description automatically generated

Text, letter

Description automatically generated

There were no other issues with building the rest of these action listener classes. Following that came building the “verify account” method, which was a major doozy at times. In the following section, I will detail the creation of this monster method and the issues that I had, but fixed, while building the rest of the create account algorithm.

*Section 1: Building the “verifyAcc” Method*

First off, in order for the functionality of this method to work, I had to import multiple java packages. The two packages were: java.io and java.util. This was because I was utilizing file operations and the Scanner class, which wouldn’t work without them. Then, I declared all of my Boolean variables. There are 7 in total: goodPass, uppercase, lowercase, digits, longEnough, goodUser, and goodAccount. With variable names like these, it is pretty easy to understand what their purpose is or what they represent. For example, longEnough is a Boolean variable that tracks if the password is greater than or equal to 9 characters in length. If the variable holds true, then the password has been found to be long enough to satisfy the requirements. There are in-line comments in the source code that describe what each variable represents and tracks.

Boolean Variables in the “verifyAcc” Method:

Text

Description automatically generated with low confidence

Following the variables is the actual algorithm for verifying the potential created account. This was where things got difficult and complex. The first thing I did was develop the code that checks all aspects of the password that the user enters. Initially, this consisted of a variable declaration to get the length of the password, a small if statement to check if it is long enough (if so then it assigns “true” to the longEnough variable), and a for loop that gets the character at each index of the password and goes through if statements to check for if it is a digit, an uppercase letter, or a lowercase letter. As long as one of each is in the password, then each if statement will be run through at least once and that switches the “false” in each of their respective Boolean variables to “true”, indicating that there is indeed that type of character or digit within the password string. Lastly, there is one last if statement that checks if every one of the variables that relate to the password equate to “true’, and if so, then the “goodPass” variable will be assigned “true”, meaning that the given password satisfies all of the requirements. However, this changed as I went along. Due to the aforementioned issue with the text field containing the default text or just being blank, there were issues that would arise if the user just left it in or left it blank, so after a while and a good amount of trial and error, I remembered that I could just use a simple if-else statement block and put all of the code that I just described for the password algorithm into the “else” part of the block and then have the “if” part check if the text field is blank. If it is, then I just had an error message dialog box appear saying to the user that the password text field was left blank and to please enter a valid password. So, that is what I did to fully solve that problem (this would also be an issue with verifying the username, too later on). At last, the password part of the verify account algorithm was complete. Next came the absolute hardest part of developing this area of the project: the username verification algorithm.

Password Verification Code:

Graphical user interface, text, application, email

Description automatically generated

*Section 1, Subsection 1: The Username Verification Algorithm*

After the password algorithm, I began work on the username verification algorithm. There were two major requirements of this part of the algorithm: 1.) I must check the account information file for if there are any usernames of already created accounts that are the same as the inputted username from the user. (As well as file handling and exceptions for if the file is missing or fails to open) 2.) If the username and password are both found to be valid, then I must write the account information to the file (and have exception handling for a possible input/output error as well). With the experience I had with the password algorithm, I started by setting up the username algorithm in the same manner, with an if-else statement block. The “if” part handles if the username text field was left blank. It works in the same way as it does with the password, sending an error message out if true. The “else” part contains the rest of the algorithm, which starts with declaring a new file to store valid account information in (if the file already exists in the folder, then it just assigns it to the accountFile variable).

Top of Username Verification Algorithm:

Graphical user interface, text, application, email

Description automatically generated

Then it tries to declare a new Scanner for the file. This is where the first try-catch block is. If the Scanner cannot be declared, then it means that the file cannot be found or it cannot be opened, so there is a catch block at the very end of the code here that catches the potential FileNotFoundException. So, I got one of the exception handlers done.

If the “try” is successful, then the flow of control goes into that block of the code, where it then meets another fork in the flow of control road. There is another if-else statement. The reason why this one is here is to solve the largest problem I had with developing this project, so far. Many times, during testing, I would have problems where there is nothing in the file, but the username would not be accepted (the password would be valid though) and then the code would fall through and give me the error message from the “create account” button action listener where it says that that username and/or password is invalid. However, this should not happen as long as the password is valid because the file is empty so there are no accounts on file meaning that as long as the password is valid, then the account should be created. After using the debugger to follow the flow of control line by line and block by block. I found that in the while loop that I had, the “in.hasNext()” method was returning “false”. So, that means that if the file was blank, then the while loop would never run and thus no accounts would be created. At this time, the code for the username algorithm was all under this while loop and all it had was that “in.hasNext()” call. It was simply looking for if there was something in the file. I did not realize initially that if the file was empty, then the method would return false. So, what I did was split, with another if-else block, the code between if there are no accounts in the file (which is checked using accountFile.length() == 0), and if there are accounts in the file.

Middle Portion of the Username Algorithm:

Graphical user interface, text, application

Description automatically generated

So, when no accounts are on file, then any username is available and as long as the password is valid, then the account is valid. Thus, in this part of the block, a system message is printed to the console and the “goodUser” variable is set to true. Then another if statement checks for if both the “goodUser” and the “goodPass” variables equal true, indicating that both the username and password are valid. If so, then the variable “goodAccount” (which the method returns back to the create account button action listener at the very bottom and end of the method) is set to true, meaning that the account is fully valid! Then the information tries to be written to the file. There is an exception in the catch block following it that will catch an IOException if that error occurs when trying to write the account information to the file. After that, the value of “goodAccount” is returned as I said before. But if the password is not valid, then the flow would never even go into that if statement and it would just fall through the rest of the code, returning “false” back. However, if the “goodAccount” variable is true, then the account is successfully created and a dialog box indicating so will appear.

But that is how the code works if there are no accounts on file. In the “else” part of the if statement contains the code that runs when there are accounts on file. Inside this else block, is the aforementioned while loop. I modified it so that both the in.hasNext() method and the goodUser variable must return true in order for the while loop to run. I did it this way so that it guarantees that the while loop will always run (because there will always be at least one line to read as this block only runs if there are accounts on file, and the goodUser variable is set to true by default) but also to solve an issue that I had at times when there were more than one account on file. This issue was that the while loop would read the usernames, and then find an identical one and do all of the code involved with that if statement, but then the while loop would keep going through the rest of the file and what would happen is that when it read another account’s information, the usernames would not match, thereby setting the goodUser variable to true again when it was false due to the identical username read earlier. At the end, the variable would be true despite an identical username being found on file and say that the inputted username was available, which should not be true. So, my fix to this was having the goodUser variable as part of the conditions for the while loop to run. That way, if the while loop finds an identical username, then goodUser would be set to false, thereby ending the looping of the while loop and moving the flow of control on through the rest of the program. This prevented and solved this problem. Now, inside the while loop, the text in the file, up to the whitespace, is read and put into a temporary String variable named “text”. Then, the code checks for if the read text is the same as the inputted username by the user, and if so, then an error message dialog box appears saying that the username is already in use and to please choose a different one. So, this is the input validation that prevents repeat usernames from being used; every username must be unique per the instructions and this achieves that goal.

Then, the while loop repeats until there is no more lines to read. But because the in.next() method reads up to whitespace, this means that the passwords in the file would also be read. As such, it is also compared to the inputted username, but because they would not (and should not) be equal, it would just fall through the loop and the loop would then repeat, grabbing the next username if there are more lines in the file. But I did add an else-if block to account for if the inputted password is the same as a read password from the file. If they are the same, I made it so a caution message would appear saying that an identical password was detected in the file and that it is recommended that the user changes it. This message could technically be printed to the console multiple times depending on how many passwords in the file are identical to the inputted password. I do not consider this an “error” because I think it would be very important and useful for the user to see just how many passwords in the file are identical to theirs. Once the while loop ends, the scanner is closed. Then, goodUser is checked to see if it is still true after going through the while loop, if so, the username is available message will be printed to the console. Then, the code is the same as it is in the “if” part of this giant if-else block where if both of the variables, goodUser and goodPass, are true, then the account is said to be valid and the information is written to the file, checking for the IOException of course. Finally, the value of the goodAccount variable is returned.

Rest of the Username Algorithm and the verifyAcc Method:

Graphical user interface, text, application

Description automatically generated

That covers the entirety of the “verifyAcc” method and the entirety of the code added or changed in the “WeatherCreateAccount.java” file and the “WeatherCreateAccount” class. In the next section, I will discuss the Login algorithm and the changes made to the “WeatherLogin.java” file and the “WeatherLogin” class.

*Section 2: Building the Login Algorithm*

Overall, building the login algorithm was WAY faster than the create account one. Obviously, this is due to much of the algorithm already being built/developed in the WeatherCreateAccount.java file already and because there is not as much input validation and checks in the login process. As such, a majority of the login algorithm is copy and pasted from the create account algorithm and then slightly altered or has aspects removed. For example, both button’s action or event listeners as well as the part where the while loop is were copied and transferred over to this class. However, some changes were made of course. In the login button’s action listener, the only thing that majorly changed was that the method and method call that creates the dialog box after an account was successfully created was removed as it is not needed in this class. Much of the rest of the class is the same. As for the cancel button’s class in this file, it was 100% copied and pasted from the create account’s version of the class. The only difference between the two are names; that’s it. With the verify account method, many of the Boolean variables were removed as they were not necessary in this class, checking all aspects of the password were removed, making sure a username was unique was removed, checking for if there are accounts on file or not is not checked, and there is no writing to a file, among many other things that are in the create account version of the verify account method but are removed or not there in the login version of the method. Really, the only things that both versions of the method share are the if statements checking for blank text fields, scanning the file and seeing if the text is the same as the inputted username and password, and utilizing exception handling for file operations. The flow of control and the operations of the WeatherLogin class’s code are practically identical to WeatherCreateAccount class’s code, so it may be redundant to repeat myself. All in all, when a successful login happens, a dialog box stating that the login was successful will appear, then the login window will go away, and finally, at last, the main GUI of this Weather Program will appear. So, in the next section, I will go over the design and construction of the main GUI interface of the project.

Final Login Window + Successful and Failed Login Dialog Boxes: Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface, text, application

Description automatically generated

*Section 3: Building the “WeatherGUI” Class and the Main Interface*

Following the completion of the functionality for the login window, I began to build the main interface’s components and window. Everything related to the main interface’s design is within its own, new class called “WeatherGUI”. The main interface’s design is similar to that of the other windows in the program. Like the “create account” and “login” windows, the main interface here utilizes the GridBagConstraints and GridBagLayout classes to build its window. So, everything is inset by 1 on all sides, components and text are either centered, aligned to the left, or aligned to the right within their column, and the border layout is the same as seen in the other windows. As a result of this design decision, it was easier to create this window as it generally followed the same code structure as in the “create account” and “login” windows, so I was initially able to copy and paste a good amount of the code over, and then change it based on how I named the labels, text box fields, and buttons. However, there is one major difference between the other windows and this window. That difference being that the main interface here has many image files that will be integrated into the window. My idea here was to have two images appear at the top of the window to serve as good informative information for the user. I believe this will work out great as not only does it make the window look a bit distinct from the other windows, but it utilizes the two-column structure of the window to provide more information for the user, which is exactly what I wanted to do. These two images that will always appear when the main interface appears are a picture showing wind chill temperatures and the times it will take frostbite to set in, and a picture showing the different cloud types and the relative height zones that they appear, some of which stretch multiple height layers. The former picture appears in the left column, and the other one in the right column. To implement these images properly, I invoked two try-catch blocks that each contain the code related to that image. So, inside the try blocks, there are the code that determines the row and column constraints for the image, the image alignment code, and the code that actually makes the image file a usable image object that then is converted into a JLabel and then is added to the WeatherPanel. But, if the image does not exists or can’t be read, then an exception would be thrown. Thus, two catch blocks follow the try blocks and they each catch the exception, if thrown, and they set a Boolean variable to false (so that the interface will not appear if *any* of the images are not found), an error message dialog box appears notifying the user that an image file is not found, and then a message and the stack trace is printed to the console. Ultimately, these two blocks of code check to see if all of the images that are necessary for the main interface to function exist in the program folder. If at least one of the images cannot be found, then the main interface will not appear, and the program will end.

Try-Catch Blocks for Images and Error Messages:

Graphical user interface, text, application

Description automatically generatedGraphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

After checking if all of the image files necessary for the window’s functionality are there, if they are all indeed there, then the rest of the window’s components are built. The code here is the same simple set up as seen in the code for the “create account” and “login” windows, where each component, each label, each text box, each image, each button, is set column constraints, row constraints, insets specific to that component, an alignment within their column, and then is added to the panel. But unlike the other windows, most of the components here are aligned to the center instead of the left or right of their column. The reason for this is due to the large size of the pictures and window itself, and the labels and text field boxes being small compared to it. So, to make everything look better, I centered these components. But then, a problem occurred when it came to the layout of the buttons. In this window, I have three buttons: a “compute” button, a “show file data” button, and a “logout” button. The functionality of these buttons, as well as the whole window, will be done in the next milestone, so at the current moment, the buttons and the whole window itself are not functional. All that happens here is that the window and its components appear. But back to the point, I was having trouble deciding where I wanted these buttons to go. That is because, due to the GridBagLayout style, I cannot, as far as I know, have the “show file data” button stretch over the right end of the first column and the left end of the second column so that the button lays comfortably in the center of the window. This is not possible as far as I know. So, I had to get creative. I thought of just having the “compute” and “show file data” in the second column aligned to the right and left ends of the column respectively. But then I realized that when data is entered and the compute button is clicked, that when the results show, that column may look pretty inappropriate, and I wanted to avoid that. So, what I decided was this: I created a new label, named “cloudTypeLabel”, that will be a part of an added function of the window and program, which is that whenever the cloud base is calculated, the level that the cloud would be at based on the cloud base value (so if the cloud would be considered a low-level cloud, a medium-level cloud, or a high-level cloud) will be shown on the main interface window at the bottom, right around the buttons. The label that says “This cloud would be considered a:” would be right aligned in the first column, and the cloud type value will be left aligned in the second column. Then, to accommodate for this, the “show file data” button will instead be centered in the second column. The “compute” button would stay in its place at the right end of the second column and the same goes for the “login” button, too, where the “login” button will be aligned to the left end of the first column. So, this fixed the issue to me and added a nice, fun, new, educational, and informational function to the program (it is also the reason why the right column’s image is what it is in the first place, as it shows cloud types and the layers where they exist). Of course, these cloud level labels are not used yet in the program as they are tied to the functionality of the program (the calculation of the cloud base itself). Another unique aspect of this window that I have planned but is also not created yet as it is tied to the functionality of the program is the other images that will go into the right column of the window. My idea is that based on the value of the cloud base and the value of the cloud level, I want the image in the right column (where the initial image of the cloud types and levels is) to change and represent the cloud level type that was calculated, giving the user a real-life image of a potential cloud that is within the level of the atmosphere. However, as mentioned before, this part is not created yet as it is tied to functionality that is also not built yet. These images are already checked within the second try-catch block, so if these images are not in the project folder, the main interface will still not appear; again, all of the images for this program must be in the project folder in order for it to work. Lastly, at the bottom of the code is the standard functions for the window that each of the other two windows also have. Those being adding the panel to the GUI, making the GUI not resizable, making the window appear in the center of the screen, making the window visible, and setting the default closing operation to exiting the program (so if the user clicks on the “X” in the top right corner, the entire program is exited, too).

Main Interface of the Program:

Graphical user interface

Description automatically generated

All in all, there is more work to be done. At the current moment, at the end of this milestone, the startup window, the create account window, and the login window are all developed and functional. They are all functioning as I designed and intended. Additionally, the main interface of the program is made and appears in the way that I want it to initially appear. However, the main interface is not currently functional. The next steps that I will be taking with this program are:

* Complete the functionality of the main interface by adding event/action listeners for the “compute” and “exit” buttons (the functionality of the “show file data” button will be done and completed in milestones 4 and 5 as its functionality is tied to things that are a part of those milestones, like the data output window and the plot graph; it is possible that I may just have the button create a temporary “generic” default window just to show that the button is functional).
* Add error handling and data validation (as well as error message dialog boxes) for the user entered values of Temperature, Wind Speed, and Dew Point.
* Add the equation calculations for the wind chill factor and the cloud base altitude, as separate methods that are called.
* If the value for wind chill is invalid, then have error messages and dialog boxes appear informing the user as such.
* Implement functionality where the cloud level is determined by the value of the cloud base height, and then have that cloud level show up on the interface. Then, have the picture in the right column change depending on the level of the cloud.
* Have the resultant data appear on the main interface as long as it is valid.

**Milestone 3:**

For Milestone 3, I started developing and implementing the functionality of the main interface of the program, as well as the major functionality of the entire program. In this case, the main functionality is the ability for the user to enter in a valid value for the temperature (in degrees Fahrenheit), a valid value for the wind speed (in MPH), and a valid value for the dew point (in degrees Fahrenheit), and then for the program to compute, if applicable, the wind chill factor and the cloud base and display those results on the main interface. Additional functionality that I added but was not really required or necessary was to have the image on the right side of the window (the image with the different types of clouds) change depending on what the value of the cloud base is. I wanted the picture to correspond with how high or low the cloud is, which is determined by the cloud base. So, that was added, as well as a row on the main interface (right below where the cloud base is shown) where the program uses the cloud base to determine and tell the user what type of cloud it would be considered. Thus, the first step I took was building the action listener class for the “show file data” and “logout” buttons. The reason I did this is because they were easy to build and code, due to mainly being copy-and-paste or “unfinished”. The former condition was the case for the logout button, as it act and does the exact same thing as the “cancel” button on the “create account” and “login” windows. Therefore, I just copied-and-pasted the code and then changed the names of some of the text within it to correspond with this button and this class. In later Milestones, however, when more windows are created by the main interface, this class will be altered to add functionality that makes sure the program disposes of those other windows, panels, and frames as well when the user “logs out” of the program. The “show file data” button is simply incomplete due to its functionality being a part of Milestone 5. So, its functionality is developed later and thus all the button does right now is print a line of text to the console just like all the other buttons in the program do: printing a message that says that the button was clicked. That now takes us to the “compute” button. Which was where the true functionality of this program really begins.

*Section 1: The “Compute” Button and the Methods It Calls*

This button first gets the text from the 3 text fields that the user enters information into: the temperature field, the wind speed field, and the dew point field. Then it calls the “validWindChill” method to see if the wind chill is valid. Hopping into that method, there are multiple Boolean variables that act as “switches”. Basically, they will flip from their default (initialized) value of either true or false and be assigned the opposite value if the thing that it is looking for is detected. Going through the algorithm within this method, first the text field for the temperature is checked to see if it is blank, just like with the other text boxes in the program; if so, then an error message dialog box appears. Next, if the check is successful, then it gets the length of the text within the box and individually goes through each character in the text string to see if it is a digit. If there is a letter or other forbidden character, then the “switch”, tempDigits, is flipped to false, meaning not all of the characters in the text are digits. I did have trouble figuring out how to allow decimal numbers and negative numbers in it, but then I figured out that I could just check for the decimal point and negative sign characters and make sure that if that is detected, that the Boolean variable stays set to true. After all of that, if the text in the temperature box is all digits (as it should be), then the “goodTemp” switch, which checks for if the temperature value is valid, flips to true. Otherwise, an error message dialog box appears. Following the temperature verification, the wind speed value entered by the user goes through the exact same process: the box is checked to see if it is blank, each character is sifted through to see if it is all made of digits (excluding a possible decimal point; negative is not checked because the wind speed cannot be less than 3.0 MPH), and then if it is all digits, the “switch” which is the “goodWindSpeed” variable is flipped. There are error messages in the same spots as in the temperature verification process. It is all the exact same. Now that both the temperature and wind speed values have been examined by the program, if both “goodTemp” and “goodWindSpeed” are true, then the strings are converted into doubles and checked to see if they meet the physical bounds that nature allows for a wind chill to exist (temperature must be <= 50 degrees and wind speed must be > 3 MPH). If so, then the wind chill is valid and “goodWindChill” is set to true. Otherwise, the variable is set to false and an error message is printed in the console. The value of the variable is returned back to the “compute” button action listener class. Now the cloud base gets checked for validity.

This “verifyCloudBase” method is set up in the exact same way as the “verifyWindChill” method, except in this instance, only the dew point text field is being evaluated. So, same process, same thing. Boolean variables act as “switches”. The box is checked to see if it is blank; if not the verification continues along, and if so then an error message box appears. Each character in the text box string is looked at to see if it is a digit; if all of them are digits (excluding a decimal point or negative sign), then the “goodDew” switch is checked to true, otherwise an error message appears. Lastly, if that variable is true, then the “goodCloudBase” marker is also true. If not, then it is false. The value of “goodCloudBase” is returned to the action listener class.

Now that both the wind chill and the cloud base have been determined to be valid or not, the flow of control of the program goes into an if-else block. If both “validWindChill” and “validCloudBase” are deemed true, then we go on to calculate the wind chill and the cloud base, and we go a display all the data. If at least one of them is false, then we see which variable is indeed false, and display a corresponding error message dialog box to whichever variable was false. That way, the user knows exactly what went wrong with their values: whether the wind chill was incalculable (meaning that something was wrong with either their entered temperature value or their entered wind speed value) or the cloud base was incalculable (meaning that something was wrong with their entered dew point value). Then the text boxes are reset to be blank.

Going back to the situation where both variables are true, as I mentioned before, we then call the methods that calculate the wind chill and cloud base. Both of these methods are extremely simple. All they do is they take in the necessary variables that were passed to them (temperature and wind speed for the wind chill and temperature and dew point for the cloud base), convert the string variables it receives into doubles, and then uses it to calculate the wind chill or cloud base using their respective equation. For the cloud base though, there is a special case where it can not be negative, so if the calculated value of the cloud base from the equation is negative, well then, a simple if statement reassigns the cloud base value to be 0. After the calculations, the wind chill and cloud base values are returned to the “compute” button action listener class. Next, the values are printed out to the console for the user to see (they are formatted to two decimal places), and then the “displayDataMainGUI” method is called. This method is the main hub where, based on the data passed to it, all of the changes to the main interface is done.

*Section 2: Displaying the Data*

This method that displays all of the data onto the main GUI interface panel of the program begins by formatting the wind chill and cloud base text strings into formatted strings and assigns the result into a new string variable. Then, that new string variable is concatenated to include the units of each value (degrees Fahrenheit for the wind chill and feet for the cloud base). Next, the final text string is set as the text for the "windChillValueLabel” and “cloudBaseValueLabel” respectively. This solves one of the biggest problems that I had when developing this method, which was preventing the labels from overlapping each other when the user clicks on the “compute” button more than one time in a session (as in the user does not log out or exit the program every time after pressing the “compute” button). Using the .setText() method on those labels fixes the issue. This was also done for “cloudTypeValueLabel” so that this label, which displays the cloud type on the interface, also does not overlap on itself. But after setting the text for the labels, the GridBagConstraints are set for both the wind chill and cloud base value labels. They are the same constraints (same insets and centered) except for the wind chill its row constraint is 8, while it is 10 for the cloud base. This way they line up with their corresponding “result” label in the other column to their left. After that, the program uses the value of the cloud base to determine the type of cloud it is (low, medium, or high). This part is a giant if-else if-else if block of code, and inside each block, the code is exactly the same except some text is changed to correspond with the level of cloud that the block of code is for. For instance, in the first block, which is for if the cloud base is >= 0 and <= 7000, the parameter inside the .setText() method call is “Low-Level Cloud”. Additionally, the image variable used in that block is “rightImageFile4” and “rightImage4” as that is the image of the low clouds. But in the else if block following it, the text is changed to “Medium-Level Cloud” and the image used is “rightImageFile3” and “rightImage3” instead. But the constraints put on the “cloudTypeValueLabel” and the BufferedImage are the same in all three cases. After the cloud type value label is added to the panel, there is a try-catch block to add the respective image to the panel in replacement of the initial image that appears when the main interface is displayed. If the image is not there when it tries to add it to the panel, then an exception is thrown and caught, and an error message dialog box is display and a message is printed to the console saying that the image file is not found. This is the same for all three cases of the if-else if-else if block. Lastly, since aspects and components that are on the main interface panel have changed, the panel is revalidated and repainted so that the new data and whatnot is displayed for the user to see.

Screen Captures Showing Functionality and Error Messages

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generatedGraphical user interface, text, application, website

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedText

Description automatically generatedText

Description automatically generated with low confidenceGraphical user interface

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface

Description automatically generated

For now, that is the end of the method and that is the end of the “compute” button action listener class. Now that the main interface is mostly functional and the data from the user can be entered, validated, calculated, and displayed, the next steps are:

* To Create The Data Output Display Window (which displays the keyboard entered values for temperature, wind speed, and dew point, as well as the calculated wind chill and cloud base, in a separate window)
* To make sure the data table in the output display window is properly formatted and aligned (with Title, Headers, and Units).
* To make sure that the window will be updated and appended each time new valid values and data are entered on the Main GUI.

**Milestone 4:**

The main goal of Milestone 4 here was to create a simple, separate window that would display the keyboard-entered and calculated data in a neat, organized, and aligned columned data table complete with headers and units so that the user can know what each piece of data is. This columned data table displays the user entered values for the temperature, wind speed, and dew point, as well as the calculated wind chill and cloud base values. While it was not necessary, I also wanted to have the “cloud type” be displayed in the data table window as well so that the user has some context for interpreting the data that would be in the table. Essentially, displaying the “cloud type” in the window would allow the user to easily see what type of cloud the data values create. Another unnecessary function I added to this data display window was adding a scroll bar to the window. This way, the user can have as many rows of data as they want (and also this would help matters for me when I have to add file saving and displaying functionality in Milestone 5 so I don’t have to worry about the size of the file and how many rows are in it). Anyway, the window is working perfectly with all of the functionality that I outlined at the end of the previous milestone above. So, diving right in, we start at actually creating the window.

*Section 1: Creating the Window*

One of the first things that I thought of when I began working on this milestone was “**when** should I have this separate data display window appear?” I originally thought to have it appear when the “compute” button was clicked for the first time, and then have it stay up on subsequent clicks of the button. But after going through my code, I felt like doing it that way would force me to have to restructure the whole action listener that is tied to the “compute” button, so I decided against it. When looking through my code again, my thought was that I wanted to try to avoid altering a lot of my preexisting code from Milestone 3. I wanted to change as little of the code in the “WeatherGUI.java” file as possible. As a result, I just decided to have the window declared and appear when the main GUI appears as well. It made things so easy because all I had to do was add one line of a comment and one line of code at the top of the WeatherGUI class that declares the window. I also added a line right before the computeButtonListener class that moves the main GUI to be in front of the data output window, as the requirements said that the main GUI must appear in front first. The only other places where I had to add code into the “WeatherGUI.java” file was a line of code in the LogoutButtonListener class that disposes of this separate window when the “Logout” button is clicked, and the actual call to the updateData method in the WeatherDataOutputWindow class that will append the data in the data output window. Those were all the modifications I had to make to the code in the WeatherGUI class, which is good since I did not want to change much of its code. Now comes the class and constructor in the “WeatherDataOuputWindow.java” file that actually creates the window and all of the components that are with it. At the top of the WeatherDataOutputWindow class, I created the components that I wanted in this window: the main JFrame and JPanel, the text area where the data output will go (which was created to have a size of 40 rows and 95 columns, even though the scroll bar makes it basically infinite based on my testing), a JScrollPane that is attached to the aforementioned text area, and a String to hold the actual concatenated line of data. Then, in the constructor, I set the sizes for the window and panel. The sizes I landed on were based on trial and error and seeing if it looked good with the way the title and label headers are set up. The size of the window and panel are both 733 pixels wide and 308 pixels tall. Of course, after that I created the text areas for the title and the labels. The reason I choose to have three separate text areas is because it then makes it easy to have them all centered in the window. I then set the fonts for both headers and added then to the panel. The next paragraph of code was the more difficult to figure out as it was the part of the code that integrates the vertical scroll bar. I tried all different configurations and methods to try to get it to work. Most of the time in these early attempts, the scroll bar either: wouldn’t appear, appear on the side of the window but be “filled” (as in the bar wasn’t scrollable), be too small, or my data would not appear due to the existence of the scroll bar. After a whole day’s work of research, testing, and trial and error, I finally got the scroll bar to appear and function as I wanted it to. The solution was simple actually. I figured out that the reason why my data would not appear properly even though I had the scroll bar “attached” to the text area for the data output was due to the scroll bar being too small. There were some instances where I tried setting the size of the text area to enlarge it but that would not work because the size of the text area was not the problem. The issue was with the size of dataWindowScroll. So, what I had to do was set the size of dataWindowScroll. Setting the size of that would expand the text area that was attached to it, making both the text area and the scroll wheel seeable. When I figured it out, there was also a border around the text area due to the scroll bar, and I wanted to remove it, so I just simply set the scroll bar’s border to be an empty border, thus removing the border that was there by default. So, after all of that, I added dataWindowScroll to the Panel (which added the text area with it), set the font for the rest of the panel and began adding the JMenuBar that would be used in Milestone 5. I decided to add it now just to make my life easier when I eventually work on Milestone 5. It was pretty simple to set that up to have the file drop down menu and then to have the ‘save as’ item in the menu. I created the action listener for the ‘save as’ menu item but it is not functional yet because the “performSaveAs” method will not be built until Milestone 5. At the end of the constructor, I added the whole component filled panel to the window and set the other standard window settings that I would set like the location and default close operation. I set the initial window location to be slightly offset the main GUI so that the user would see it. Now that the window was created and complete, all that was needed now was to build the “updateData” method and add in the window’s update functionality.

Screen Capture of the Data Output Window (No Data Entered Yet)

Graphical user interface

Description automatically generated

*Section 2: Appending the Text Area to Display the Data*

With the window appearing, the last step was to actually make the window functional by having it have the entered and calculated data appear on the window and having it append or update itself when valid data is detected and the “compute” button is clicked. With the temperature, wind speed, dew point, wind chill, and cloud base coming into the method as parameters, I declared a string builder and an empty string to hold the result of concatenating all of the data outputs together into one long, one line string. Then, like what I had to do in the “displayDataMainGUI” method to display the data on the main GUI, I had to convert the temperature, wind speed, and dew point parameters into doubles since they came in as Strings. Then, I set the decimal and comma formatting all in one simple line with the Decimal Format method. That line makes sure that all of the data will satisfy the formatting requirements by making everything to 2 decimal places and by properly adding commas when needed in the numbers. After that, there is a for loop that loops 5 times (once for each piece of data). Inside this for loop, there are if statements to catch when the program should be working on the temperature, the wind speed, the dew point, the wind chill, and the cloud base values. The inside of each of these if statements all practically do the same thing, just for a different data variable. In these if statements, it applies the decimal formatting I mentioned before to the data value (and assigns the result to a variable called “item”, which was the empty String I talked about before), concatenates the data value to its respective units (creating a string that includes the value of the data variable and its appropriate units), sends “item” (now a String with data value and units) to the “formatItem” method (details on this method are below), and lastly appends the final String “item” to the big long “dataString” at the end. In the wind chill version of these if statements, there is also a check for if the value is negative and if it is, then the “item” string is concatenated slightly differently (by not adding a space because that space that would be there normally, when the value is positive, is now being taken up by the negative sign). Due to me deciding that I wanted to also display the cloud type in the output window, after the end of the for loop, I created an if-else if-else if block to handle adding the cloud type to the “dataString”. I set each type to have a certain “count” (for the “formatItem” method), and then follow the same process as in the others where I concatenate the string to the empty string “item” (due to no for loop, I simply reset the string in each of the three if blocks so that it would not display the cloud base value twice), send it to the “formatItem” method, and append it to the long String. I did this for each of the three cloud types, added a line feed to the “dataString” outside the if block, converted the “dataString” to a full String and appended it to the empty “dataOutputString” from the top of the class, and set the text in the “dataOutputTextArea” to be that fully complete, long String. While that is the end of the “updateData” method, there is still the “formatItem” method that needs to be discussed. The “formatItem” method that each output data value is put through is constructed similarly to the if block that concatenates the cloud type to the main String in the “updateData” method. There is a lot of if-else if statements, one for each possible “count” that keeps track of what data value is the program “looking at” or “focusing on”. I get the length of the “item” string that is passed to the method, which contains the data value concatenated to its respective units (except for the cloud type) and set the “spaces” variable to 0. Then, depending on what data value the program is looking at based on the “count”, I set the “spaces” variable to be whatever the length of the column is (which is from the end of the previous header’s text, for example “Temperature”, to the end of the header text for the data value it is looking at) minus the length of the string. This gives the amount of spaces that must be added to the string in order for the data to be formatted and aligned properly (which was basically “whatever looked nice” to me) in relation to its header above it. I wanted to pretty much have the data centered under its respective header label. I also built the method this way because each data value’s column is not a uniform length to each other; they are all basically different column sizes and lengths. So, I wanted to take that into account and personalize the spaces for each data value individually. That way, in all cases (whether the value of the data is large or small), the data will be aligned as I see fit. After the number of spaces are found, the method continues with a for loop that actually incorporate the spaces into a string which is then appended to the “item” string. The new “item” string is then returned and assigned to the “item” variable again.

Screen Capture of the Window (With Data Entered and Displayed)

Table

Description automatically generated

And that is it. The separate data output display window is made and functional, with the keyboard-entered data and the calculated data being displayed in the window every time the “compute” button is clicked and the data is valid. Now, the only things left to do regarding this program (according to the requirements) are:

* Implement the “File Save As” functionality to the output window’s JMenuBar so that the data in the output window could be saved as/into a text file.
* Implement the functionality for the “Show File Data” button, which will use a JFileChooser to allow the user to select a text file and have the data in that selected text file to be displayed in the data output display window.
* Implement a window that plots the temperature and wind chill values entered from the keyboard onto a scatter plot (point) graph that also updates when new valid data is entered and the “compute” button is clicked. The values will be represented by different colors on the graph and the values of the variables will also be displayed.

**Milestone 5:**

With the knowledge I accrued from the first 4 milestones, milestone 5 was a breeze for the most part. I first started with actually implementing the ‘saving’ functionality for the Data Output Window and applying it to the ‘save as’ menu option in the window’s drop-down menu that I created during milestone 4. Then, I went to develop the algorithm for opening a file with data saved in it and being able to view it in the Data Output Window. Lastly, I made a scatter plot graph in a separate class and window that plots the user-inputted temperature value and the subsequent, calculated wind chill value on the graph.

*Section 1: The ‘Save As’ Functionality*

Inside the WeatherDataOutputWindow class, I created an ActionListener for the ‘save as’ menu option in the output window’s drop-down menu. What this ActionListener class does is it prints a message to the console saying that the ‘save as’ button was selected, and then it calls the method that performs the actual ‘file saving’ functionality, which is called performSaveAs. In the performSaveAs method, I declared a JFileChooser that automatically opens to the current directory that the program/user is in, and I also set a file name extension filter so that only text files (.txt) will be shown. This makes things easier because I only want the user to be saving the data in the output window as a simple, easy to read text file. So, when the user goes to save the data in the output window, I made it so a JFileChooser window will appear for the user to select a file from. There is a simple if-else statement that controls the flow-of-control depending on what the user does. Every possible thing that the user can do when the JFileChooser window is opened is handled in the program. If the user selects a file, then I get that file, print the file path to the console, and then the program tries to write the output window’s data to the file. The reason that code is in a try-catch block is just in case the FileWriter throws an exception. If it throws an exception for if the file is not found and can’t be created or for if the file cannot be opened, then the error stack and an error message is printed to the console. If the user does not select a file that already exist, but instead types in a name for the file, the file will automatically be created and the writing process will go on as normal. So, if no exception is thrown, then the PrintWriter will write the data to the file. After that is done, the PrintWriter is closed. Lastly, there is the scenario where the user hits the ‘cancel’ button on the JFileChooser window. If that occurs, it is handled in the ‘else’ part of the if-else statement. All the program does is cancel the operation and a message saying that no file was selected is displayed on the console. It is as simple as that.

Screen Captures of the ‘Save As’ Functionality in Action

Graphical user interface, text, application

Description automatically generatedA picture containing logo

Description automatically generatedTable

Description automatically generated

*Section 2: Opening and Viewing the Contents of a File*

Now that the ‘saving’ functionality was implemented, there was one last method that I needed to create in the WeatherDataOutputWindow class: the openViewFileData method. This method allows the user to select a file, open its contents, which should just be the header labels and lines of data, and display the contents of the file onto the output window. So, I created the openViewFileData method in the WeatherDataOutputWindow class. This method is called by the ActionListener for the “Show File Data” button that is on the main GUI of the program. So, when that button is clicked. A JFileChooser will appear, similar to that with the ‘save as’ functionality: the user’s current directory will automatically be opened and the user will be filtered to only be able to select text (.txt) files to open. As I alluded to already, the structure of this method is extremely similar to the performSaveAs method. However, in this method, the file MUST already exist, if the file cannot be found or opened, the Scanner will throw an exception and it is handled in a catch block: A error message dialog box will appear saying that the file either cannot be found or cannot be opened, and then that same message is printed to the console along with the error stack trace. Additionally, what happens next, after the user selects a file, is also different from the previous method. When the user selects a file to open, like before, the program brings in the file to be opened and prints its file path to the console. But unlike in the performSaveAs method, I then declared a Scanner to scan the file, and a StringBuilder to help build the string that later will be displayed in the output window’s JTextArea. After those declarations, I have a while loop set up. This while loop will keep looping until there is no more lines in the file to be read. So, I was all set up to start reading in the lines of the file when I realized that when I read the first two lines of the file, those will always be the header that is identical to the header in the output window. Due to that, I had to engineer a way to essentially “skip” over the header and only take in what I wanted which is the actual lines of data. So, what I did is in the while loop, I initialized temporary String variables to hold the text that make up the header; this includes the row or line that are the actual labels and the row or line below it which is just a line of dashes. Then I would use nextLine to take the next line in the file. Again, for the first two lines read, the StringBuilder would take in and contain the parts of the header. Next, I have an if-if else-else block that changes the flow-of-control depending on if what is read into the StringBuilder is one of the two parts of the header. In both instances where what is read is a part of the header from the file, I simple invoke the delete method of the StringBuilder to remove the header parts from the StringBuilder (since each part is of two different lengths, I had to separate them). Then, the while loop loops back again to read the first line with the data in it. When the data lines are read from the file and appended into the StringBuilder, I then concatenate it into the dataOutputString, which is the String variable that is used in setText to add the String to the JTextArea of the output window, and a line feed so that each read line of data will be on a new line. Then, the full string of data is then added to dataOutputTextArea which is the JTextArea variable of the output window. With all of this combined, the data from the file is placed onto the output window with its formatting and spacing all preserved, so it looks neat and nice.

Screen Captures of the ‘File Opening and Viewing’ Functionality

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generatedTable

Description automatically generatedGraphical user interface, text

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

*Section 3: Plotting the Temperature and Wind Chill*

Finally, there is the last part of the program that I needed to integrate: a window containing a scatter plot graph that plots the user-entered value for the temperature as well as the calculated value for the wind chill. Starting out, I created the WeatherPlotGraphDisplayWindow file and class to contain all of the code relating to this graphing window and functionality. Once the class was created, I then went back to the WeatherGUI class to do the same process that I did when I made the output window’s class. What I did here was: 1.) At the top of the WeatherGUI class, I declared the actual window that the graph will be on. Then, I went down the class and right under where I put the call to the “updateData” method for the output window, I put the call to the “updateGraph” method for the graph’s window (details below). Lastly, in the WeatherGUI class’s action listener for the “Logout” button, I added a call to a “disposeOfWindow” method, which simply gets rid of the window when the “Logout” button is clicked, just like it does with the Main GUI and the Output Window (inside the WeatherPlotGraphDisplayWindow class, the “disposeOfWindow” method is built exactly the same as the one that is in the WeatherDataOutputWindow class). Once I was done with the last additions to the WeatherGUI class, I went back to the WeatherPlotGraphDisplayWindow class and declared the JFrame for the window’s GUI and an ArrayList, which will store all of the data point values to be drawn on the graph. Next, I constructed the constructor for the class. Inside the constructor, I set the size and background color of the window. And then I declared a new JComponent. This JComponent contains a method called paintComponent, and that method, inside the constructor, sets the fonts for the title and the data point text, draws the title, the numbers that label the degrees Fahrenheit for the temperature and wind chill, the x-axis, the y-axis, and the major (lines indicating the increments of 10 degrees) and minor (lines indicating the increments of 5 degrees) horizontal lines for the graph. At the end of this paintComponent method (which is also the end of the JComponent block itself), is a for loop. This for loop is the loop that actually conducts the process of plotting the data points for the temperature and wind chill values. In the for loop, the ArrayList gets its first item, which is the temperature. Then the program gets the temperature’s “height”, sets the color that the point and text should be, and draws the text label and the point itself. Then, the ArrayList gets the next item, which is the wind chill, and then the program does the same things for the wind chill value as it did for the temperature. At the end of the for loop, the counter *i* is incremented by 1 and the x coordinate position for the next point is set to be 45 units to the right of the previous point’s position. The cycle continues for however big as the ArrayList is. Next, at the bottom of the constructor, but outside the JComponent’s method and declaration block, I added the typical lines of code that I have for GUI windows, including adding the JComponent to the JFrame, making the window visible, making it so the user cannot resize the window, setting the initial location of the window when it appears on screen (which I made to be offset of both the Output Window’s position and the Main GUI’s position), and making sure that the default close operation for the window is that the window closes when the ‘X’ is clicked in the top right corner of the window. The absolute last thing to discuss is the “updateGraph” method that I mentioned earlier. This method is below the class’s constructor, outside of it. It takes in the temperature and the wind chill in as parameters from the WeatherGUI class. The temperature is taken in as a String variable and the wind chill as a double variable. Now, due to the temperature being taken in as a String variable, I had to convert it to a double first so that the ArrayList can take in the temperature value since the ArrayList is defined to take in Integers. When the temperature and wind chill values are added to the ArrayList, the ArrayList’s add method will convert them to integers in the process. Lastly, the window calls the repaint method so that the window’s graphics update to repaint all of the data points that should be plotted on the graph.

Plotting the Temperature and Wind Chill on a Scatter Plot Graph

Table

Description automatically generatedText, application, email

Description automatically generated

Note: For those two “-10 dF” values (the 3rd and 4th Temperature points), the wind chill value was lower than -30 dF, so they are off the graph.

And that is that. The program now fulfills all of the requirements on the assignment’s instructions.

While I did add things that the instructions did not say I needed to add, I felt like adding them enhances the program for the user significantly, especially things like the dialog box after the user creates an account asking them if they want to go directly to the login screen or to go back to the home screen and the vertical scroll bar on the data output window. There were also aspects that I wanted to add, mainly I wanted to see if I could add a horizontal scroll bar to the scatter plot graph so that the user could plot as many data point values as they wanted, but I unfortunately was running out of time I felt and scrapped the idea. Maybe in the future I will revisit this and add different aspects and functionalities as I learn more and more. But, for now, I feel like the program is complete!

All of the source code for the program is included below.

**Source Code:**

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherData.java

// This class serves as the main operator for the flow of control of the program.

**package** weatherPackage;

// imports

**import** javax.swing.\*;

**import** javax.swing.JOptionPane;

**public** **class** WeatherData **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

**public** **static** **void** main(String[] args) {

// When the program launches, a dialog box appears asking

// the user if they are looking to create an account, login,

// or cancel the program.

Object[] options = {"Cancel", "Create Account", "Login"}; // choices for the dialog box

**int** selection = JOptionPane.*showOptionDialog*(**null**, // no parent window, so it is null

"\n\n\n\tWeather Data Analysis Program \t\n\n\n\n", // text inside the dialog box

"Weather Data Analysis Program", // text on the border of the dialog box

JOptionPane.***YES\_NO\_CANCEL\_OPTION***, // option type of dialog box

JOptionPane.***QUESTION\_MESSAGE***, // message type

**new** ImageIcon("WeatherIcon.png"), // image icon

options, options[2]); // the options, and initial value (buttons)

// user selects one of three options in the dialog box; flow of control

**if**(selection == 1) // if user selects "Create Account", then create its window

{

System.***out***.println("Creating Account");

**new** WeatherCreateAccount();

}

**else** **if**(selection == 2) // if user selects "Login", then create its window

{

System.***out***.println("Login");

**new** WeatherLogin();

}

**else** // if user selects "Cancel", then close out the program

{

System.***out***.println("Cancel");

}

} // end of main

}; // end of class

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherCreateAccount.java

// This class creates and holds the functionality for the "Create Account" window of the program.

**package** weatherPackage;

// imports

**import** java.awt.GridBagConstraints;

**import** java.awt.GridBagLayout;

**import** java.awt.Insets;

**import** javax.swing.\*;

**import** java.awt.event.\*;

**import** java.io.\*;

**import** java.util.\*;

**public** **class** WeatherCreateAccount **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// Creates GUI Frame and Panel

JFrame createAccGUI = **new** JFrame("Weather Data Analysis Program");

JPanel createAccPanel = **new** JPanel(**new** GridBagLayout());

// Creates Labels for Username and Password

JLabel userNameLabel = **new** JLabel("Enter a unique Username: ");

JLabel passWordLabel1 = **new** JLabel("Create a password. Password must:");

JLabel passWordLabel2 = **new** JLabel(" - Have at least one (1) uppercase letter.");

JLabel passWordLabel3 = **new** JLabel(" - Have at least one (1) lowercase letter.");

JLabel passWordLabel4 = **new** JLabel(" - Be at least nine (9) characters long.");

JLabel passWordLabel5 = **new** JLabel(" - Have at least one (1) digit.");

// Creates Username and Password Text Boxes

JTextField textFieldUsername = **new** JTextField("", 11);

JTextField textFieldPassword = **new** JTextField("", 11);

// Creates the "Create Account" Confirmation Button

JButton createAccButton = **new** JButton("Create Account");

// Creates the "Cancel" Button

JButton createAccCancelButton = **new** JButton("Cancel");

// Constructor With Configurations

**public** WeatherCreateAccount() {

createAccGUI.setSize(500, 350); // sets the width, then the height of the window

GridBagConstraints con = **new** GridBagConstraints();

con.insets = **new** Insets(1, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 1; // Sets the Row Constraint to 1

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(userNameLabel, con); // Adds Username Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

createAccPanel.add(textFieldUsername, con); // Adds Username Text Box to Panel

textFieldUsername.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 4; // Sets the Row Constraint to 4

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(passWordLabel1, con); // Adds Password Label #1 to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 5; // Sets the Row Constraint to 5

con.insets = **new** Insets(5, 3, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(passWordLabel2, con); // Adds Password Label #2 to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 6; // Sets the Row Constraint to 6

con.insets = **new** Insets(5, 3, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(passWordLabel3, con); // Adds Password Label #3 to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 7; // Sets the Row Constraint to 7

con.insets = **new** Insets(5, 3, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(passWordLabel4, con); // Adds Password Label #4 to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 8; // Sets the Row Constraint to 8

con.insets = **new** Insets(5, 3, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

createAccPanel.add(passWordLabel5, con); // Adds Password Label #5 to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 8; // Sets the Row Constraint to 8

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

createAccPanel.add(textFieldPassword, con); // Adds Password Text Box to Panel

textFieldPassword.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(40, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

createAccPanel.add(createAccButton, con); // Adds the "Create Account" Button to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(40, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns to the Left

createAccPanel.add(createAccCancelButton, con); // Adds the "Cancel" Button to the Panel

// sets a nice boarder for the panel

createAccPanel.setBorder(BorderFactory.*createTitledBorder*(BorderFactory.*createEtchedBorder*(), "Weather Data Anyalsis Project - Create Account"));

createAccGUI.add(createAccPanel); // Adds the Panel to the GUI

createAccGUI.setResizable(**false**); // Makes User Unable to Resize the Window

createAccGUI.setLocationRelativeTo(**null**); // Makes Initial Window Location Centered on the Screen

createAccGUI.setVisible(**true**); // Makes the GUI visible

// Presumes that the user clicked on the "X" in the right corner of the window, not the "Cancel" button

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); // When the "Create Account" window is closed, the default option is to stop/terminate the whole program

// the fclickollowing is a class that uses action listeners to implement functionality of the Create Account button

**class** CreateButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\nWeatherCreateAccount class: Create Account button was clicked.");

// get the inputs that the user entered into the text fields

String uName = textFieldUsername.getText();

String pWord = textFieldPassword.getText();

// input validation - verify the account's credentials

**boolean** success = verifyAcc(uName, pWord);

// flow of control for if the account is valid or not

**if**(success == **true**)

{

JOptionPane.*showMessageDialog*(**null**, "\n Successful Account Creation! \n"); // Successful Account Creation dialog box

// system message to print username and password to the console for user reference

System.***out***.println("\nUsername: " + uName + "\tPassword: " + pWord);

createAccGUI.dispose(); // get rid of the account creation window

callSuccAccDirectBox(); // calls the method

}

**else**

{

JOptionPane.*showMessageDialog*(**null**, "\n Invalid Username and/or Password! \n Please enter another Username and Password. \n");

// system message to print invalid username and password to the console for user reference

System.***out***.println("\nUsername: " + uName + "\tPassword: " + pWord);

textFieldUsername.setText("");

textFieldPassword.setText("");

}

} // end of method

// this method calls the method that brings up a dialog box for the user to choose where to go next: either back to the "home" screen or straight to the login window

**public** **void** callSuccAccDirectBox() {

successAccountDirectBox();

}

}; // end of CreateButtonListener class

// Add the above ActionListener to the "Create Account" button

createAccButton.addActionListener(**new** CreateButtonListener());

// the following class uses action listeners to implement functionality of the Cancel button

**class** CancelButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("WeatherCreateAccount class: Cancel button was clicked.");

createAccGUI.dispose(); // get rid of the account creation window

callWeatherDataMain(); // calls the method

}

// this method calls the main method of WeatherData to bring up the startup dialog box

**public** **void** callWeatherDataMain() {

WeatherData.*main*(**null**);

}

}; // end of CancelButtonListener class

// Add the above Action Listener to the "Cancel" button

createAccCancelButton.addActionListener(**new** CancelButtonListener());

} // end of constructor

// verify account method

**boolean** verifyAcc(String uName, String pWord) {

// declare boolean variables for verification; will flip to true if it exists in the username/password

**boolean** goodPass = **false**; // does password have all required elements?

**boolean** upperCase = **false**; // are there uppercase letters?

**boolean** lowerCase = **false**; // are there lowercase letters?

**boolean** digits = **false**; // are there digits?

**boolean** longEnough = **false**; // is it greater than or equal to 9 characters?

**boolean** goodUser = **true**; // does username fit requirements? as in: does an account already use it or not; flips to true if not already used

**boolean** goodAccount = **false**; // are username and password valid?; flips to true if both are valid

// algorithm

// if password field is blank

**if**(pWord.contentEquals(""))

{

JOptionPane.*showMessageDialog*(**null**, "\n Password field is blank! \n Please enter a valid password. \n");

}

**else** // otherwise, continue password verification process

{

**int** length = pWord.length(); // get the length of the password

**if**(length >= 9) {

longEnough = **true**; // set longEnough to true if password is 9 characters or longer

}

// for each character of the password, check if it is a digit, an uppercase letter, or a lowercase letter

// if it is any of them, set that variable to true

**for**(**int** i = 0; i < length; i++) {

**char** c = pWord.charAt(i); // get character

**if**(Character.*isDigit*(c)) {

digits = **true**;

}

**if**(Character.*isUpperCase*(c)) {

upperCase = **true**;

}

**if**(Character.*isLowerCase*(c)) {

lowerCase = **true**;

}

} // end of for loop

// if all password requirements are met, flip goodPass to true

**if**((longEnough == **true**) && (digits == **true**) && (upperCase == **true**) && (lowerCase == **true**))

{

goodPass = **true**;

}

} // end of if-else block for the password verification

// if the username field is left blank, display error message

**if**(uName.contentEquals(""))

{

JOptionPane.*showMessageDialog*(**null**, "\n Username field is blank! \n Please enter a Username. \n");

}

**else** // otherwise, continue with account verification

{

File accountFile = **new** File("AccountInfo.txt"); // declares new file to store valid account information

**try**(Scanner in = **new** Scanner(accountFile)) // declares new scanner to scan the file and tries it to see if it exists; if file does not exist, exception is thrown and handled

{

**if**(accountFile.length() == 0) // if there are no accounts on file; nothing in text file

{

System.***out***.println("\nUsername is available!"); // system message

goodUser = **true**;

// if username and password satisfies all requirements, then it is verified!

**if**((goodUser == **true**) && (goodPass == **true**)) {

goodAccount = **true**; // password is valid!

// now write the valid account information to the file

FileWriter FW;

**try** {

FW = **new** FileWriter(accountFile, **true**);

BufferedWriter BW = **new** BufferedWriter(FW);

PrintWriter PW = **new** PrintWriter(BW);

PW.println(uName + "\t" + pWord); // prints the valid username and password into the file, they are on one line but tab delimited

PW.close(); // close print writer

}

**catch**(IOException e) {

e.printStackTrace();

}

} // end of inside if statement

}

**else** // if there are accounts on file

{

**while**((in.hasNext()) && (goodUser == **true**)) {

String text = in.next(); // reads up to the delimiter (whitespace)

// if the read content equals the username entered, then the username already exists

**if**(text.contentEquals(uName))

{

JOptionPane.*showMessageDialog*(**null**, "\n Username already in use! \n Please enter another Username. \n");

goodUser = **false**;

}

**else** **if**(text.contentEquals(pWord)) // if what is read is a password that is the same as the inputted one

{

System.***out***.println("\n CAUTION: Identical password detected on file. Please consider changing it."); // system message

}

} // end of while loop

in.close(); // closes Scanner when done

**if**(goodUser == **true**) // if the username is not detected, then it is avaliable

{

System.***out***.println("\nUsername is available!"); // system message

goodUser = **true**;

}

// if username and password satisfies all requirements, then it is verified!

**if**((goodUser == **true**) && (goodPass == **true**)) {

goodAccount = **true**; // password is valid!

// now write the valid account information to the file

FileWriter FW;

**try** {

FW = **new** FileWriter(accountFile, **true**);

BufferedWriter BW = **new** BufferedWriter(FW);

PrintWriter PW = **new** PrintWriter(BW);

PW.println("\n" + uName + "\t" + pWord); // prints the valid username and password into the file, they are on one line but tab delimited

PW.close(); // close print writer

}

**catch**(IOException e) {

e.printStackTrace();

}

} // end of inside if statement

} // end of else block

} // end of try block

**catch**(FileNotFoundException e) {

System.***out***.println("\nThe Account Info input file cannot be found or opened.\n");

e.printStackTrace();

} // end of catch block

} // end of outside else block

**return** goodAccount;

} // end of verifyAcc method

// Successful Account Completion Directive Dialog Box Method

**public** **void** successAccountDirectBox() {

// After successful account creation, this dialog box will appear asking the user where they want to go next

Object[] options = {"Go to Home Screen", "Go to Login Screen"}; // choices for the dialog box

**int** selection = JOptionPane.*showOptionDialog*(**null**, // no parent window, so it is null

"\n\n\nWhere Would You Like To Go Next?\n\n\n", // text inside the dialog box

"Weather Data Analysis Program", // text on the border of the dialog box

JOptionPane.***YES\_NO\_OPTION***, // option type of dialog box

JOptionPane.***QUESTION\_MESSAGE***, // message type

**null**, // no image icon

options, options[1]); // the options, and initial value (buttons)

// if user selects to go to the Home menu

**if**(selection == 0)

{

System.***out***.println("\nGoing to Home Screen...\n"); // system message

WeatherData.*main*(**null**); // calls method for main startup of program

}

**else** **if**(selection == 1) // if the user selects to go to the Login screen directly

{

System.***out***.println("\nGoing to Login Screen...\n"); // system message

**new** WeatherLogin(); // calls the constructor for the Login window

}

**else** // otherwise, like clicking the "X" in the top right of the window, close out the window and program

{

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

}

} // end of successAccountDirectBox method

}; // end of WeatherCreateAccount class

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherLogin.java

// This class creates and holds the functionality for the "Login" window of the program.

**package** weatherPackage;

//imports

**import** java.awt.GridBagConstraints;

**import** java.awt.GridBagLayout;

**import** java.awt.Insets;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.Scanner;

**import** javax.swing.\*;

**public** **class** WeatherLogin **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// Creates GUI Frame and Panel

JFrame loginGUI = **new** JFrame("Weather Data Analysis Program");

JPanel loginPanel = **new** JPanel(**new** GridBagLayout());

// Creates Labels for Username and Password

JLabel userNameLabel = **new** JLabel("Enter a Username: ");

JLabel passWordLabel = **new** JLabel("Enter a Password: ");

// Creates Username and Password Text Boxes

JTextField textFieldUsername = **new** JTextField("", 11);

JTextField textFieldPassword = **new** JTextField("", 11);

// Creates the "Login" Confirmation Button

JButton loginButton = **new** JButton("Login");

// Creates the "Cancel" Button

JButton loginCancelButton = **new** JButton("Cancel");

// Constructor with Configurations

**public** WeatherLogin() {

loginGUI.setSize(400, 250); // sets the width, then the height of the window

GridBagConstraints con = **new** GridBagConstraints();

con.insets = **new** Insets(1, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 1; // Sets the Row Constraint to 1

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

loginPanel.add(userNameLabel, con); // Adds Username Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

loginPanel.add(textFieldUsername, con); // Adds Username Text Box to Panel

textFieldUsername.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 4; // Sets the Row Constraint to 4

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns Text to the Left

loginPanel.add(passWordLabel, con); // Adds Password Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 4; // Sets the Row Constraint to 4

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

loginPanel.add(textFieldPassword, con); // Adds Password Text Box to Panel

textFieldPassword.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(40, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Right

loginPanel.add(loginButton, con); // Adds the "Login" Button to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(40, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns to the Left

loginPanel.add(loginCancelButton, con); // Adds the "Cancel" Button to the Panel

// sets a nice boarder for the panel

loginPanel.setBorder(BorderFactory.*createTitledBorder*(BorderFactory.*createEtchedBorder*(), "Weather Data Anyalsis Project - Login"));

loginGUI.add(loginPanel); // Adds the Panel to the GUI

loginGUI.setResizable(**false**); // Makes User Unable to Resize the Window

loginGUI.setLocationRelativeTo(**null**); // Makes Initial Window Location Centered on the Screen

loginGUI.setVisible(**true**); // Makes the GUI visible

// Presumes that the user clicked on the "X" in the right corner of the window, not the "Cancel" button

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); // When the "Login" window is closed, the default option is to stop/terminate the whole program

// the following is a class that uses action listeners to implement functionality of the Login button

**class** LoginButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\nWeatherLogin class: Login button was clicked.");

// get the inputs that the user entered into the text fields

String uName = textFieldUsername.getText();

String pWord = textFieldPassword.getText();

// input validation - verify the account's credentials

**boolean** success = verifyAcc(uName, pWord);

// flow of control for if the account is valid or not

**if**(success == **true**)

{

JOptionPane.*showMessageDialog*(**null**, "\n Login Successful!\n"); // Successful Login dialog box

// system message to print username and password to the console for user reference

System.***out***.println("\nUsername: " + uName + "\tPassword: " + pWord);

loginGUI.dispose(); // get rid of the login window

**new** WeatherGUI(); // create a new instance of the main Weather GUI

}

**else**

{

JOptionPane.*showMessageDialog*(**null**, "\n Invalid Username and/or Password\n or\n Account does not exist! \n\n Please enter another Username and Password\n or\n Create an account. \n");

// system message to print invalid username and password to the console for user reference

System.***out***.println("\nUsername: " + uName + "\tPassword: " + pWord);

textFieldUsername.setText("");

textFieldPassword.setText("");

}

} // end of method

}; // end of CreateButtonListener class

// Add the above ActionListener to the "Login" button

loginButton.addActionListener(**new** LoginButtonListener());

// the following class uses action listeners to implement functionality of the Cancel button

**class** CancelButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("WeatherLogin class: Cancel button was clicked.");

loginGUI.dispose(); // get rid of the account creation window

callWeatherDataMain(); // calls the method

}

// this method calls the main method of WeatherData to bring up the startup dialog box

**public** **void** callWeatherDataMain() {

WeatherData.*main*(**null**);

}

}; // end of CancelButtonListener class

// Add the above Action Listener to the "Cancel" button

loginCancelButton.addActionListener(**new** CancelButtonListener());

} // end of constructor

// verify account method

**boolean** verifyAcc(String uName, String pWord) {

// declare boolean variables for verification

**boolean** userExists = **true**; // is the username text field not blank (does it have text in it)?

**boolean** passExists = **true**; // is the password text field not blank (does it have text in it)?

**boolean** goodAccount = **false**; // are username and password in the file?; flips to true if both are in the file

// algorithm

// if password field is blank

**if**(pWord.contentEquals(""))

{

JOptionPane.*showMessageDialog*(**null**, "\n Password field is blank! \n Please enter a valid password. \n");

passExists = **false**;

}

**if**(uName.contentEquals("")) // if username field is blank

{

JOptionPane.*showMessageDialog*(**null**, "\n Username field is blank! \n Please enter a Username. \n");

userExists = **false**;

}

// if there is a username and password entered into the text fields

**if**((userExists == **true**) && (passExists == **true**))

{

File accountFile = **new** File("AccountInfo.txt"); // declares new file to store valid account information

**try**(Scanner scan = **new** Scanner(accountFile)) // declares new scanner to scan the file and tries it to see if it exists; if file does not exist, exception is thrown and handled

{

**while**((scan.hasNext()) && (goodAccount == **false**)) {

String user = scan.next(); // reads up to the delimiter (whitespace); gets the username

String pass = scan.next(); // reads up to the next delimiter (whitespace); gets the password

// if both the inputted username and password are found in the file, then the account is found and valid

**if**((user.contentEquals(uName)) && (pass.contentEquals(pWord))) {

goodAccount = **true**;

}

} // end of while loop

scan.close(); // closes Scanner when done

}

**catch**(FileNotFoundException e) {

System.***out***.println("\nThe Account Info file cannot be found or opened.\n");

e.printStackTrace();

} // end of try-catch block

} // end of if statement

**return** goodAccount;

} // end of verifyAcc method

}; // end of WeatherLogin class

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherGUI.java

// This class creates and holds the functionality for the main GUI interface of the program.

**package** weatherPackage;

// imports

**import** java.awt.GridBagConstraints;

**import** java.awt.GridBagLayout;

**import** java.awt.Insets;

**import** java.awt.event.\*;

**import** java.awt.image.\*;

**import** javax.swing.\*;

**import** javax.imageio.\*;

**import** java.io.\*;

**public** **class** WeatherGUI **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// Create the separate data display output window

WeatherDataOutputWindow ddWindow = **new** WeatherDataOutputWindow();

// Create the plot graph display window

WeatherPlotGraphDisplayWindow plotWindow = **new** WeatherPlotGraphDisplayWindow();

// Creates GUI Frame and Panel

JFrame WeatherGUI = **new** JFrame("Weather Data Analysis Program");

JPanel WeatherPanel = **new** JPanel(**new** GridBagLayout());

// Create Labels for User Inputs and Results

JLabel enterTempLabel = **new** JLabel("Enter the Temperature in Degrees Fahreheit: ");

JLabel enterWindSpeedLabel = **new** JLabel("Enter the Wind Speed in MPH: ");

JLabel enterDewPointLabel = **new** JLabel("Enter the Dew Point in Degrees Fahreheit: ");

JLabel windChillResultLabel = **new** JLabel("The Wind Chill Factor in Degrees Fahrenheit is: ");

JLabel cloudBaseResultLabel = **new** JLabel("The Cloud Base Altitude in Feet is: ");

JLabel cloudTypeLabel = **new** JLabel("This Cloud Would Be Considered A: ");

// Create Tempertature, Wind Speed, and Dew Point Text Boxes

JTextField textFieldTemperature = **new** JTextField("", 8);

JTextField textFieldWindSpeed = **new** JTextField("", 8);

JTextField textFieldDewPoint = **new** JTextField("", 8);

// Create Image Files

File leftImageFile = **new** File("WindChill.png");

File rightImageFile = **new** File("CloudTypes.jpg");

File rightImageFile2 = **new** File("HighClouds.jpg");

File rightImageFile3 = **new** File("MediumClouds.jpg");

File rightImageFile4 = **new** File("LowClouds.jpg");

JLabel rightImageLabel; // declare JLabel for the "rightImage" Files above

BufferedImage rightImage; // declares a BufferedImage for the "rightImage" Files

Boolean foundImage = **true**; // Boolean variable to determine if the image files exist or not

// create labels for the resulting data values for the wind chill, cloud base, and cloud type

JLabel windChillValueLabel = **new** JLabel();

JLabel cloudBaseValueLabel = **new** JLabel();

JLabel cloudTypeValueLabel = **new** JLabel();

// Create Compute Confirmation Button

JButton computeButton = **new** JButton("Compute");

// Create Show File Data Button

JButton showFileDataButton = **new** JButton("Show File Data");

// Create Logout Button

JButton logoutButton = **new** JButton("Logout");

// Constructor with Configurations

**public** WeatherGUI() {

WeatherGUI.setSize(1000, 620); // sets the width, then the height of the window

GridBagConstraints con = **new** GridBagConstraints();

con.insets = **new** Insets(1, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

// for image on the left of the interface panel

**try** {

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 0; // Sets the Row Constraint to 0

con.anchor = GridBagConstraints.***NORTH***; // Aligns Image to the North (Top)

BufferedImage leftImage = ImageIO.*read*(leftImageFile); // makes the image file into a useable image

JLabel leftImageLabel = **new** JLabel(**new** ImageIcon(leftImage)); // make image a JLabel so that the .add function below can use it

WeatherPanel.add(leftImageLabel, con); // add image to panel

}

**catch**(Exception e) { // if image is not found/does not exist/can't be read by ImageIO.read

foundImage = **false**; // set Boolean variable to false since image is missing

JOptionPane.*showMessageDialog*(**null**, "\nWindChill.png image file not Found!\n"); // dialog box

System.***out***.println("\nWindChill.png image file not found!\n"); // system error message

e.printStackTrace();

}

// for image on the right of the interface panel

**try** {

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 0; // Sets the Row Constraint to 0

con.anchor = GridBagConstraints.***NORTH***; // Aligns Image to the North (Top)

rightImage = ImageIO.*read*(rightImageFile); // makes the image file into a useable image by assigning it to the BufferedImage variable

rightImageLabel = **new** JLabel(**new** ImageIcon(rightImage)); // make image a JLabel so that the .add function below can use it

WeatherPanel.add(rightImageLabel, con); // add image to panel

}

**catch**(Exception e) { // if image is not found/does not exist/can't be read by ImageIO.read

foundImage = **false**; // set Boolean variable to false since image is missing

JOptionPane.*showMessageDialog*(**null**, "\nCloudTypes.jpg Image file not Found!\n"); // dialog box

System.***out***.println("\nCloudTypes.jpg Image file not found!\n"); // system error message

e.printStackTrace();

}

**if**(foundImage == **true**) // if all images exist and are found, then continue building the main interface; otherwise, program ends (code falls through)

{

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 2; // Sets the Row Constraint to 2

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(enterTempLabel, con); // Adds Temperature Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 2; // Sets the Row Constraint to 2

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(textFieldTemperature, con); // Adds Temperature Text Box to Panel

textFieldTemperature.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 4; // Sets the Row Constraint to 4

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(enterWindSpeedLabel, con); // Adds Wind Speed Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 4; // Sets the Row Constraint to 4

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(textFieldWindSpeed, con); // Adds Wind Speed Text Box to Panel

textFieldWindSpeed.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 6; // Sets the Row Constraint to 6

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(enterDewPointLabel, con); // Adds Dew Point Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 6; // Sets the Row Constraint to 6

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(textFieldDewPoint, con); // Adds Dew Point Text Box to Panel

textFieldDewPoint.setHorizontalAlignment(JTextField.***RIGHT***); // Sets the Alignment of the Text Box to the Right

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 8; // Sets the Row Constraint to 8

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(windChillResultLabel, con); // Adds Wind Chill Results Label to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(cloudBaseResultLabel, con); // Adds Cloud Base Results Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 15; // Sets the Row Constraint to 15

con.insets = **new** Insets(15, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***EAST***; // Aligns to the Left

WeatherPanel.add(computeButton, con); // Adds the "Compute" Button to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 15; // Sets the Row Constraint to 15

con.insets = **new** Insets(15, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(showFileDataButton, con); // Adds the "Show File Data" Button to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 12; // Sets the Column Constraint to 12

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(cloudTypeLabel, con); // Adds the Cloud Type Label to the Panel

con.gridx = 0; // Sets the Column Constraint to 0

con.gridy = 15; // Sets the Row Constraint to 15

con.insets = **new** Insets(15, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***WEST***; // Aligns to the Left

WeatherPanel.add(logoutButton, con); // Adds the "Logout" Button to the Panel

// sets a nice boarder for the panel

WeatherPanel.setBorder(BorderFactory.*createTitledBorder*(BorderFactory.*createEtchedBorder*(), "Weather Data Anyalsis Project - Meteorological Calculations"));

WeatherGUI.add(WeatherPanel); // Adds the Panel to the GUI

WeatherGUI.setResizable(**false**); // Makes User Unable to Resize the Window

WeatherGUI.setLocationRelativeTo(**null**); // Makes Initial Window Location Centered on the Screen

WeatherGUI.setVisible(**true**); // Makes the GUI visible

// Presumes that the user clicked on the "X" in the right corner of the window, not the "Exit" button

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); // When the main GUI window is closed, the default option is to stop/terminate the whole program

// Move the Main GUI to Appear In Front of the Weather Data Output Window

WeatherGUI.toFront();

// the following is a class that uses action listeners to implement functionality of the Compute button

**class** ComputeButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\nWeatherGUI class: Compute button was clicked.");

// get the inputs that the user entered into the text fields

String temp = textFieldTemperature.getText();

String wind = textFieldWindSpeed.getText();

String dew = textFieldDewPoint.getText();

// verify the wind chill; input validation for temperature, wind speed, and wind chill are within the method

**boolean** validWindChill = verifyWindChill(temp, wind);

// verify the cloud base; input validation for the dew point is within this method

**boolean** validCloudBase = verifyCloudBase(dew);

// if the wind chill and cloud base can be calculated, then calculate them and then display the data

**if**((validWindChill == **true**) && (validCloudBase == **true**))

{

// calculate the wind chill

**double** windChill = computeWindChill(temp, wind);

// calculate the cloud base

**double** cloudBase = computeCloudBase(temp, dew);

// system message to print out the results to two decimal places

System.***out***.println("\nWind Chill: " + String.*format*("%.2f", windChill) + "\tCloud Base: " + String.*format*("%.2f", cloudBase));

// display the data on the main GUI interface; calls a method that conducts this operation

displayDataMainGUI(windChill, cloudBase, con, rightImageFile2, rightImageFile3, rightImageFile4, rightImageLabel);

// update the separate data display output window

ddWindow.updateData(temp, wind, dew, windChill, cloudBase);

// update the plot graph display window

plotWindow.updateGraph(temp, windChill);

}

**else** // if the wind chill or cloud base can not be calculated, then it is invalid; do not display data

{

**if**(validWindChill == **false**) // if it is the wind chill that cannot be calculated

{

JOptionPane.*showMessageDialog*(**null**, "\n Invalid Wind Chill! \n Please enter a Temperature less than or equal to 50.0 degrees Fahrenheit, and a Wind Speed greater than 3.0 MPH. \n");

}

**if**(validCloudBase == **false**) // if it is the cloud base that cannot be calculated

{

JOptionPane.*showMessageDialog*(**null**, "\n Invalid Cloud Base! \n Please enter a valid dew point (practically any number). \n");

}

// reset the text boxes to be blank

textFieldTemperature.setText("");

textFieldWindSpeed.setText("");

textFieldDewPoint.setText("");

} // end of if else block

} // end of method

}; // end of ComputeButtonListener class

// Add the above ActionListener to the "Compute" button

computeButton.addActionListener(**new** ComputeButtonListener());

// the following class uses action listeners to implement functionality of the "Show File Data" button

**class** ShowFileDataListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\nWeatherGUI class: Show File Data button was clicked.");

ddWindow.openViewFileData(); // calls the openViewFileData method in the WeatherDataOutputWindow class

} // end of method

}; // end of ShowFileDataListener class

// Add the above Action Listener to the "Show File Data" button

showFileDataButton.addActionListener(**new** ShowFileDataListener());

// the following class uses action listeners to implement functionality of the "Logout" button

**class** LogoutButtonListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\nWeatherGUI class: Logout button was clicked.");

WeatherGUI.dispose(); // get rid of the main GUI interface of the program

ddWindow.disposeOfWindow(); // get rid of the separate data display output window

plotWindow.disposeOfWindow(); // get rid of the plot graph display window

callWeatherDataMain(); // calls the method

}

// this method calls the main method of WeatherData to bring up the startup dialog box

**public** **void** callWeatherDataMain() {

WeatherData.*main*(**null**);

}

}; // end of LogoutButtonListener class

// Add the above Action Listener to the "Logout" Button

logoutButton.addActionListener(**new** LogoutButtonListener());

} // end of if block

} // end of constructor

// verify wind chill method

**boolean** verifyWindChill(String temp, String wind) {

// declare boolean variables for verification

**boolean** goodTemp = **false**; // is the temperature valid?

**boolean** goodWindSpeed = **false**; // is the wind speed valid?

**boolean** tempDigits = **true**; // is the temperature only digits (numbers)?

**boolean** windSpeedDigits = **true**; // is the wind Speed only digits (numbers)?

**boolean** goodWindChill = **false**; // is the wind chill valid/calculable?

// algorithm

// if the temperature text field is blank

**if**(temp.contentEquals(""))

{

goodTemp = **false**; // temperature is invalid due to being blank

JOptionPane.*showMessageDialog*(**null**, "\n Temperature field is blank! \n Please enter a temperature. \n");

}

**else** // otherwise, move on to next verification process

{

**int** length = temp.length(); // get the length of the temperature string

// for each position of the temperature string, check if it is a digit

// if there are any letters or other characters other than digits (if there is something that is not a digit), set the variable to false

**for**(**int** i = 0; i < length; i++) {

**char** c = temp.charAt(i); // get the character

// if character is NOT a digit; so isDigit returns false

**if**(!(Character.*isDigit*(c)))

{

tempDigits = **false**;

}

**if**(temp.charAt(i) == 0x002E) // if there is a decimal point in the text box, let it go through and accept it

{

tempDigits = **true**;

}

**if**(temp.charAt(i) == 0x002D) // if there is a negative sign in the text box, let it go through an accept it

{

tempDigits = **true**;

}

} // end of for loop

// if the temperature is all digits (excluding a decimal point or negative sign), then temperature is good

**if**(tempDigits == **true**) {

goodTemp = **true**;

}

**else** // if there are non-digits in the temperature string

{

JOptionPane.*showMessageDialog*(**null**, "\n WARNING: Temperature field has non-digits in it! \n Please enter a valid temperature. \n");

}

} // end of if-else block for temperature verification

// if the wind speed text field is blank

**if**(wind.contentEquals(""))

{

goodWindSpeed = **false**; // wind speed is invalid due to being blank

JOptionPane.*showMessageDialog*(**null**, "\n Wind Speed field is blank! \n Please enter a wind speed. \n");

}

**else** // otherwise, move on to next verification process

{

**int** length = wind.length(); // get the length of the wind speed string

// for each position of the wind speed string, check if it is a digit

// if there are any letters or other characters other than digits (if there is something that is not a digit), set the variable to false

**for**(**int** i = 0; i < length; i++) {

**char** c = wind.charAt(i); // get the character

// if character is NOT a digit; so isDigit returns false

**if**(!(Character.*isDigit*(c)))

{

windSpeedDigits = **false**;

}

**if**(wind.charAt(i) == 0x002E) // if there is a decimal point in the text box, let it go through and accept it

{

windSpeedDigits = **true**;

}

} // end of for loop

// if the wind speed is all digits (excluding a decimal point), then wind speed is good

**if**(windSpeedDigits == **true**) {

goodWindSpeed = **true**;

}

**else** // if there are non-digits in the wind speed string

{

JOptionPane.*showMessageDialog*(**null**, "\n WARNING: Wind Speed field has non-digits in it! \n Please enter a valid wind speed. \n");

}

} // end of if-else block for wind speed verification

// if both the temperature and wind speed are valid (consists of only digits), then check if the wind chill can be calculated

**if**((goodTemp == **true**) && (goodWindSpeed == **true**))

{

// convert the temperature and wind speed strings into doubles

**double** doubTemp = Double.*parseDouble*(temp);

**double** doubWindSpeed = Double.*parseDouble*(wind);

// if the temperature entered is above 50.0 degrees Fahrenheit, or if the wind speed is less than or equal to 3.0 MPH, then the wind chill is NOT calculable (the wind chill is invalid)

**if**((doubTemp > 50.0) || (doubWindSpeed <= 3.0))

{

goodWindChill = **false**;

System.***out***.println("\nThe wind chill is not calculable!\n");

}

**else** // if the temperature and wind speeds are within the right ranges for a wind chill to occur

{

goodWindChill = **true**;

System.***out***.println("\nThe wind chill is calculable!\n");

}

}

**return** goodWindChill;

} // end of verifyWindChill method

// verify cloud base method

**boolean** verifyCloudBase(String dew) {

// boolean variables for verification

**boolean** dewDigits = **true**; // does the dew point consist of only digits?

**boolean** goodDew = **false**; // is the dew point valid?

**boolean** goodCloudBase = **false**; // is the cloud base valid?

// if the Dew Point field is blank

**if**(dew.contentEquals(""))

{

goodDew = **false**; // Dew Point is invalid due to being blank

JOptionPane.*showMessageDialog*(**null**, "\n Dew Point field is blank! \n Please enter a dew point. \n");

}

**else** // if the Dew Point field is not blank, then continue with the verification process

{

**int** length = dew.length(); // get the length of the dew point string

// for each position of the dew point string, check if it is a digit

// if there are any letters or other characters other than digits (if there is something that is not a digit), set the variable to false

**for**(**int** i = 0; i < length; i++) {

**char** c = dew.charAt(i); // get the character

// if character is NOT a digit; so isDigit returns false

**if**(!(Character.*isDigit*(c)))

{

dewDigits = **false**;

}

**if**(dew.charAt(i) == 0x002E) // if there is a decimal point in the text box, let it go through and accept it

{

dewDigits = **true**;

}

**if**(dew.charAt(i) == 0x002D) // if there is a negative sign in the text box, let it go through an accept it

{

dewDigits = **true**;

}

} // end of for loop

// if the dew point is all digits (excluding a decimal point or negative sign), then dew point is good

**if**(dewDigits == **true**) {

goodDew = **true**;

}

**else** // if there are non-digits in the dew point string

{

JOptionPane.*showMessageDialog*(**null**, "\n WARNING: Dew Point field has non-digits in it! \n Please enter a valid dew point. \n");

}

} // end of if-else block

// if dew point is valid, then cloud base is valid

**if**(goodDew == **true**)

{

goodCloudBase = **true**;

}

**return** goodCloudBase;

} // end of verifyCloudBase method

// compute wind chill method

**double** computeWindChill(String temp, String wind) {

// convert the temperature and wind speed strings into doubles

**double** doubTemp = Double.*parseDouble*(temp);

**double** doubWindSpeed = Double.*parseDouble*(wind);

// calculate the wind chill using the equation or formula

**double** windChill = 35.74 + (0.6215 \* doubTemp) - (35.75 \* (Math.*pow*(doubWindSpeed, 0.16))) + (0.4275 \* doubTemp \* (Math.*pow*(doubWindSpeed, 0.16)));

**return** windChill;

} // end of computeWindChill method

// compute cloud base method

**double** computeCloudBase(String temp, String dew) {

// convert the temperature and dew point strings into doubles

**double** doubTemp = Double.*parseDouble*(temp);

**double** doubDew = Double.*parseDouble*(dew);

// calculate cloud base using the equation or formula

**double** cloudBase = ((doubTemp - doubDew) / 4.4) \* 1000;

// the cloud base cannot be negative; if it is negative, set the cloud base to 0

**if**(cloudBase < 0) {

cloudBase = 0;

}

**return** cloudBase;

} // end of computeCloudBase method

// display data in Main GUI method

**void** displayDataMainGUI(**double** windChill, **double** cloudBase, GridBagConstraints con, File rightImageFile2, File rightImageFile3, File rightImageFile4, JLabel rightImageLabel) {

// convert the resultant wind chill and cloud base values from doubles into formatted strings (to two decimal places)

String windChillStr = String.*format*("%.2f", windChill);

String cloudBaseStr = String.*format*("%.2f", cloudBase);

// Concatenate the strings with their corresponding units (and format them to two decimal places)

String wCS = (windChillStr + " dF");

String cBS = (cloudBaseStr + " feet");

// Assign the text in the wind chill and cloud base labels to be the concatenated strings

windChillValueLabel.setText(wCS);

cloudBaseValueLabel.setText(cBS);

// Add the Labels to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 8; // Sets the Row Constraint to 8

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(windChillValueLabel, con); // Adds Wind Chill Value Label to the Panel

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 10; // Sets the Row Constraint to 10

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns Text to the Center

WeatherPanel.add(cloudBaseValueLabel, con); // Adds Cloud Base Value Label to the Panel

// Determine what level the cloud is based on its cloud base; then add the corresponding image and cloud type label to the Panel

**if**((cloudBase >= 0.00) && (cloudBase <= 7000.00)) // if the cloud base is between 0.00 feet and 7,000.00 feet, then it is considered a "low cloud"

{

// set the cloud type label as the text for the low cloud type and add it to the Panel

cloudTypeValueLabel.setText("Low-Level Cloud (0 ft. to 7,000 ft.)");

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 12; // Sets the Column Constraint to 12

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(cloudTypeValueLabel, con); // Adds the Cloud Type Label to the Panel

// Create a BufferedImage for the "LowClouds.jpg" image file for the right column of the panel, then add it to the panel by replacing the original image

**try** {

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 0; // Sets the Row Constraint to 0

con.anchor = GridBagConstraints.***NORTH***; // Aligns Image to the North (Top)

BufferedImage rightImage4 = ImageIO.*read*(rightImageFile4); // makes the image file into a useable image

rightImageLabel.setIcon(**new** ImageIcon(rightImage4)); // sets the image as the "Icon" for the "rightImageLabel" variable

}

**catch**(Exception e) { // if image is not found/does not exist/can't be read by ImageIO.read

JOptionPane.*showMessageDialog*(**null**, "\nLowClouds.jpg image file not Found!\n"); // dialog box

System.***out***.println("\nLowClouds.jpg image file not found!\n"); // system error message

e.printStackTrace();

}

} // end of block for low cloud

**else** **if**((cloudBase > 7000.00) && (cloudBase <= 17000.00)) // if the cloud base is between 7,000.00 feet and 17,000.00 feet, then it is considered a "medium-level cloud"

{

// set the cloud type label as the text for the medium cloud type and add it to the Panel

cloudTypeValueLabel.setText("Medium-Level Cloud (7,000 ft. to 17,000 ft.)");

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 12; // Sets the Column Constraint to 12

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(cloudTypeValueLabel, con); // Adds the Cloud Type Label to the Panel

// Create a BufferedImage for the "MediumClouds.jpg" image file for the right column of the panel, then add constraints and add it to the panel

**try** {

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 0; // Sets the Row Constraint to 0

con.anchor = GridBagConstraints.***NORTH***; // Aligns Image to the North (Top)

BufferedImage rightImage3 = ImageIO.*read*(rightImageFile3); // makes the image file into a useable image

rightImageLabel.setIcon(**new** ImageIcon(rightImage3)); // sets the image as the "Icon" for the "rightImageLabel" variable

}

**catch**(Exception e) { // if image is not found/does not exist/can't be read by ImageIO.read

JOptionPane.*showMessageDialog*(**null**, "\nMediumClouds.jpg image file not Found!\n"); // dialog box

System.***out***.println("\nMediumClouds.jpg image file not found!\n"); // system error message

e.printStackTrace();

}

} // end of block for medium cloud

**else** **if**((cloudBase > 17000.00) && (cloudBase <= 35000.00)) // if the cloud base is between 17,000.00 feet and 35,000.00 feet, then it is considered a "high cloud"

{

// set the cloud type label as the text for the high cloud type and add it to the Panel

cloudTypeValueLabel.setText("High-Level Cloud (17,000 ft. to 35,000 ft.)");

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 12; // Sets the Column Constraint to 12

con.insets = **new** Insets(20, 1, 1, 1); // Top, Left, Bottom, then Right - Padding

con.anchor = GridBagConstraints.***CENTER***; // Aligns to the Center

WeatherPanel.add(cloudTypeValueLabel, con); // Adds the Cloud Type Label to the Panel

// Create a BufferedImage for the "HighClouds.jpg" image file for the right column of the panel, then add constraints and add it to the panel

**try** {

con.gridx = 1; // Sets the Column Constraint to 1

con.gridy = 0; // Sets the Row Constraint to 0

con.anchor = GridBagConstraints.***NORTH***; // Aligns Image to the North (Top)

BufferedImage rightImage2 = ImageIO.*read*(rightImageFile2); // makes the image file into a useable image

rightImageLabel.setIcon(**new** ImageIcon(rightImage2)); // sets the image as the "Icon" for the "rightImageLabel" variable

}

**catch**(Exception e) { // if image is not found/does not exist/can't be read by ImageIO.read

JOptionPane.*showMessageDialog*(**null**, "\nHighClouds.jpg image file not Found!\n"); // dialog box

System.***out***.println("\nHighClouds.jpg image file not found!\n"); // system error message

e.printStackTrace();

}

} // end of block for high cloud

// refresh the panel to update the data and images

WeatherPanel.revalidate();

WeatherPanel.repaint();

} // end of displayDataMainGUI method

}; // end of WeatherGUI class

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherDataOutputWindow.java

// This class creates and holds the functionality of the separate data output window for the program.

**package** weatherPackage;

//imports

**import** javax.swing.filechooser.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**import** java.text.\*;

**import** java.util.\*;

**import** java.awt.\*;

**import** java.io.\*;

**public** **class** WeatherDataOutputWindow **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// Create the Window's GUI Frame

JFrame ddWin = **new** JFrame("Weather Data Output Window");

// Create the Window's GUI Panel

JPanel ddPanel = **new** JPanel();

// Create the Data Output Text Area

JTextArea dataOutputTextArea = **new** JTextArea(40, 95); // 40 rows, 95 columns

// Create A Vertical Scroll Bar for the Side of the dataOutputTextArea

JScrollPane dataWindowScroll = **new** JScrollPane(dataOutputTextArea, JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***, JScrollPane.***HORIZONTAL\_SCROLLBAR\_NEVER***);

// String that Holds the Data Output (Used in the updateData method)

String dataOutputString = ("");

// Constructor with Configurations

**public** WeatherDataOutputWindow() {

// Set the Size and Background Color for the Window and Panel

ddWin.setSize(733, 308);

ddPanel.setSize(733, 308);

ddPanel.setBackground(Color.***white***);

// Create the headers' text areas and the headers themselves

JTextArea titleHeaderTextArea = **new** JTextArea(1, 18); // 1 row, 18 columns

JTextArea labelsHeaderTextArea = **new** JTextArea(0, 95); // 0 rows after the header text, 95 columns

String titleHeader = ("\n Weather Ouput Data\n");

String labelsHeader = (" Temperature Wind Speed Dew Point Wind Chill Cloud Base Cloud Type" + "\n-----------------------------------------------------------------------------------------------");

// Set the Font and Text for the Title Header Text Area, and Then Add the Title Header Text Area to the Panel

titleHeaderTextArea.setFont(**new** Font("Source Code Pro", Font.***BOLD***, 20));

titleHeaderTextArea.setText(titleHeader);

titleHeaderTextArea.setEditable(**false**); // Makes It So The Text In The Header Can't Be Edited

ddPanel.add(titleHeaderTextArea, BorderLayout.***CENTER***);

// Set the Font and Text for the Labels Header Text Area, and Then Add the Labels Header Text Area to the Panel

labelsHeaderTextArea.setFont(**new** Font("Consolas", Font.***BOLD***, 12));

labelsHeaderTextArea.setText(labelsHeader);

labelsHeaderTextArea.setEditable(**false**); // Makes It So The Text In The Header Can't Be Edited

ddPanel.add(labelsHeaderTextArea, BorderLayout.***CENTER***);

// Set the Preferred Size for the Scroll Bar and the Font for the Data Output Text Area, and Add the Scroll Bar To the Panel (The Scoll Bar contains the Text Area Where the Columned Data Will Go)

dataOutputTextArea.setFont(**new** Font("Consolas", Font.***PLAIN***, 12));

dataOutputTextArea.setEditable(**false**); // Makes It So The Data Output Can't Be Edited

dataWindowScroll.setPreferredSize(**new** Dimension(664, 110)); // Sets the Size of the Scroll Bar/dataOutputTextArea

dataWindowScroll.setBorder(BorderFactory.*createEmptyBorder*(0, 0, 0, 0)); // Sets the Border of the Scroll Bar

ddPanel.add(dataWindowScroll, BorderLayout.***CENTER***); // Add the Scroll Bar to the Side of the dataOutputTextArea

// Set the Font for the Panel

ddPanel.setFont(**new** Font("Consolas", Font.***PLAIN***, 12));

// Add a Menu Bar to the Window for File Saving Functionality

JMenuBar menuBar = **new** JMenuBar();

JMenu fileMenu = **new** JMenu("File");

JMenuItem saveAsItem = **new** JMenuItem("Save As");

fileMenu.add(saveAsItem); // Adds the Item to the Menu

menuBar.add(fileMenu); // Adds the Dropdown Menu to the Menu Bar of the Window

// the following is a class that uses action listeners to detect if the user selected the "save as" item in the menu bar and conduct the "save as" function

**class** MenuItemListener **implements** ActionListener {

// method to capture action/event

**public** **void** actionPerformed(ActionEvent e)

{

System.***out***.println("\n'Save As' Was Selected.");

performSaveAs(labelsHeader); // calls the method that performs the 'Save As' Functionality

} // end of method

}; // end of MenuItemListener class

// Adds the above Action Listener to the 'Save As' Menu Item

saveAsItem.addActionListener(**new** MenuItemListener());

// Adds the Menu Bar to the Window

ddWin.setJMenuBar(menuBar);

ddWin.add(ddPanel); // Adds the Panel to the Frame

ddWin.setVisible(**true**); // Makes the Window visible

ddWin.setResizable(**false**); // Makes User Unable to Resize the Window

ddWin.setLocation(200, 200); // Makes Initial Window Location Slightly Offset from the Main GUI's Location

// Presumes that the user clicked on the "X" in the right corner of the window

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); // When the window is closed, the default option is to stop/terminate the whole program

} // end of constructor

// update data method

**public** **void** updateData(String temp, String wind, String dew, **double** windChill, **double** cloudBase) {

// Declare a StringBuilder to Make Concadenation of the Data Output Easier

StringBuilder dataString = **new** StringBuilder();

String item = ("");

// convert the temperature, wind speed, and dew point strings into doubles

**double** doubTemp = Double.*parseDouble*(temp);

**double** doubWindSpeed = Double.*parseDouble*(wind);

**double** doubDew = Double.*parseDouble*(dew);

DecimalFormat df = **new** DecimalFormat("#,##0.00"); // Sets the Formatting for the Data

// for loop that checks the data and sees what piece it is (ex. temp or windChill) to add the appropiate units

**for**(**int** count = 1; count < 6; count++)

{

// these if statements actually check what the data is and adds its units

**if**(count == 1) // this is the temp

{

item = df.format(doubTemp);

item = item + " dF ";

item = formatItem(item, count); // this method formats the data by adding spaces before placing it on the dataOutputTextArea

dataString.append(item);

}

**if**(count == 2) // this is the wind (speed)

{

item = df.format(doubWindSpeed);

item = item + " MPH ";

item = formatItem(item, count);

dataString.append(item);

}

**if**(count == 3) // this is the dew (point)

{

item = df.format(doubDew);

item = item + " dF";

item = formatItem(item, count);

dataString.append(item);

}

**if**(count == 4) // this is the windChill

{

item = df.format(windChill);

**if**(item.charAt(0) == '-') // if the windChill is negative

{

item = item + " dF ";

}

**else** // otherwise, if the windChill is not negative

{

item = " " + item + " dF ";

}

item = formatItem(item, count);

dataString.append(item);

}

**if**(count == 5) // this is the cloudBase

{

item = df.format(cloudBase);

item = item + " ft";

item = formatItem(item, count);

dataString.append(item);

}

} // end of the for loop

// Determines what level the cloud is based on its cloud base; then adds the corresponding cloud type to the dataString

**if**((cloudBase >= 0.00) && (cloudBase <= 7000.00)) // if the cloud base is between 0.00 feet and 7,000.00 feet, then it is considered a "low cloud"

{

**int** count = 6; // sets the count for the formatItem method

item = (""); // Reset the item string

// set the item string text for a low cloud, format it, and append it to the dataString

item = item + "Low ";

item = formatItem(item, count);

dataString.append(item);

} // end of block for low cloud

**else** **if**((cloudBase > 7000.00) && (cloudBase <= 17000.00)) // if the cloud base is between 7,000.00 feet and 17,000.00 feet, then it is considered a "medium cloud"

{

**int** count = 7; // sets the count for the formatItem method

item = (""); // Reset the item string

// set the item string text for a medium cloud, format it, and append it to the dataString

item = item + "Medium ";

item = formatItem(item, count);

dataString.append(item);

} // end of block for medium cloud

**else** **if**((cloudBase > 17000.00) && (cloudBase <= 35000.00)) // if the cloud base is between 17,000.00 feet and 35,000.00 feet, then it is considered a "high cloud"

{

**int** count = 8; // sets the count for the formatItem method

item = (""); // Reset the item string

// set the item string text for a high cloud, format it, and append it to the dataString

item = item + "High ";

item = formatItem(item, count);

dataString.append(item);

} // end of block for high cloud

dataString.append("\n"); // Add a Line Feed

// Create or Append the Empty String to Create One Long Line of Data

dataOutputString = dataOutputString + dataString.toString();

// Add the Long Line of Data to the Data Output Text Area

dataOutputTextArea.setText(dataOutputString);

} // end of updateData method

// format item method

**public** String formatItem(String item, **int** count)

{

**int** stringLength = item.length(); // gets the length of the String item

**int** spaces = 0;

// Each column width is a different length (as in it is not uniform), so these if statements will place the appropiate amount of spaces within the string for the unique width of each column

**if**(count == 1) // this is the temp, whose column width is 21, but it is 16 if you go to the end of the header text

{

spaces = 16 - stringLength;

}

**else** **if**(count == 2) // this is the wind (speed), whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

**else** **if**(count == 3) // this is the dew (point), whose column width is 19, but it is 14 if you go to the end of the header text

{

spaces = 14 - stringLength;

}

**else** **if**(count == 4) // this is the windChill, whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

**else** **if**(count == 5) // this is the cloudBase, whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

**else** **if**(count == 6) // this is the Low Cloud Type, whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

**else** **if**(count == 7) // this is the Medium Cloud Type, whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

**else** **if**(count == 8) // this is the High Cloud Type, whose column width is 20, but it is 15 if you go to the end of the header text

{

spaces = 15 - stringLength;

}

String spacing = (""); // String variable for the following for loop

// This for loop incorporates the spaces into the string

**for**(**int** i = 0; i < spaces; i++)

{

spacing = spacing + " ";

}

// append the amount of spaces to the string

spacing = spacing + item;

item = spacing;

**return** item;

} // end of formatItem method

// dispose of window method

**public** **void** disposeOfWindow() {

ddWin.dispose();

} // end of disposeOfWindow method

// file saving method

**public** **void** performSaveAs(String labelsHeader) {

// Declare a JFileChooser for the user to choose a text file to save the output window's contents to

File workingDir = **new** File(System.*getProperty*("user.dir"));

JFileChooser fileChooser = **new** JFileChooser();

fileChooser.setDialogTitle("Specify a Text File (.txt) to Save To");

fileChooser.setCurrentDirectory(workingDir);

// declare a filter for a specific file extension: .txt

FileNameExtensionFilter filter = **new** FileNameExtensionFilter("txt", "TXT");

fileChooser.setFileFilter(filter);

**int** userSelection = fileChooser.showSaveDialog(**null**); // no parent frame/component

**if**(userSelection == JFileChooser.***APPROVE\_OPTION***) // if user chooses to save to a file

{

// get the filename of the selected file

File fileToSave = fileChooser.getSelectedFile();

System.***out***.println("Save As File: " + fileToSave.getAbsolutePath());

// Now we write the text that is in the JTextArea to the File

**try** // try to see if file exists; if not, exception is thrown and handled

{

FileWriter fileWrite = **new** FileWriter(fileToSave);

BufferedWriter buffWrite = **new** BufferedWriter(fileWrite);

PrintWriter printWrite = **new** PrintWriter(buffWrite);

printWrite.println(labelsHeader + "\n" + dataOutputString);

// close the PrintWriter

printWrite.close();

}

**catch**(IOException e) { // if FileWriter throws an exception, as in the file does not exist but cannot be created or the file cannot be opened

e.printStackTrace();

System.***out***.println("\nThe file you are looking to save to either: does not exist but cannot be created or it cannot be opened.\n");

} // end of try-catch block

}

**else** // if user does not select a file to save to

{

System.***out***.println("File Saving Canceled: NO FILE SELECTED!");

}

} // end of performSaveAs method

// open and view saved data method

**public** **void** openViewFileData() {

// Declare a JFileChooser for the user to choose a text file to open and view the contents of in the output window

File workingDir = **new** File(System.*getProperty*("user.dir"));

JFileChooser fileChooser = **new** JFileChooser();

fileChooser.setDialogTitle("Specify a Text File (.txt) to Open and View");

fileChooser.setCurrentDirectory(workingDir);

// declare a filter for a specific file extension: .txt

FileNameExtensionFilter filter = **new** FileNameExtensionFilter("txt", "TXT");

fileChooser.setFileFilter(filter);

**int** userSelection = fileChooser.showOpenDialog(**null**); // no parent frame/component

// Now we read the data from the File and write it to the JTextArea

**try** { // try to see if file exists; if not, exception is thrown and handled

**if**(userSelection == JFileChooser.***APPROVE\_OPTION***) // if user selects a file to open

{

// get the filename of the selected file

File fileToOpen = fileChooser.getSelectedFile();

System.***out***.println("\nOpening File: " + fileToOpen.getAbsolutePath());

Scanner in = **new** Scanner(fileToOpen);

StringBuilder data = **new** StringBuilder();

// loop that gets the data from the file

**while**(in.hasNext()) // while the end of the file has not be reached

{

// temportary variables to get the full header into this method for use to compare to the lines of data in the file

String header1 = (" Temperature Wind Speed Dew Point Wind Chill Cloud Base Cloud Type");

String header2 = ("-----------------------------------------------------------------------------------------------");

data.append(in.nextLine()); // get the next line from the file

String dataString = data.toString(); // convert the read line to a String

**if**(dataString.equals(header1)) // if the String line that was read is the first part of the header, the labels

{

data.delete(0, 90); // deletes or remove the header from the stringBuilder

}

**else** **if**(dataString.equals(header2)) // if the String line that was read is the second part of the header, the dashed line

{

data.delete(0, 105); // deletes or remove the header from the stringBuilder

}

**else** // if the String line that was read is the actual data

{

// Append the dataOutputString to Create One Long Line of Data

dataOutputString = dataOutputString + dataString + "\n";

// Add the Long Line of Data to the Data Output Text Area

dataOutputTextArea.setText(dataOutputString);

}

} // end of while loop

in.close(); // close the Scanner

}

**else** // if user does not select a file to open

{

System.***out***.println("\nFile Opening Canceled: NO FILE SELECTED!");

} // end of if-else block

}

**catch**(FileNotFoundException e) { // if Scanner throws an exception, as in the file can not be found or opened

JOptionPane.*showMessageDialog*(**null**, "\n The file you are looking to open cannot be found or opened. \n");

System.***out***.println("\nThe file you are looking to open cannot be found or opened.\n");

e.printStackTrace();

} // end of try-catch block

} // end of openViewFileData method

}; // end of WeatherDataOutputWindow class

// Gregory Zacharko

// CSE 223: Spring 2021 Semester - Professor Simber

// WeatherPlotGraphDisplayWindow.java

// This class creates and holds the functionality of the plot graph display window for the program.

**package** weatherPackage;

// imports

**import** java.awt.Color;

**import** javax.swing.\*;

**import** java.util.\*;

**import** java.awt.\*;

**public** **class** WeatherPlotGraphDisplayWindow **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// Create the Plot Window's GUI Frame

JFrame plotGraphWin = **new** JFrame("Weather Data Scatter Plot Graph Window");

// Create an ArrayList to contain the data points

ArrayList<Integer> dataList = **new** ArrayList<Integer>();

// Constructor with Configurations

**public** WeatherPlotGraphDisplayWindow() {

// Set the Size and Background Color for the Window

plotGraphWin.setSize(710, 500); // Width, Height

plotGraphWin.setBackground(Color.***WHITE***);

// Create New JComponent

JComponent comp = **new** JComponent() {

**private** **static** **final** **long** ***serialVersionUID*** = 1;

// method that paints the graphics for the component

**public** **void** paintComponent(Graphics graphic) {

**super**.paintComponent(graphic);

// create and set the fonts for the title and the text of the graph

Font titleFont = **new** Font("Source Code Pro", Font.***BOLD***, 20);

Font textFont = **new** Font("Consolas", Font.***BOLD***, 12);

graphic.setFont(titleFont);

graphic.setColor(Color.***BLACK***);

graphic.drawString("Temperature and Wind Chill Scatter Plot Graph", 137, 33); // draw the title

graphic.setFont(textFont);

// this is the y-axis of the graph

graphic.drawLine(50, 450, 50, 50);

graphic.drawLine(51, 450, 51, 50);

// labels for the y-axis

graphic.drawString("-30", 20, 450);

graphic.drawString("-20", 20, 405);

graphic.drawString("-10", 20, 355);

graphic.drawString("0", 33, 305);

graphic.drawString("10", 27, 255);

graphic.drawString("20", 27, 205);

graphic.drawString("30", 27, 155);

graphic.drawString("40", 27, 105);

graphic.drawString("50", 27, 55);

// this is the x-axis of the graph

graphic.drawLine(50, 300, 680, 300);

graphic.drawLine(50, 301, 680, 301);

// set the color for the horizontal major axis lines (the lines indictating each increment of 10)

graphic.setColor(Color.***DARK\_GRAY***);

// Horizontal Major Axis Lines

graphic.drawLine(50, 450, 680, 450);

graphic.drawLine(50, 400, 680, 400);

graphic.drawLine(50, 350, 680, 350);

graphic.drawLine(50, 300, 680, 300);

graphic.drawLine(50, 250, 680, 250);

graphic.drawLine(50, 200, 680, 200);

graphic.drawLine(50, 150, 680, 150);

graphic.drawLine(50, 100, 680, 100);

graphic.drawLine(50, 50, 680, 50);

// set the color for the horizontal minor axis lines (the lines indictating the increments of 5 inbetween each major axis line)

graphic.setColor(Color.***LIGHT\_GRAY***);

// Horizontal Minor Axis Lines

graphic.drawLine(50, 425, 680, 425);

graphic.drawLine(50, 375, 680, 375);

graphic.drawLine(50, 325, 680, 325);

graphic.drawLine(50, 275, 680, 275);

graphic.drawLine(50, 225, 680, 225);

graphic.drawLine(50, 175, 680, 175);

graphic.drawLine(50, 125, 680, 125);

graphic.drawLine(50, 75, 680, 75);

// x coordinate starting point

**int** xCoord = 65;

// for loop that actually plots the points

**for**(**int** i = 0; i < dataList.size(); i++)

{

**int** temp = dataList.get(i); // gets the first item

**int** tempHeight = temp \* 5;

graphic.setColor(Color.***RED***); // sets the color for these temperature points to be red

graphic.drawString(temp + " dF", xCoord, 300 - tempHeight - 5);

graphic.fillOval(xCoord, 300 - tempHeight, 5, 5);

**int** windChill = dataList.get(i + 1); // gets the next item

**int** windChillHeight = windChill \* 5;

graphic.setColor(Color.***BLUE***); // sets the color for these wind chill points to be blue

graphic.drawString(windChill + " dF", xCoord, 300 - windChillHeight - 5);

graphic.fillOval(xCoord, 300 - windChillHeight, 5, 5);

i = i + 1;

xCoord = xCoord + 45;

} // end of for loop

} // end of paintComponent block

}; // end of JComponent block

plotGraphWin.add(comp); // Add JComponent to the Frame

plotGraphWin.setVisible(**true**); // Makes the Window visible

plotGraphWin.setResizable(**false**); // Makes User Unable to Resize the Window

plotGraphWin.setLocation(300, 325); // Makes Initial Window Location Slightly Offset from the Main GUI's and Output Window's Location

// Presumes that the user clicked on the "X" in the right corner of the window

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***); // When the window is closed, the default option is to stop/terminate the whole program

} // end of constructor

// update plot graph method

**public** **void** updateGraph(String temp, **double** windChill) {

// convert the temperature string into an double

**double** doubleTemp = Double.*parseDouble*(temp);

// add the temperature and wind chill to the ArrayList as integers

dataList.add((**int**)doubleTemp);

dataList.add((**int**)windChill);

// call the repaint method

plotGraphWin.repaint();

} // end of updateGraph method

// dispose of window method

**public** **void** disposeOfWindow() {

plotGraphWin.dispose();

} // end of disposeOfWindow method

}; // end of WeatherPlotGraphDisplayWindow class