**Institute of Technology Tralee**

**Computing Department**

**Structured Programming 2**

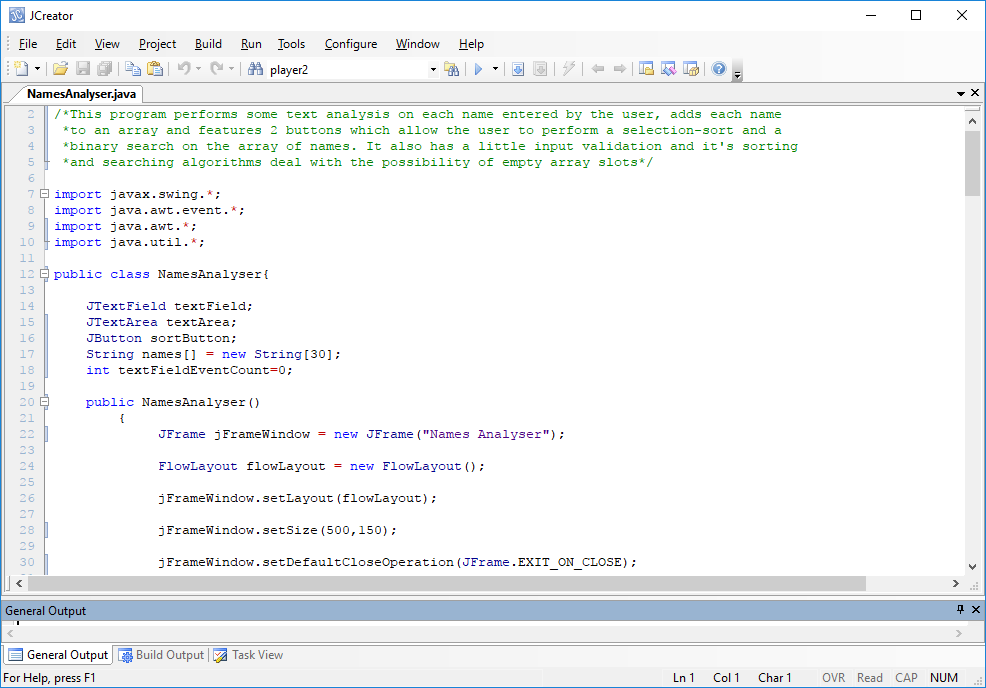
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

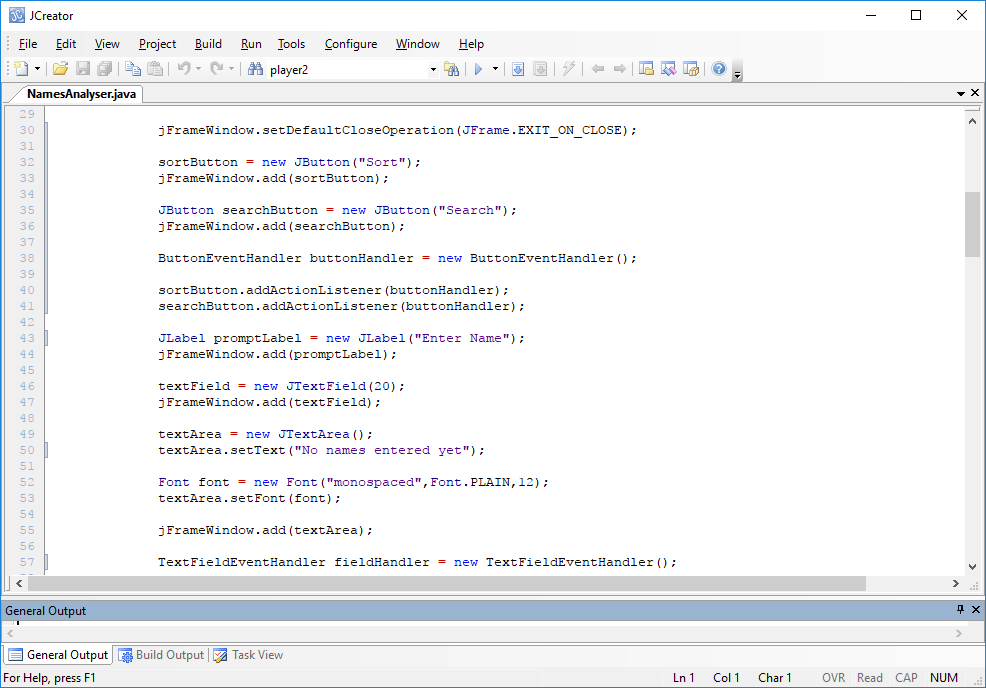
**Practical 16 – GUIs and Arrays**

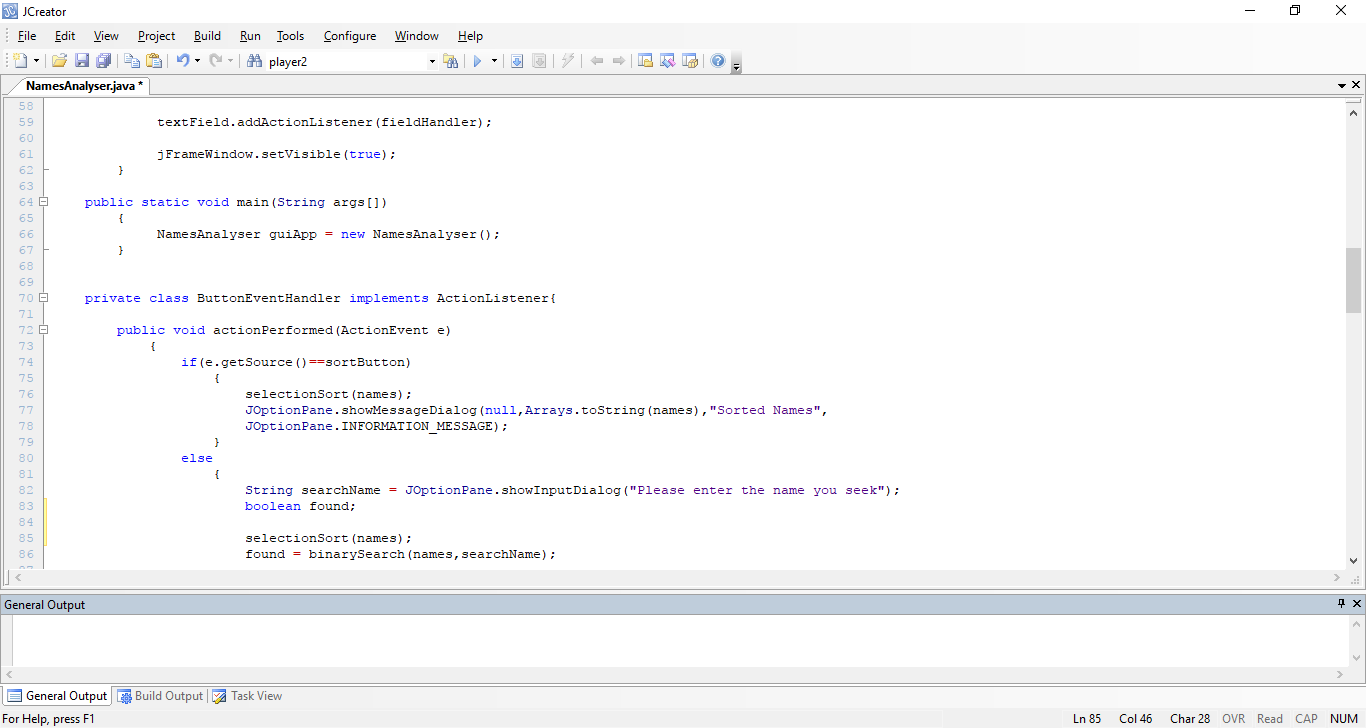
In this last lab sheet of the semester, we bring together the two very important topics that have absorbed most of our time over the last few weeks, GUIs and arrays. We will look at some sample programs that illustrate how they may be combined to useful effect. You will also get the chance to attempt a few exercises based on both topics in combination and these should be good preparation for the module’s final exam.

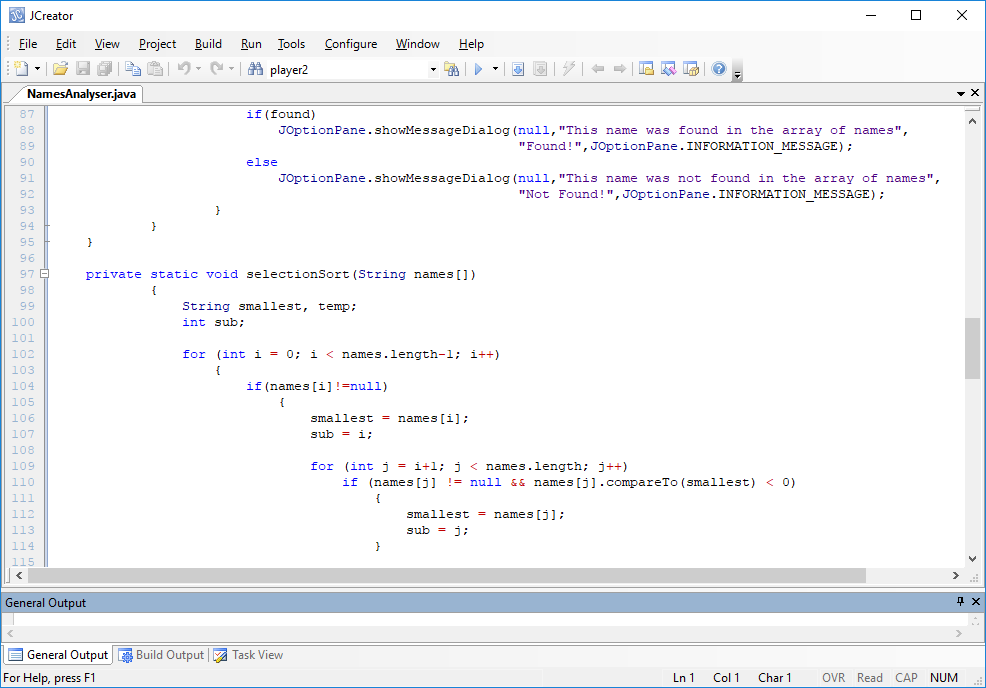
**Aim:** This program takes in an arbitrary number of names via a text-field, adds each name to an array, performs some simple validation and, once valid, performs text analysis on each one, as they are entered. The GUI also features two buttons that allow the user to perform a selection-sort and binary search respectively on the array of names – note the **code for most of the program is already available to you** in the LabSheet16 folder

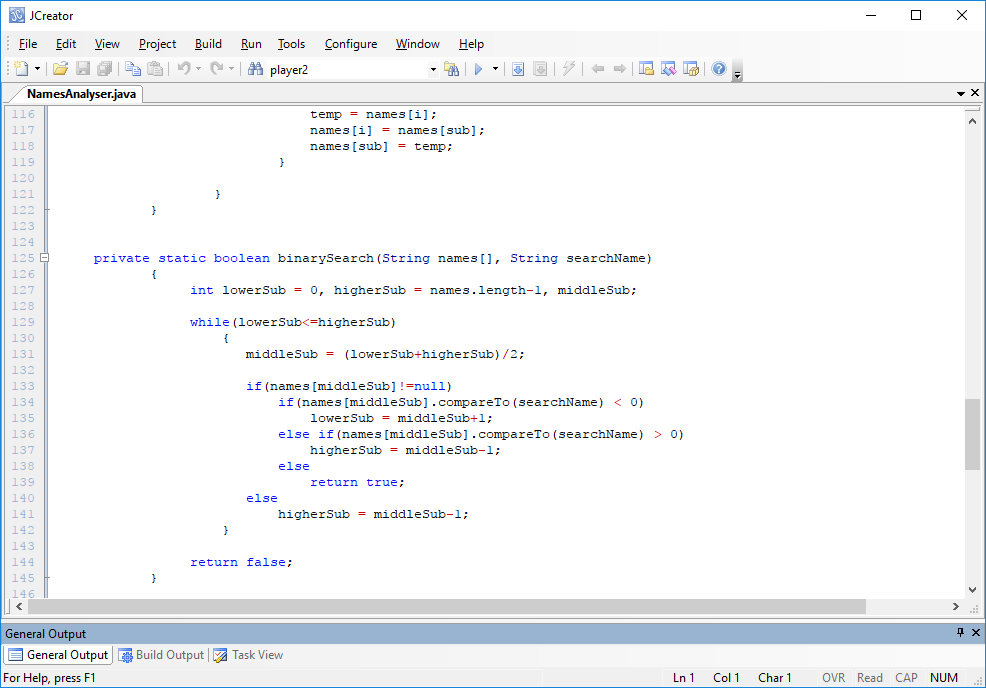
**Java Code:**

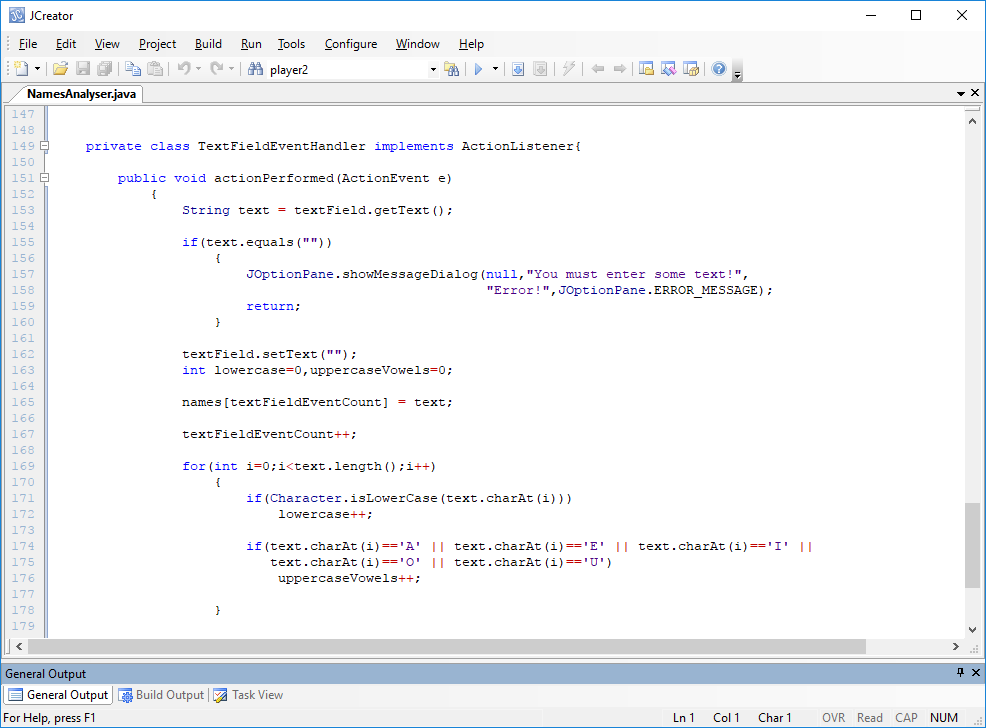


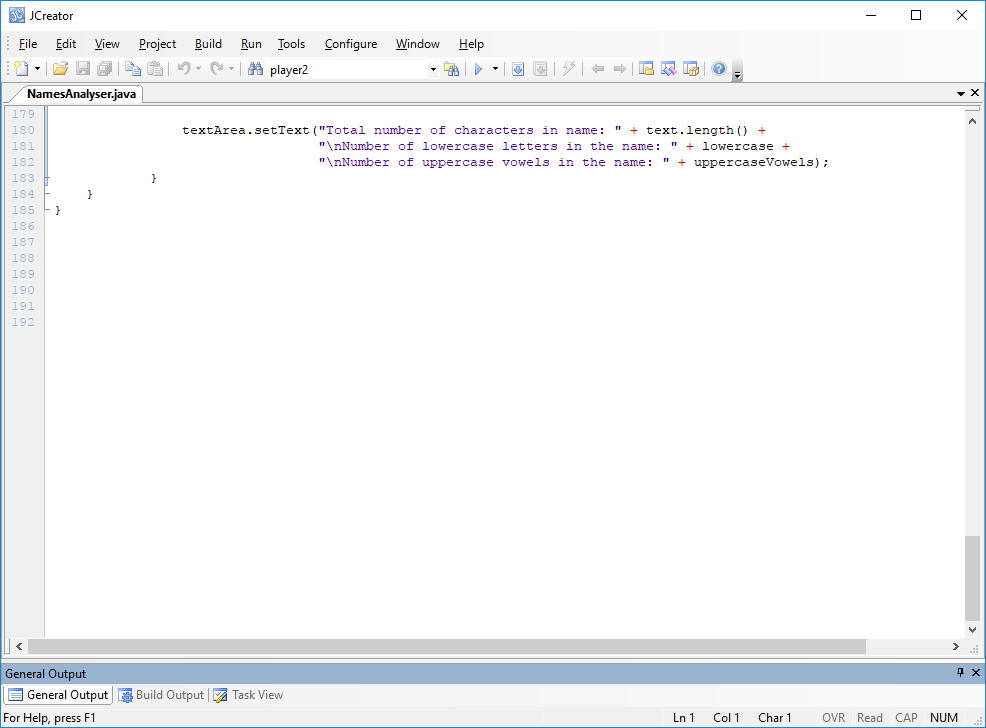












**Program Analysis:**

● The program begins by importing all the necessary packages. Recall that the **Arrays** class is part of the **java.util** package and we make use of that class in this program.

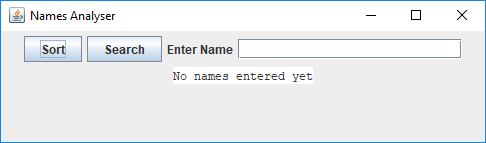
● There are a number of **globals** defined at the start of the NamesAnalyser class. These are defined outside of any method so that they can be **accessed from anywhere** within the class definition. For example, the reference **textField** is defined here because, when the application is running, we will need to be able to read information from the text-field when the user hits return on it. Similarly, the reference **sortButton** is defined here because, when the application is running, we need to be able to determine, when a button event occurs, whether the “sort” button or the “search” button was the source of the event so the appropriate course of action can ensue. The **textArea** reference is yet another global reference to a GUI component, needed so that text can be written to the text-area at runtime.

● You can also see that there is an array of String of size 30 called **names** declared as a global. This array is central to the program’s operation and, each time the user enters a name into the text-field and hits return on it, that name will be added to the array. In this case we don’t know how many names will be entered at runtime, so the array is only going to be partially filled – it will be completely empty at the start, with all of its slots set to the **default value** of **null**. This does complicate things a little bit for us later, as you will see, compared to the “normal” situation where we have a fully populated array, but some complexity is good when it comes to learning ☺

● You see one final global called **textFieldEventCount** and it is given an initial value of zero. This global is there to keep track of the number of text-field events that occur at runtime. This is necessary because we want to add a newly entered name to the array whenever the user hits return on the text-field. This counter will be increased every time a new valid name is entered by the user (in this case a valid name is basically anything at all as long as the user has entered something, otherwise they will get an error message and nothing will be added to the array in that case).

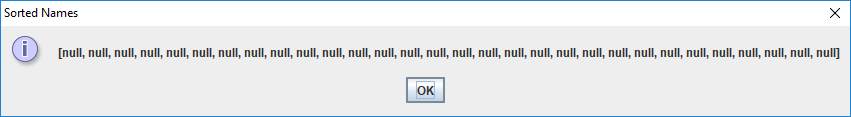
● Next, the GUI constructor method is defined and you can see the “typical” GUI construction code going on. There is nothing really new in this method as such, although you can see that there are two different event-handling classes defined for the application – one for the text-field called **TextFieldEventHandler** and another for the two buttons on the GUI called **ButtonEventHandler** (I’m nothing if I amn’t imaginative with my identifier names ☺). Strictly speaking, I could have gotten away with just a single event-handling class here, but it is good for you to see that an application can have several of them.

● So, when the GUI constructor executes at runtime you see the following



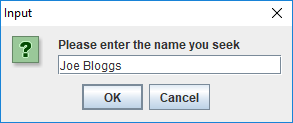
The “**Sort**” button, when pressed, will display the sorted contents of the array of names on a message dialog through a **selection-sort** algorithm. The “**Search**” button, when pressed, will display an input dialog and give the user the chance to enter a name and, when the user hits OK on the input dialog, the binary search algorithm will search the (sorted) array of names for the name entered. At launch time, since there have been no names entered yet, the text-area just displays the text “No names entered yet”.

At this point, should you press the “Sort” button you will get the following output:

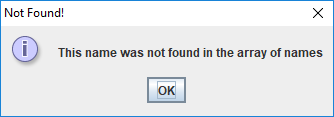


This is just displaying the value stored in every slot of the array of names. As Strings are objects in Java, every slot receives the default value of **null**, as mentioned earlier, and this is what gets displayed for all 30 slots.

Pressing the “Search” button at this point will display an input dialog prompting the user to enter a name



but, since the array has no names stored in it yet, the user will always get the reply



● Within the ButtonEventHandler private inner class’s **actionPerformed**() method, you can see the code

**if(e.getSource()==sortButton)**

**{**

**selectionSort(names);**

**JOptionPane.showMessageDialog(null,Arrays.toString(names),"Sorted Names",**

**JOptionPane.INFORMATION\_MESSAGE);**

**}**

This is checking to see whether the source of the event that just occurred was the “Sort” button. If it was, then a call is made to the **selectionSort**() method, whose job is to perform the selection-sort algorithm on the names array it is passed and then display the contents of the sorted array on a message dialog.

● The code:

**else**

**{**

**String searchName = JOptionPane.showInputDialog("Please enter**

**the name you seek");**

**boolean found;**

**selectionSort(names);**

**found = binarySearch(names,searchName);**

executes if the source of the event was the “**Search**” button instead. In this case, the user is prompted to enter a name, the array of names is sorted into ascending order via **selectionSort**() and then the name entered is passed to the **binarySearch**() method along with the names array. Should the method return the value true, then the name was found in the array and an appropriate message displays on a dialog, otherwise a “not found” message appears.

● The code for the **selectionSort**() method appears next. The code used for the algorithm in this case is very similar to what you have seen before with some small differences. The test code

**if(names[i]!=null)**

appears at the beginning of the outer for loop. This code is essential to the operation of the algorithm in this case because the array may not be full (in fact at the start it is empty). Without this test to see if the first element of a particular subarray is not null, then the application will usually crash at runtime if the “Sort” button gets pressed, since the chances are the array will have some empty slots when the sort is performed, leading to a so-called **nullPointerException**.

In a similar fashion, the code

**if (names[j] != null && names[j].compareTo(smallest) < 0)**

at the beginning of the inner for loop first of all checks to see whether the slot of the names array at position j is empty. If it is, then the remainder of the if test expression and the code associated with the if test is completely bypassed, again avoiding the **nullPointerException**.

If, however, both the slots at position i and position j are not null (i.e. not empty) then the algorithm proceeds in the same manner as usual, tracking the smallest element in each successive subarray and then swapping the first element in the subarray in question with the smallest element found in that subarray.

● One other thing to mention about the algorithm is the code

**names[j].compareTo(smallest) < 0**

within the inner for loop if test expression. Recall that the names array is a String array here so we are trying to look for the smallest String within a given subarray. As you learned in an earlier lab sheet, we **cannot use the less than operator** < to check to see whether one String is “less than” another String alphabetically – instead we must use the **compareTo**() method to get the job done. If the method returns a value less than zero in this case then it must be that the String names[j] must be “alphabetically less than” the String smallest.

● The code for the **binarySearch**() method appears next. This code is very similar to what you have seen already but, again, there are a few small differences to account for the fact that we are dealing with an array that is likely to contain empty slots when the search is performed. The code

**if(names[middleSub]!=null)**

is there to make sure that the String with subscript number middleSub is not null. If it turns out to be not null then it means that there is an actual string value stored at that slot of the array and the search algorithm can proceed as normal. However, if it is null then the corresponding else section executes and the code

**higherSub = middleSub-1;**

runs instead. This code changes the value of higherSub should it turn out that the value located at middleSub is null. If this is the case, then we take it that all slots beyond that slot are also null, so our search String could not possibly be in any of these slots (this assumption is fine as we are adding names to our array in consecutive slots starting from the first slot). So, for the next iteration of the while loop, the search array range will be from the first slot to the slot with subscript number middleSub-1.

Again you can see the algorithm here referring to the **compareTo**() method as the array in question is storing **String** values.

● The last part of the application to discuss is the private inner class **TextFieldEventHandler**. The actionPerformed() method within this class defines what will happen when the user hits return on the text-field of the GUI. First of all, the text entered (if any) is retrieved from the text-field and then a test is performed to see whether any text was actually entered. If no text was entered, a message dialog is displayed to give the user an error message and the method immediately exits through the statement

**return;**

This statement is similar to a break in what it does, but a standalone return statement **immediately exits from a method** rather than a looping structure or a decision-making structure. So here, we are essentially saying “the user entered nothing so give them a message indicating this and do nothing else associated with this method”. It means that if the user enters nothing, nothing gets added to the names array either, **not even the empty string**.

However, should the user enter even a single character on the text-field, the method will continue to execute after this if test. The next thing that happens is that the text-field will be immediately cleared of information and the name entered will be added to the array in the location indicated by the global variable **textFieldEventCount**. Recall that when the GUI first launches the value of this global is zero and so, assuming this statement is reached when the user hits return on the text-field for the first time, the name entered will be put into the very first slot of the names array i.e. **names[0].** The global is then incremented for the next time the user hits return on the text-field, so the next name entered will then go into the next slot of the array.

● The for loop that follows iterates through all the characters in the name entered and keeps track of the number of lowercase letters and uppercase vowels using the **charAt**() and **isLowerCase**() methods. Finally, the text-area is set to display the length of the name entered, the number of lowercase letters in the name and the number of uppercase vowels in the name. The counters lowercase and uppercaseVowels get reset to zero every time actionPerformed() is called with a non-empty String input.

**Organising your Work**

You should have a folder under X: called SP2Stuff created. This time, create a folder called **Lab16** within SP2Stuff to save your work from this lab session.

**Typing in (some of) the Code for the Program Just Analysed**

This program is very long so I have already put most of it in the file called NamesAnalyser.java in the LabSheet16 folder on X: drive. However, it is missing the code for the **selectionSort**() and **binarySearch**() methods, so do type those in to refresh your minds on these important algorithms – note that your final exam will consist of a combination of GUI and arrays within the one application so it important to be able to manipulate and process arrays in various ways.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and **test it fully**. Can you see how it might be possible to crash the application? Put your theory to the test!

The next exercise is a biggie! Take your time trying to code the solution to this exercise and feel free to make use of some **pseudocode** when formulating your solution since it is a very “chunky” program with lots of parts - my own solution is about 230 lines of code, we’ve come a long way since “Hello World!” ☺.

**Exercise 1**

Write a Java GUI application called Exercise1.java that contains a JFrame window. This window should contain 2 labels, 2 text-fields (of size 8) and 3 buttons. It should have dimensions of 500 x 150 pixels. It should use a flow-layout style and the application should terminate when the close button is pressed on its title bar. The first label should have the text “Lower Limit” on it while the second label should have the text “Upper Limit” on it. When the application runs, the user will be able to enter numeric values into the text fields.

The first button on the GUI has the text “Generate Random Numbers” on it. When this button is pressed, a method called **populateArray**() should be called which will generate a set of 100 random whole numbers within the range lower limit to upper limit, as entered by the user on the text-fields (note that your populateArray() method here will therefore take **3 arguments** when it is called). Should either text-field be blank or contain anything other than a whole number (positive or negative), an error message dialog appears instead. There should also be a check done to ensure that the lower limit value is less than or equal to the upper limit value. If this is not the case, then again a suitable error message should be issued to the user. Provided that everything is in order, when the user presses the “Generate Random Numbers” button, the program should go ahead and populate the array with the 100 random values and then display this array neatly aligned in rows of 10, on a text-area within a message dialog, through another method called **displayArray**().

As mentioned already, you should put validation code into your program so that only whole numbers are accepted in the text-fields. If the user enters anything other than a whole number, then they will get an error message dialog. Put the validation code into a separate user-defined method called **isWholeNumber**() and make it as foolproof as you can – when the “Generate Random Numbers” button is pressed, this method will take in one of the values entered into the text-field (a String) and then check to see if it constitutes a whole number, returning the boolean value true or false. So this method will be called twice, one for each text-field.

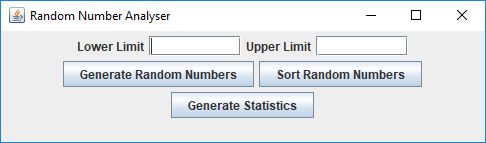
The second button on the GUI has the text “Sort Random Numbers” on it. When this button is pressed, the current array of 100 random numbers should be sorted into ascending order and then displayed neatly aligned in rows of 10, on a text-area within a message dialog, using the **displayArray**() method as before. You should use the selection-sort algorithm here to perform the sort, through a method called **selectionSort**().

The third button on the GUI has the text “Generate Statistics” on it. When this button is pressed, some statistical analysis is performed on the array of random numbers to determine the following:

* The count of positive odd numbers generated
* The count of numbers generated that end in a zero
* The list of the 10 largest numbers generated (need to call **selectionSort**() first for this part, as you are really just picking out the last 10 slots of the sorted array, so might as well call it before doing any of these stats)
* The average of the numbers generated (to 2 decimal places)

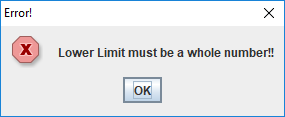
All the statistics should be displayed on a message dialog.

When the application is first launched it will look as follows:

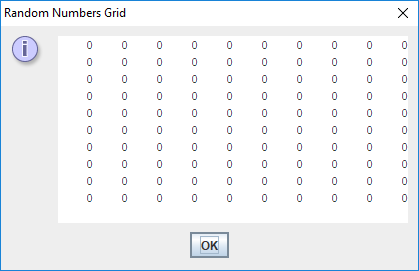


If you just press the various buttons at this point, without entering anything you should see the following:

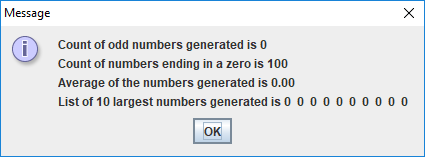
**Pressing the “Generate Random Numbers” button (lower limit is validated first)**



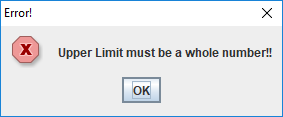
**Pressing the “Sort Random Numbers” button (all array slots default to zero)**



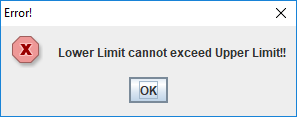
**Pressing the “Generate Statistics” button (default results as all slots are zero)**



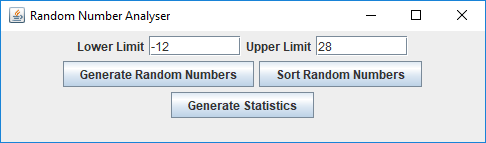
**If a valid value has been entered into the “Lower Limit” text-field and an invalid one entered into the “Upper Limit” text-field, the following dialog appears when the “Generate Random Numbers” button is pressed**



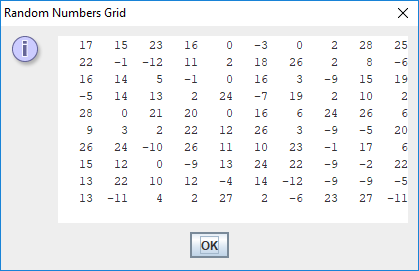
**If both values entered into the text-fields are valid whole numbers, but the lower limit exceeds the upper limit, the following dialog appears when the** **“Generate Random Numbers” button is pressed**



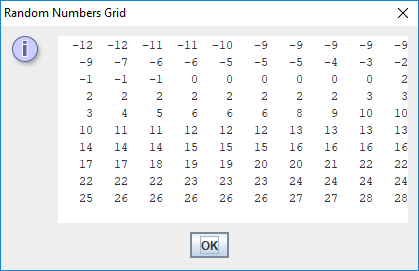
**Entering some valid values into the text-fields such as -12 and 28**



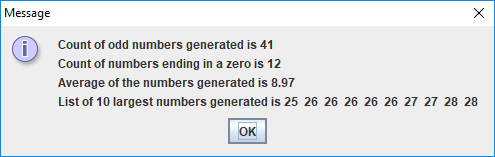
**When the “Generate Random Numbers” is pressed I get the following (yours will be different due to the random nature of the program)**



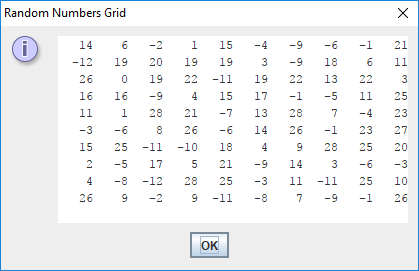
**When I now pressed the “Sort Random Numbers” button I got the following (again, yours will be different due to the random nature of the program)**



**Next, when I pressed the “Generate Statistics” button I got the following**



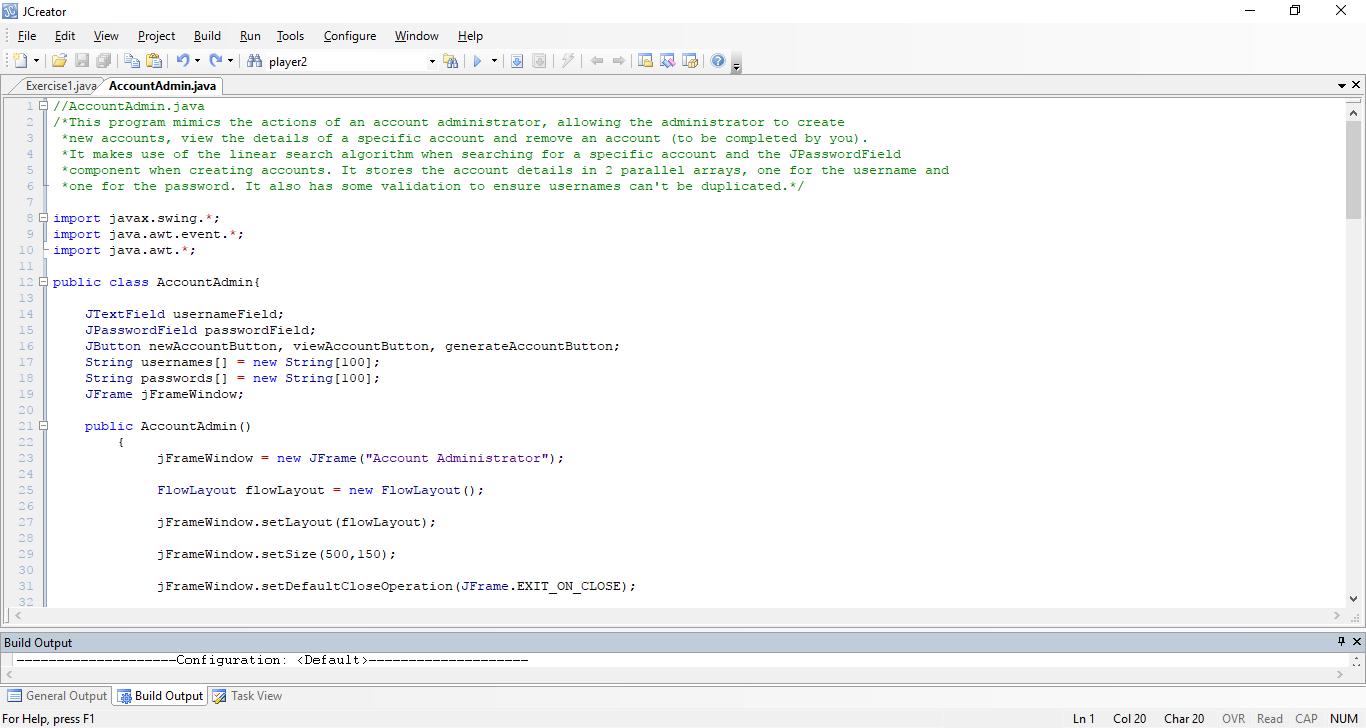
**Each time the “Generate Random Numbers” button gets pressed, even if the lower and upper limits are unchanged, a new set of random numbers will populate the array**

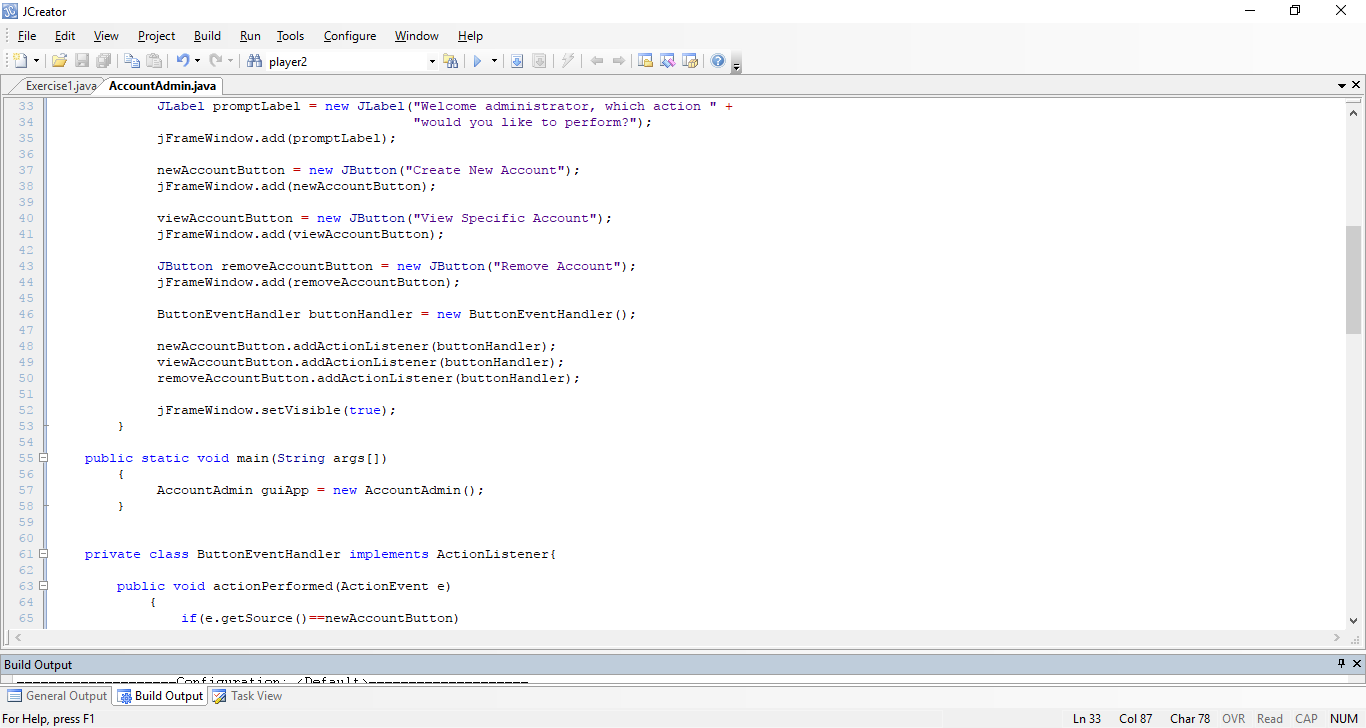


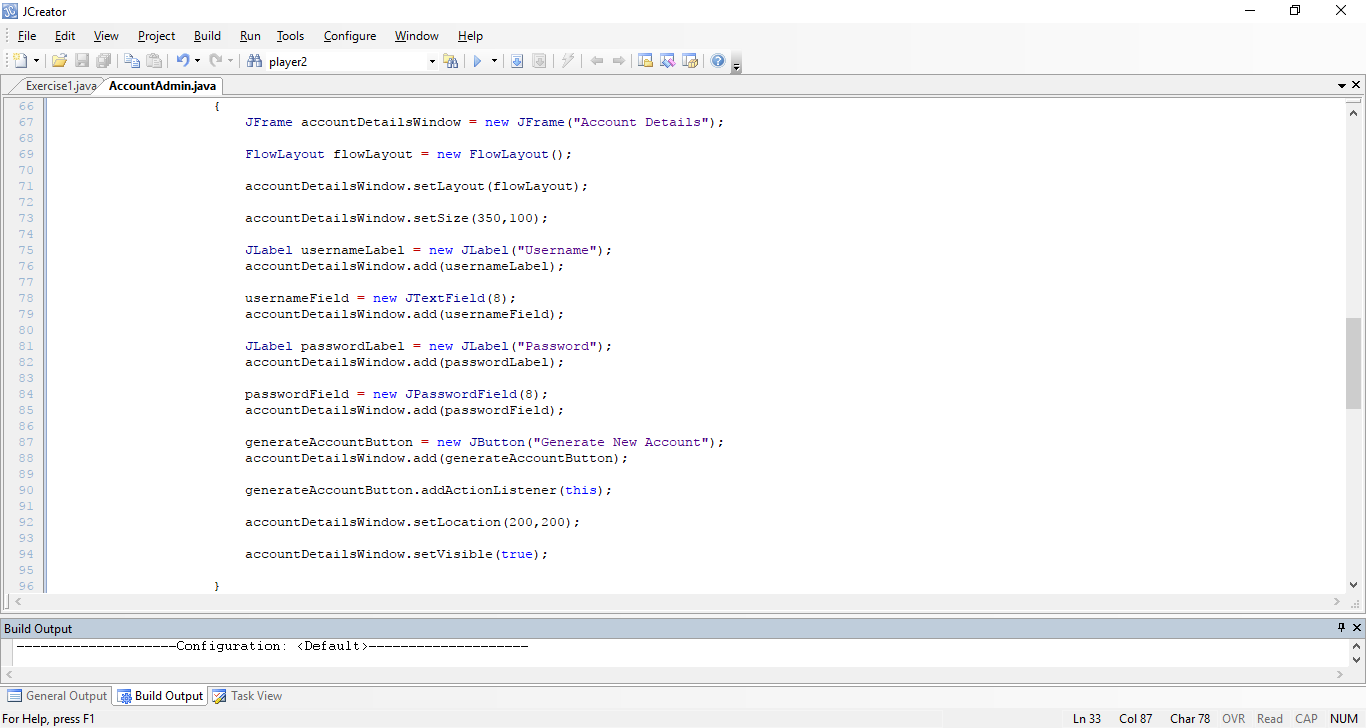
We will now look at another example of a GUI program that combines with arrays to do something useful.

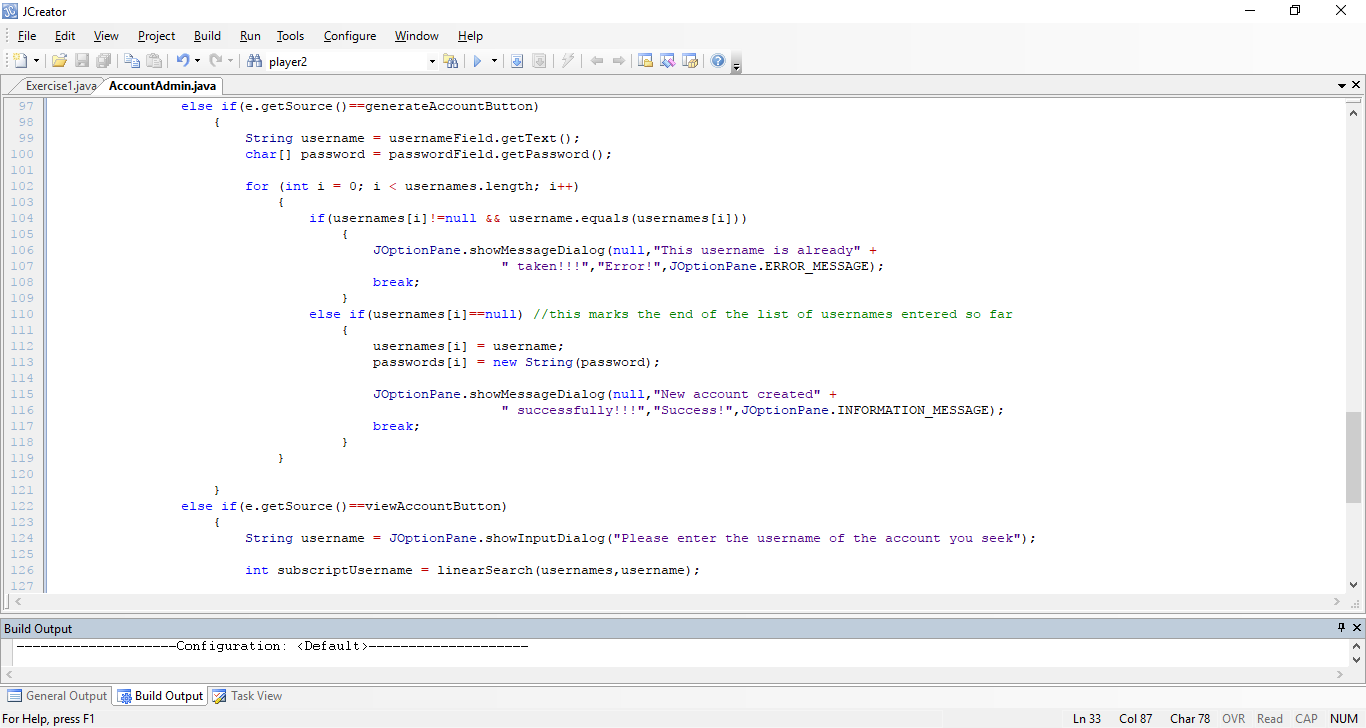
**Aim:** This program mimics the actions of an account administrator, allowing the administrator to create new accounts, view the details of a specific account and remove an account (this will be completed by you as an exercise later). It makes use of the linear search algorithm when searching for a specific account and the JPasswordField component when creating accounts. It stores the account details in 2 parallel arrays, one for the username and one for the password. It also has some validation to ensure usernames cannot be duplicated. Again, note that the **code for most of this program is already available to you** in the LabSheet16 folder

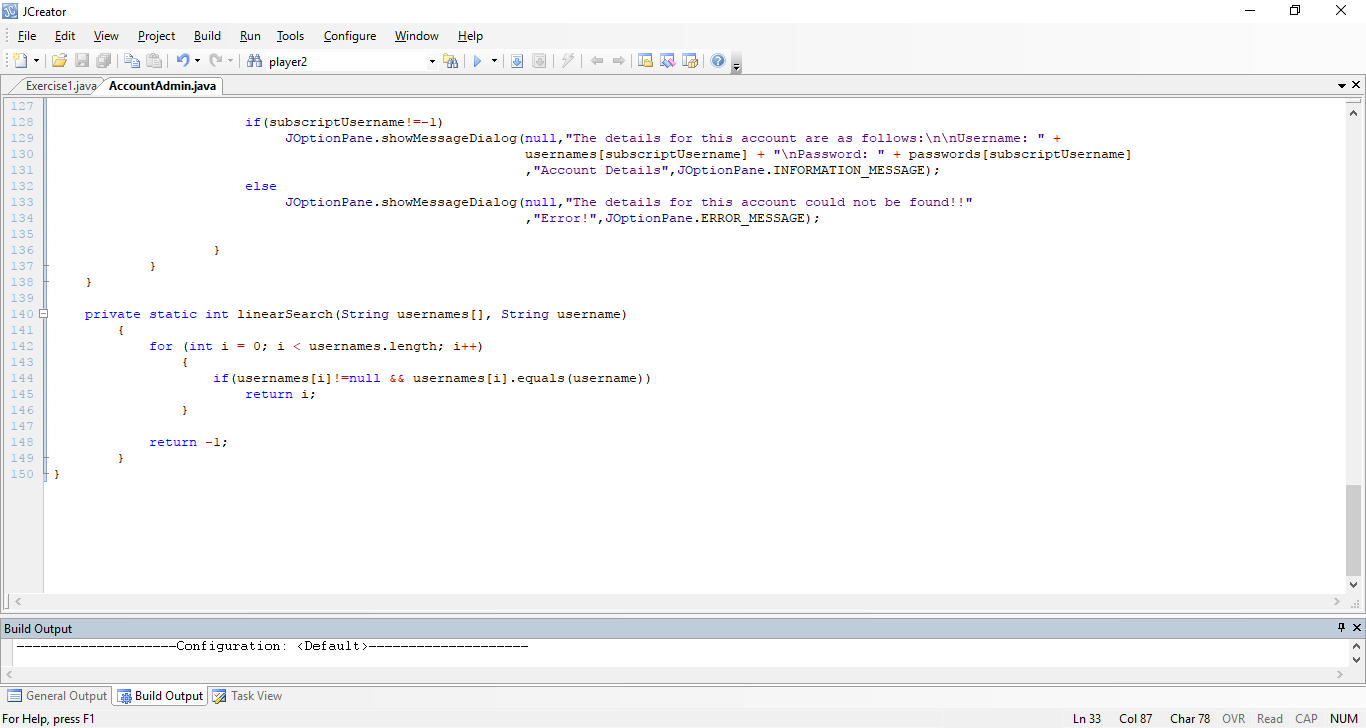
**Java Code:**









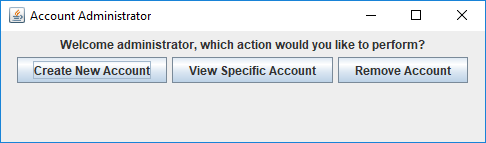


**Program Analysis:**

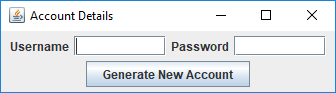
● The program begins by importing all the usual GUI packages.

● You can see in this case that several **global** references are defined. The first is a text-field into which the user will type usernames at runtime. The second is the corresponding password-field into which the user will type passwords at runtime. You may recall that a password-field obscures what is typed into it by displaying a large bullet-point type character instead, for security purposes. Next there are a number of JButton references defined, and these will be used at runtime to carry out various tasks, such as creating a new account and viewing a specific account (the “**Remove Account**” functionality will be left as an exercise for you later). Finally, there are the all-important **parallel arrays** for storing the usernames and their corresponding passwords.

● The GUI constructor AccountAdmin() begins in the usual fashion, creating a JFrame and setting up the JFrame in various ways. A label and 3 buttons are added to the JFrame and action-listeners are registered for each button. When the GUI launches you will see the following



● The private inner class **ButtonEventHandler** contains the actionPerfomed() method that gets called whenever an “action” event occurs on one of the GUI buttons. The first block of code within actionPerfomed() deals with presses on the “Create New Account” button. At lot of the code within in this section will have a very familiar look to it – all that is happening in this case is that, should the user press this button, another JFrame window will get created and we will be entering the account username and password details within this second JFrame. So, for the first time, you will see two JFrame windows on the screen simultaneously. This second JFrame window will look as follows when the “Create New Account” button gets pressed



It has 2 labels, a text-field, a password-field and a button, with an event-listener registered for this button.

● One part of this GUI set up that is different to what you have seen before is the code:

**generateAccountButton.addActionListener(this);**

which registers an action-listener for the button. What is unusual is the use of the Java keyword **this**, which you have not seen used before in the lab sheets. We would normally use the code:

**ButtonEventHandler buttonHandler = new ButtonEventHandler();**

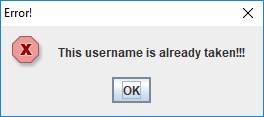
**generateAccountButton.addActionListener(buttonHandler);**

in this situation instead (and this original code still works perfectly well).

However, you see that the new version of the code above, which uses the this keyword, is contained within the **private class ButtonEventHandler** itself (whereas when we register listeners normally, we are doing it from the GUI constructor method). Whenever we want to register an action-listener for a particular button, we must make sure that the argument passed to the method **addActionListener**() is a reference to an object which subclasses ActionListener. Since the ButtonEventHandler class “**implements**” the ActionListener interface, it is definitely a subclass of ActionListener and, because the button in question is also defined within the same class, it can “see” that its parent class implements the listener interface, allowing it to refer to (an automatically created object of) its parent class ButtonEventHandler by the keyword “**this**”. You will see the **this** keyword used many times next year in various contexts, so it is no harm being aware of its existence at this stage. But if you don’t like the look of it, feel free to use the alternatively version instead.

● The second block of the actionPerformed() method deals with events that happen on the “**Generate New Account**” button of the second JFrame. In this case, the values typed by the user into the text-field and password-field are retrieved first of all. Notice that the method to read the value typed into a password-field is called **getPassword**() and that this method **returns an** **array of characters** i.e. char[] rather than a String.

Next a for loop iterates through the contents of the usernames array. If we come across a slot in the array that isn’t empty (**usernames[i]!=null**) but the username entered by the user matches exactly the value contained in that slot (**username.equals(usernames[i])**) then the username has already been taken, so we give the user a message in this regard as follows:



and the loop immediately exits after the OK button is pressed on the dialog, since there is no point in checking any of the remaining slots in the array.

However, if we come to an array slot, and its value is currently **null** (recall that both arrays in this case are **String** arrays and so will begin with all their slots set to the default value of **null**) then we know that we have reached an **empty slot** in the array. Therefore, we can now just set the value of that slot to the username entered and set the corresponding slot of the passwords array to the password entered with the code

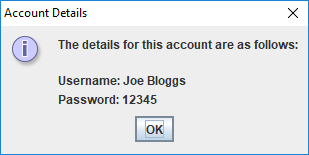
**usernames[i] = username;**

**passwords[i] = new String(password);**

notice that we must **convert** the value stored in password (a **char[]**) to a **String** here.

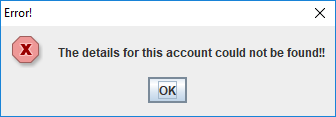
Once this is done, we give the user a confirmation message to indicate that the username used was valid and the account has been created successfully, then immediately break from the loop (no further loops needed at this point as the job has been done).

● The final block within actionPerformed() deals with viewing the details of a specific account when the “**View Specific Account**” button has been pressed on the first JFrame. Here the user is prompted for a username on an input dialog, then a **linear search** is performed on the usernames array with the username entered as the key value. In our linear search algorithm, the method will return the subscript number of the slot in the usernames array that contains the key username, if it exists, and -1 otherwise. An appropriate message is then given to the user depending on the outcome of the linear search e.g.



In this case the account with username “Joe Bloggs” was found and the corresponding password is displayed in plaintext.

If the username cannot be found the message below is given instead



**Typing in (some of the) Code for the Program Just Analysed**

Again, this program is very long, so I have placed much of the code already in a file called **AccountAdmin.java** in the LabSheet16 folder on X: - you just need to add the code for the private class **ButtonEventHandler**.

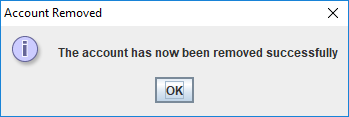
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and **test it fully**. Can you see how it might be possible to crash the application?

**Exercise 2**

Save the AccountAdmin.java program as **Exercise2.java** and carry out the following additions/modifications:

1. See if you can add the functionality for the “**Remove Account**” button. First you will have to ask the user for the username of the account they wish to remove. Then a **linear search** should be performed to find this username. If it is found, then we will “mark” the slot in the usernames array as empty by setting its value to **null**, and you will then set the corresponding slot in the passwords array to the same value. This means that this empty slot can be reused again in the future when we want to add a new account. Once you have this part coded, test it out fully to see whether it is working correctly (use **Arrays.toString()** for a quick test within a **println()**). The user should get an appropriate message on a dialog to indicate the removal was successful.

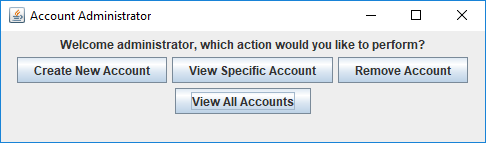


If the username was not found, then the user should get an appropriate error message instead.

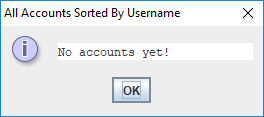
1. You’ll recall that the sample program used a **linear search** algorithm rather than the more efficient binary search. This was partly so that your minds could be refreshed on the linear search operation. However, it’s time to bite the bullet and see if you can now modify the existing code so that it uses a binary search rather than a linear search. Recall that the binary search needs the array to be **sorted** in advance and, because there are 2 parallel arrays at work here, it complicates things because, essentially, both must be sorted in tandem. Therefore, your **selectionSort**() method here will take **both arrays as arguments** and then, using the usernames array as the basis for the sorting, also sort the corresponding slots in the passwords array so they “follow” the ones in the usernames array as required.

Once the sort has been successful for you, you should then try to write the **binarySearch**() method to do your search on the usernames array. Keep in mind that you have to deal with the possibility of empty slots here. Write the binarySearch() so that it returns the subscript of the search key if found, otherwise -1. Just keep testing your code as you go along to ensure you are heading in the right direction and do try to code both methods from scratch so that the algorithms becomes second nature to you.

1. Finally, try to add another button with the text “**View All Accounts**” to the main JFrame window that allows the user to view the details of all the accounts that have been created to date, **sorted** by username in ascending order. All the account details in this case should be shown neatly formatted within a text-area on a message dialog. If no accounts have yet been created at this point then the text-area should just display the message “No accounts yet!”. The display should not show any empty (null) slots.



So if the button is pressed, right at the start before any accounts are created we get:



And then, after creating a number of accounts, we get something like:

