**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 14 – Basic GUI Components**

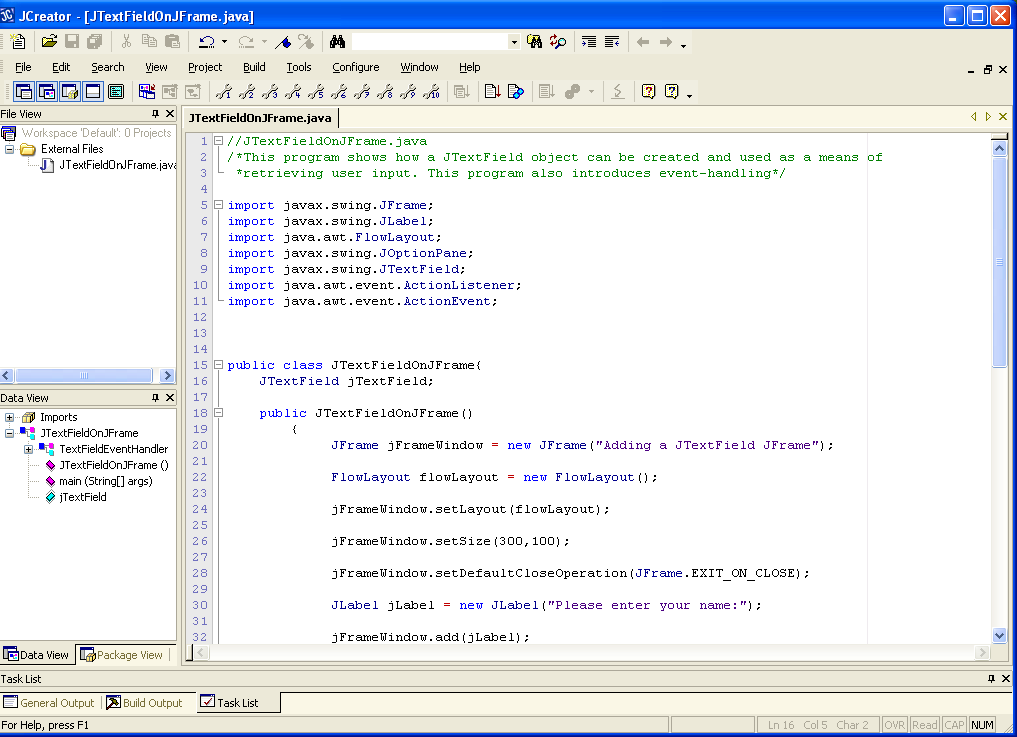
We continue in this lab with our examination of some of the more basic GUI components. The whole area of GUIs in Java is enormous and there are masses of predefined classes and methods available for their creation. Naturally you are not expected to remember the details of each and every class and method you encounter – what you need to be able to do is to use given classes and methods properly as you code.

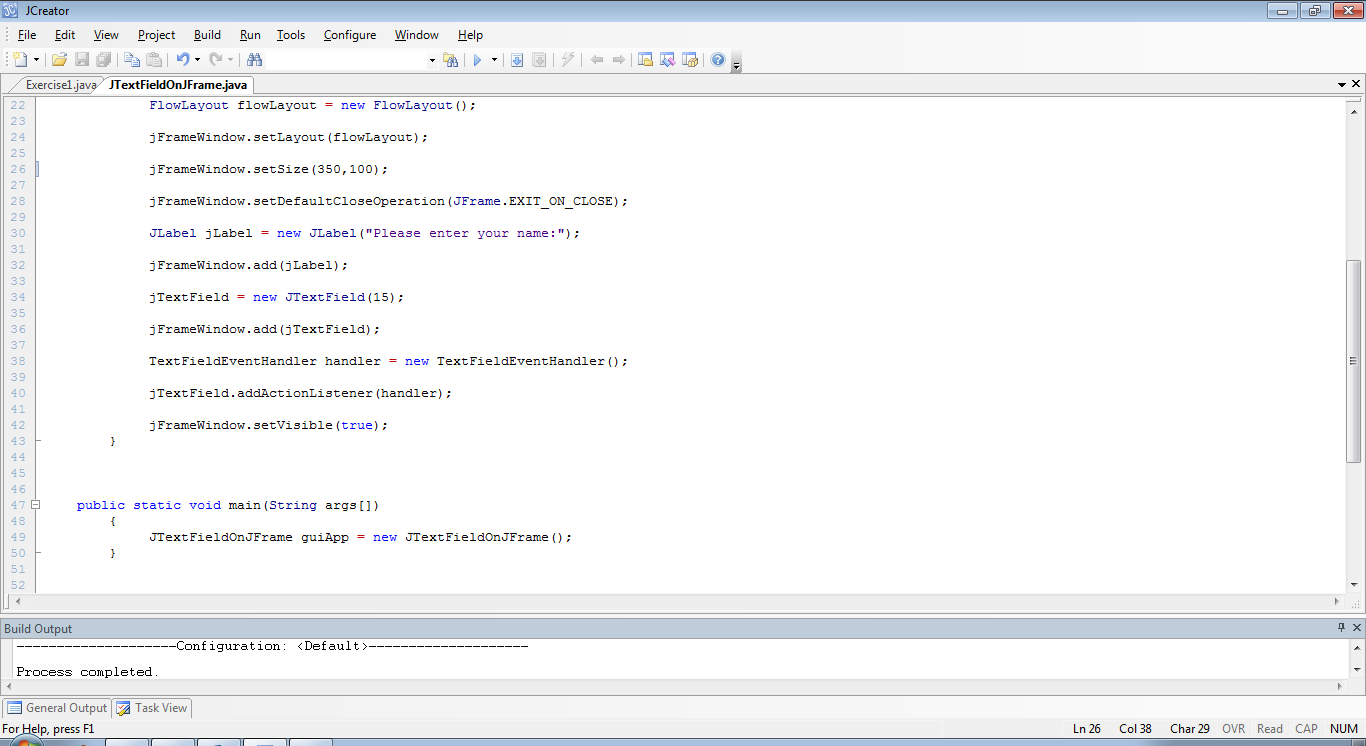
**Using a JTextField**

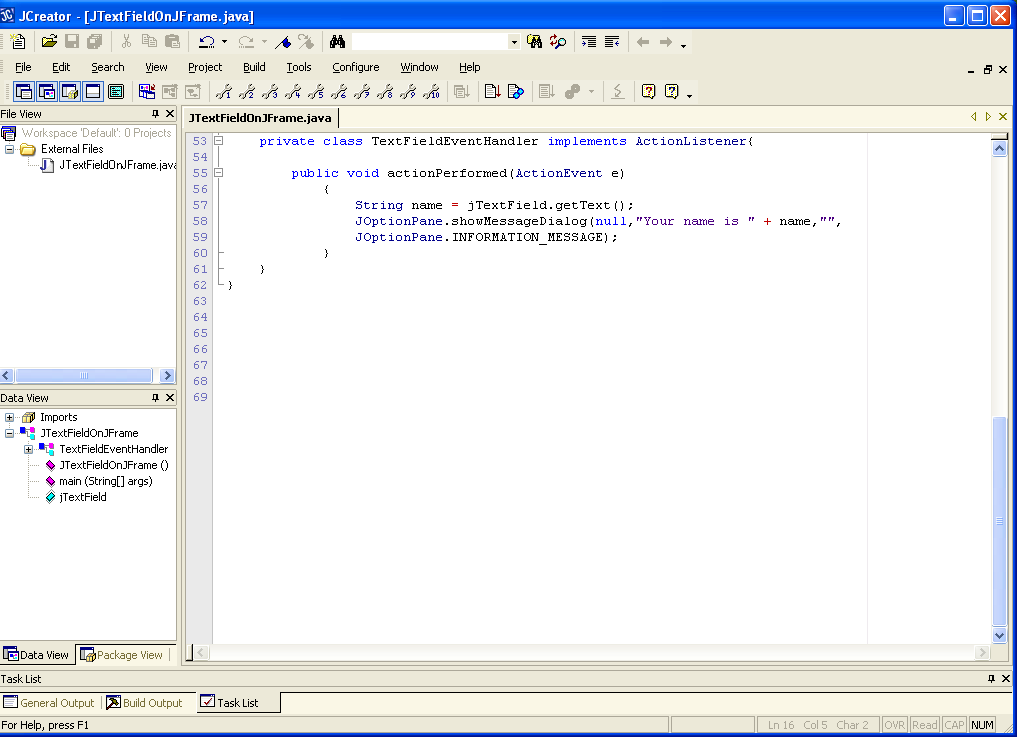
Like labels, text-fields are fundamental to GUI creation. The main difference is that a text-field is normally used to allow the user to supply information to the application, just like the text-field in the JOptionPane input dialogs.

**Aim:** This program shows a JTextField being used to retrieve some information from the user.

**Java Code:**

****



****

**Analysis of Program:**

● In addition to the ones we had last time, there are 3 further classes in this application. These are **JTextField** from the package **javax.swing** (like all the “J” classes), **ActionListener** and **ActionEvent** from the package **java.awt.event**

This is a very good time to mention that, when it comes to importing classes, there is a shorthand notation for importing several classes from the same package. For example, there are 4 classes from javax.swing being used here so we could import all of them at once as follows:

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

You can do this in future GUI applications since it saves on coding time and effort. However, the \* means that the compiler will load into memory all of those classes from the package being used in the program, which is not great in terms of memory usage but like always, there’s a tradeoff involved. Likewise, there are 2 classes from the java.awt.event package so you could import both of these as follows:

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

● There are many changes in this GUI application in terms of its general layout compared to the last few we have examined. The main reason for this is because **this application needs to handle an event**.

In this application, the user is prompted to enter their name. They are given a text-field to enter their name and when they have entered it, they should **hit return** on the text-field to signal the end of their input. When this happens, **an event occurs** on the GUI. What **triggered** the event is the user hitting return on the text-field. However, this **event will go unnoticed unless there is some way of detecting it**. In Java, the way to detect such an event is to **register a listener** for the event. The listener is literally like an “ear” which is listening out for the event to occur.

Different GUI components can generate different types of event, for example, in this case we have a text-field component which can generate an **ActionEvent** when the user hits return in it. You will see that buttons also generate an ActionEvent when they are pressed. Other components such as radio-buttons and check-boxes generate a so-called ItemEvent. A MouseEvent can be triggered on practically any GUI component when a mouse interacts with the component in some way and there is also a KeyEvent which can be triggered by hitting keys on the keyboard.

In any case, our focus here is on dealing with the ActionEvent that may be generated. Recall that in order to deal with the event, we must register a listener for the event. This is achieved with the code:

TextFieldEventHandler handler = new TextFieldEventHandler();

jTextField.addActionListener(handler);

the first line of code here creates an object of type TextFieldEventHandler and a reference to the object is stored in handler. The reference is then passed as an argument to the addActionListener() method, called on the JTextField object reference jTextField. If we did not include this code then there would be no “ear” listening out for the text-field event and no special action would be taken if it occurred. Now at least we have an “ear” in the system listening out for the event should it occur and it is the **TextFieldEventHandler** object in the system.

Now, should the event actually occur, it must be handled in some way. What happens is that details about the event (such as the object reference to the component that caused the event to occur in the first place) are **put together by the Java system automatically** and placed into an object of type **ActionEvent**. A reference to this object is then passed as an argument to the method **actionPerformed**() contained in the private class called **TextFieldEventHandler**. This **event-handling** **method is called automatically** by the Java runtime system when the event is triggered here. You’ll recall that earlier we created an object from this class and it effectively became the “ear” listening out for the text-field event.

It is the **code contained within this actionPerformed() method that dictates exactly what happens when the text-field triggers an event**. The code for this method is as follows here:

public void actionPerformed(ActionEvent e)

{

String name = jTextField.getText();

JOptionPane.showMessageDialog(null,"Your name is " + name,"",

JOptionPane.INFORMATION\_MESSAGE);

}

You can see that the method takes an argument of type **ActionEvent**. Recall that this argument is generated automatically by the Java runtime system when the event actually occurs. Now that we have a listener in the system capable of handling such an ActionEvent, the Java runtime will automatically pass on the event object reference to the method for processing.

The first line of the code here will retrieve the text typed into the text-field and store it in the String variable (object reference) name. Then a JOptionPane message dialog is used to display some information to the user, effectively just to echo back the name they typed into the text-field.

So that’s event-handling in Java – pretty confusing or what! Spend a little time trying to get to grips with it though, it will be worth it in the end – remember you’ll be writing GUI applications in Java for the next few years! The rest of the GUI applications we’ll examine will all have some event-handling in them so hopefully you’ll get more used to them with those also.

● Getting back to the code for the application now, the code

jTextField = new JTextField(15);

jFrameWindow.add(jTextField);

creates a JTextField object. The width of the field is decided by the number passed in as an argument. The 15 here means that the text-field can display about 15 capital “W” characters (the widest character there is). It could display many more ‘i’ characters. Even though it may only display a certain number of those characters, you can type in text of any length.

Once the text-field object is created, its object reference is passed as an argument to the add() method to add it to the JFrame window.

You may have noticed here that the **JTextField object reference is actually declared outside of any method**.

JTextField jTextField;

This is actually crucial here given the way the application is laid out. The **actionPerformed() method needs to be able to access this reference** so that it can call the getText() method on the text-field. If the reference was defined in a specific method, then it would have **local (block) scope** and would be invisible outside of that method. Therefore, we end up having to declare the object reference as a **“global” variable**. It is global in the sense that **any method contained in the class JTextFieldOnJFrame has access to the reference** should it need it. You will see this sort of behaviour in future GUI applications we examine – it is **normally necessary whenever there is event-handling** involved in an application.

● Notice that the main() method has become incredibly short now. Almost everything that happens in terms of creating the GUI and registering event handlers happens within the **JTextFieldOnJFrame()** method. The only thing that happens in main() is that an instance of the class **JTextFieldOnJFrame** is created through a call to the method **JTextFieldOnJFrame()**. Calling this method then kickstarts the whole GUI creation process.

You will see that, for GUI applications, the main() will generally be a one-liner! This is the general layout of GUIs in Java. It needs to be done this way for various reasons that are well beyond the scope of this module. The exact details of why the GUI application is laid out in this particular fashion may be explained to you next year in your OOP2 module. For now, if you can take the general layout on board, and get used to the methods involved, you will be doing just fine.

● Finally, I draw your attention to the fact that there is a **private** **class** defined in this application. You have not seen one of these before. All the classes you have created in the course so far have been **public** (we will discuss public and private in a little more detail in the next lab sheet). This private class is often referred to as an **“inner” class** – for the simple reason that it is **defined within an existing class** (the main class). When an inner class is defined, it is meant to **act as a helper class to the main class**. This is how the private inner class operates here. It contains the actionPerformed() method which is called whenever an event is triggered by the text-field, thereby helping the main class to carry out its job.

Note that this class says “**implements ActionListener**” at the top of it. ActionListener is an example of an **interface** and the reason behind interfaces and how they differ from regular classes will be examined in your OOP2 and OOP3 modules.

For this module, you can simply take it that whenever you need to write a private inner class that is capable of handling an **ActionEvent**, then that same class must have the words “implements ActionListener” at the top of it. You can take it that, because the class TextFieldEventHandler wishes to become an ActionListener (i.e. capable of listening for and handling an ActionEvent) then it must have “implements ActionListener” in order to achieve this goal.

You will see that other private inner classes we write in the future that wish to become listeners and handlers of various types of events must also do something similar.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab14** within OOP1Stuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **JTextFieldOnJFrame.java** in your Lab14 folder. Now, for practice, type in the code for the program above.

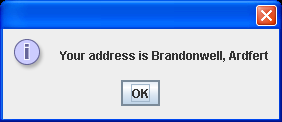
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences 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. Note that you must close both the JOptionPane message dialog (you’ll have to close this first) and the JFrame window to terminate the application.

**Exercise 1**

Save the program above as **Exercise1.java**. Now make the necessary changes to it so that, when the program runs, it appears as follows instead – notice that the text now appears in green.





When you have the output looking similar to that above, you should try to **resize** the window. Notice that it is possible to resize the JFrame window at runtime – see how the components get rearranged on the GUI as it gets smaller – this is the **layout manager** at work. If we were to use no layout manager then we would need to set absolute positions for each individual component on the GUI which is possible but painstaking. We would also have to set the width and height of each component. However, if we were then to resize the window, the **components would not get rearranged** at all – they would remain in the same position all the time. This is one of the benefits of a layout manager – they save us lots of time codewise and are essential in real-world GUI applications.

**Exercise 2**

Write a GUI application called **Exercise2.java** that creates a JFrame window. This window should contain a text-field and a **password-field**, to allow a user to enter their user-name and password combination. The password-field looks just like a text-field but when the user enters text in it, **only asterisks appear** rather than the actual plain text, for obvious security reasons. There should be **labels** associated with each of these fields to prompt the user for the information.

When the user enters the username information and hits return, the **cursor (focus) should automatically move** to the password text-field. Then, when the user enters the password information and hits return, the program should determine whether or not the username/password combination was actually valid.

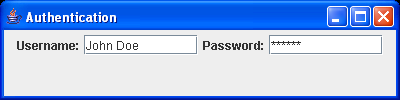
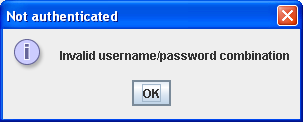
For the purposes of this program, you can take it that the correct username/password combination is “Joe Bloggs”/ “123abc”.

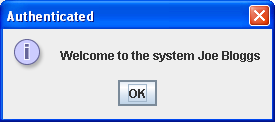
If the combination is valid, a **JOptionPane message dialog** should then appear welcoming the user by their username to the system. Otherwise, they should be told that the username/password combination was invalid.

You will need to use the **Java API documentation** to check out the password-field component class **JPasswordField**.

Note that in this case there are **2 GUI components that could potentially trigger an ActionEvent** – the JTextField and the JPasswordField. In order to determine which of the two has caused the event to be triggered you will need to use the ActionEvent argument passed into the actionPerformed() method. The method you need to call on this argument is called **getSource()**. This method tells you the name of the object reference associated with the object that triggered the event in the first place.

The program should run as indicated in the following sample screenshots.

** **

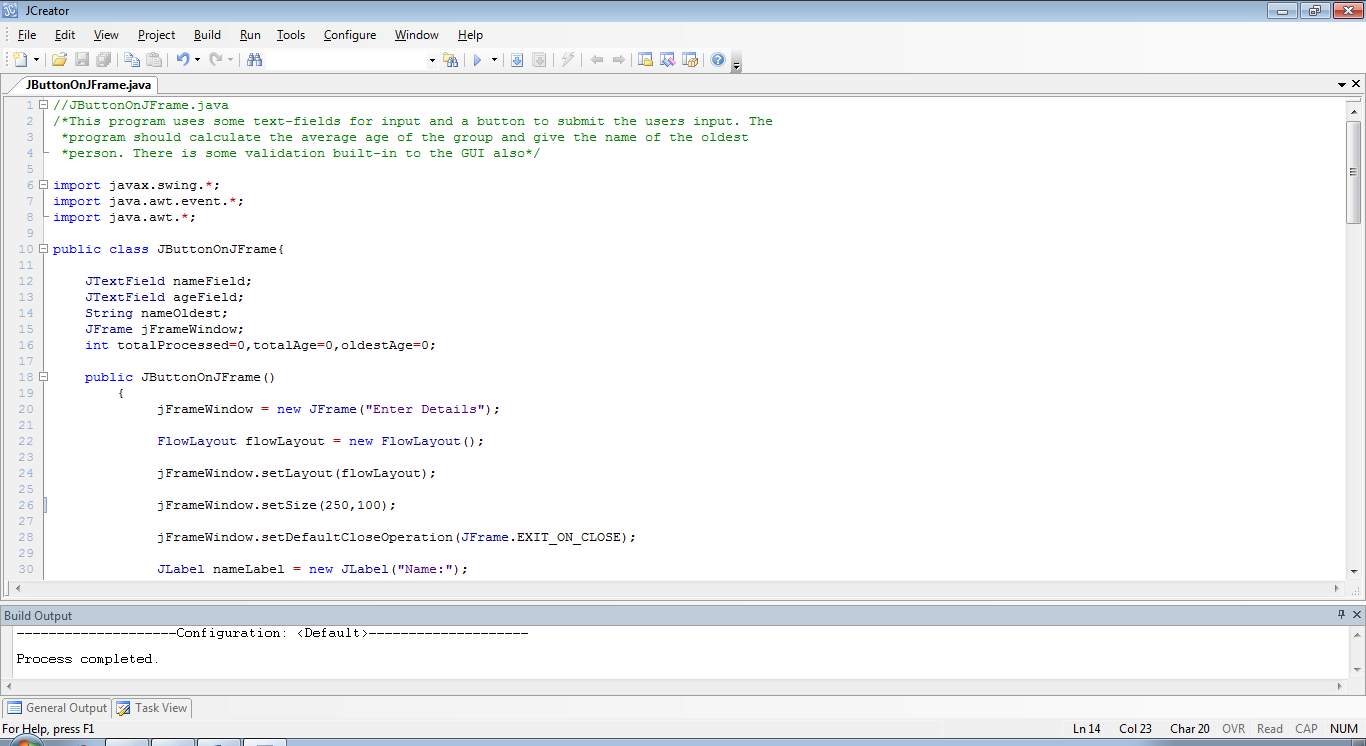
** **

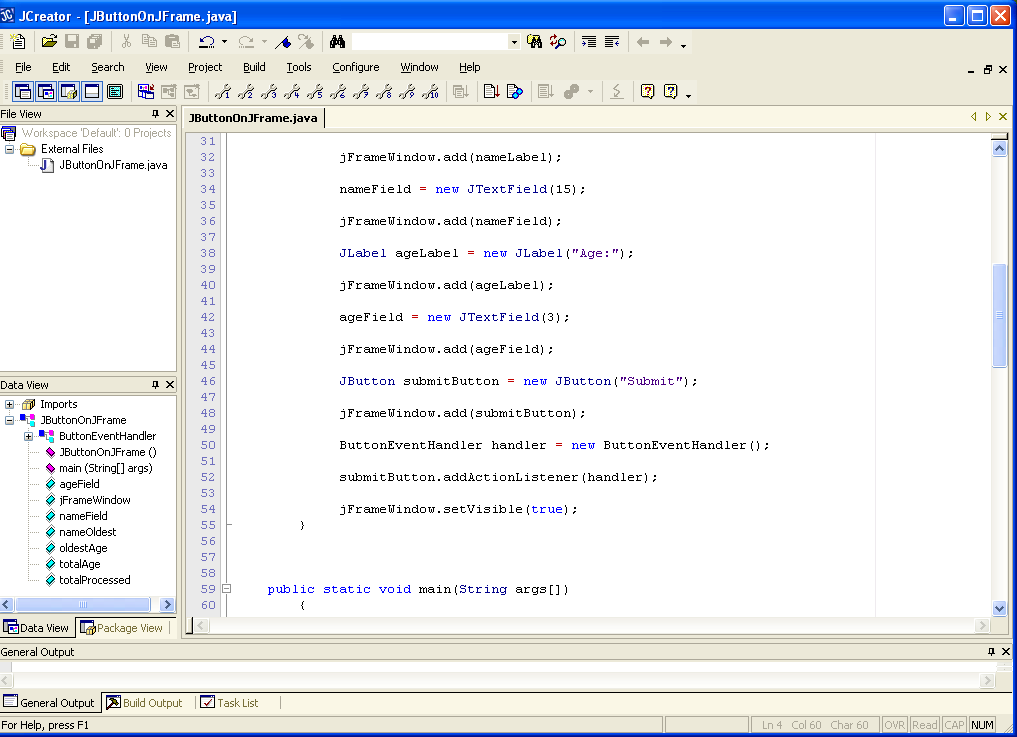
**Using a JButton**

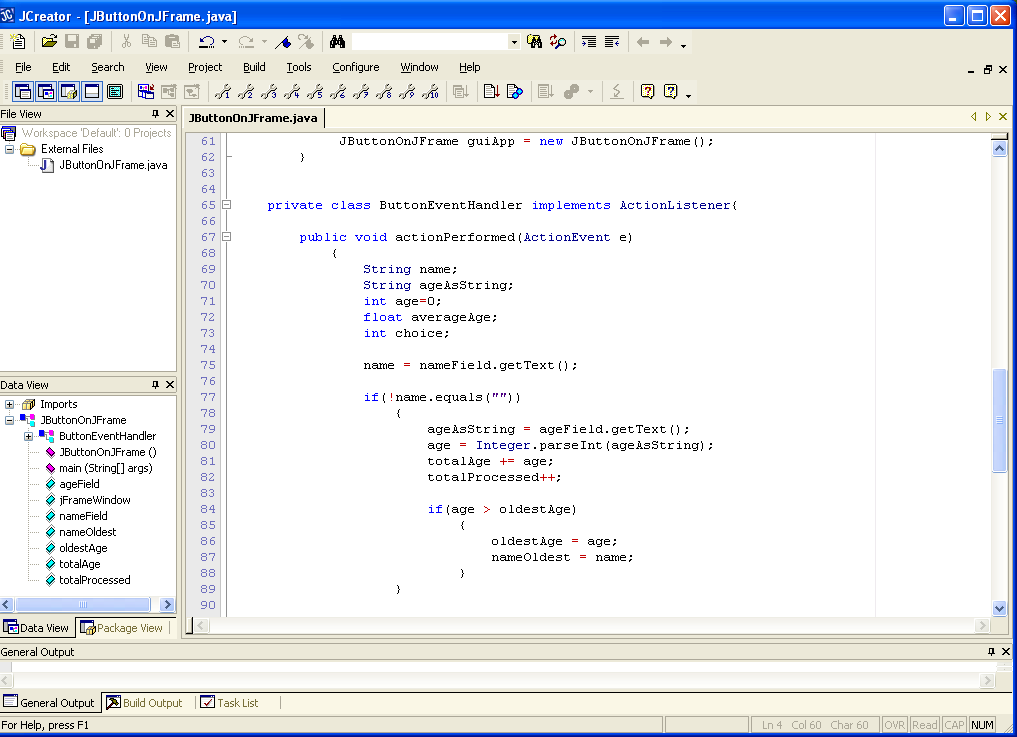
A button is a GUI component that the user clicks to trigger a specific action. Buttons abound in GUI applications. Just look at the JOptionPane message dialog above. It has an “OK” button which you click to signal that you have read the message and now wish to close the dialog. You will see that the syntax for creating a button is very similar to what you have already seen for other components and that event-handling for a JButton is very much like event-handling for a JTextField since clicking on a JButton also generates an ActionEvent.

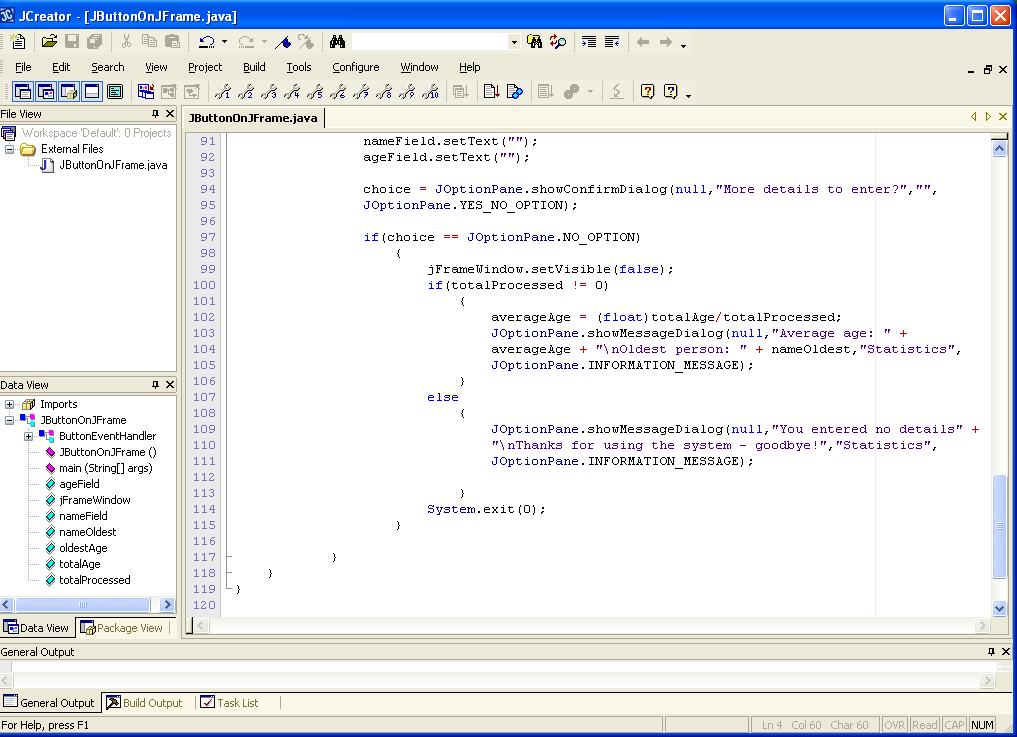
**Aim:** This program shows a JButton being used to submit the details entered by a user. The user can submit many sets of details and when finished inputting, some processing occurs before displaying the results. There is some input validation involved in the application also.

**Java Code:**



****

****

****

**Analysis of Program:**

● The program imports the 3 main GUI application packages using the \* notation. Recall that this only loads into memory the actual classes being used within the application.

● There are several “global” variables/object-references being used in this application. Remember that a variable/reference needs to be made global whenever it needs to be accessed by more than one method in a class, so that all methods can “see” it. Notice how some of these are initialised – these are the counter variables used by the application to keep track of the total number of persons processed, the total age of all persons processed and the oldest age entered.

● The JButtonOnJFrame() method does all the GUI creation in terms of creating the GUI components and adding these to the JFrame window. It also registers a listener object for the JButton. As mentioned already, a JButton triggers an ActionEvent when it is clicked. Hence, the type of listener for the JButton here must be an ActionListener. This is achieved since the listener object is a ButtonEventHandler object and this “**implements**” the ActionListener interface as required.

Note that there are no listeners registered for the 2 text-fields in this case. Therefore, when the user hits return on them, nothing happens.

The code for creating the JButton is what you might expect. The argument “Submit” just dictates the text that will appear on the button.

● The main() method contains just a single line of code. It just calls the JButtonOnJFrame() method to kick-start the whole GUI creation process.

● The private inner class ButtonEventHandler is relatively complicated. Its actionPerformed() will be called into action whenever the user clicks on the button. When this happens, the text contained in the name field will be retrieved.

As long as this is not the empty string “”, then the age will be retrieved from the age field (the thinking here is simply that if the user did not enter a name, then there is no point in checking to see if they entered an age – its “all or nothing”). The age is then converted to an integer and then the “global” variables totalProcessed and totalAge are updated.

Next a test is performed to determine whether the age just retrieved is greater than the value of oldestAge. If it is then we must reset the value of oldestAge to this age and also track the name of the person associated with that age (since we ultimately want to display the name of the oldest person).

● The next piece of code

nameField.setText("");

ageField.setText("");

might look a bit strange but it is only there to empty the contents of the two text-fields for the next batch of user input.

● The user is then asked to confirm using a confirmation dialog, whether or not they wish to enter more details. If they do then they have clicked the “Yes” button in the dialog and the dialog just disappears and we are left with the JFrame window to continue entering further details.

If the user does not wish to enter further details, then they have clicked the “No” button on the dialog and so we should now do our calculations.

First though, the JFrame window is made invisible (since we have indicated we are finished inputting details, this is a good thing to do) and then a check is performed to determine whether or not the user actually entered some details. If they did then the value of totalProcessed must be greater than zero and in this case we wish to display the results.

The average age is determined by simply dividing the value of totalAge by totalProcessed (typecasting is used here to obtain an accurate average, since both operands are int variables).

Then a JOptionPane message dialog appears with the average age and the name of the oldest person displayed.

If the user didn’t enter any details then they are just given a message indicating this and thanking them for using the system.

In both cases here, as long as they have clicked “No” on the confirm dialog, the program terminates with a System.exit(0) – this is necessary here because we have already hidden the JFrame window.

Apart from showing you how a JButton can be handled in a GUI application, the above program is very useful as it shows you how multiple data-sets can be handled in a GUI environment. Notice that there is **no loop** involved in the program as there would be for console or simple JOptionPane GUI input/output applications – here, the **GUI itself behaves like a loop in terms of its functionality**. It is like a loop that keeps iterating until we make it stop by either hiding it or closing it.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **JButtonOnJFrame.java** in your Lab14 folder. Now, for practice, type in the code for the program above.

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. Try it out with different sets of input, as well as seeing what happens if you don’t input anything. Note that the program is **not perfectly validated** either – can you see where it fails validation?

**Exercise 3**

Save the last program as **Exercise3.java** and modify it as follows:

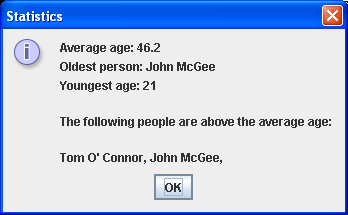
● The program should determine the age of the youngest person

● The name text-field should have an ActionEvent listener registered for it and when an action event does occur on the text-field, the cursor (focus) should immediately move to the age text-field. You can create another private inner class for this is you like called **TextFieldEventHandler**.

● The program should display the names of all the people whose age was above the average age

The last part here is the trickiest and does impact on the existing code in a fairly big way. In order to find the names of the people whose age is above the average, you need to first find the average but you cannot find this average until input has stopped. Therefore you must keep track of all the names and their corresponding ages until the average is known. To do this, you need **two arrays**, one for the names, one for the ages. Also, these arrays **must be declared as “global” variables** so that their contents does not get reset with each call to actionPerformed().

Your program’s final output should appear as indicated in the following sample screenshot (the input screenshots have been omitted here).

****

**Exercise 4**

Write a program that contains a JFrame window that has 2 labels, 2 text-fields and a button. The program should allow the user to convert an amount in pounds to an amount in kilos. If the user enters a amount into the “pounds” text-field and then clicks the button, the equivalent amount in kilos should appear in the “kilos” text-field. Likewise, if the user enters an amount into the “kilos” text-field and then clicks the button, the equivalent amount in pounds should be displayed in the “pounds” text-field.

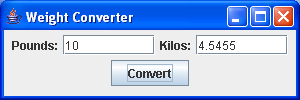
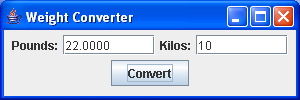
Note that there should be a little **validation** in the application. For example, if the user clicks the button and there are values in both text-fields, a suitable error message should appear and the text-fields should be reset to empty. Likewise, if the user tries to perform a conversion and no value is entered on either text-field, the same error message should appear (use the one in the screenshots below).

All converted values should be displayed to **4 decimal places**.

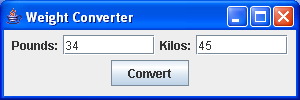
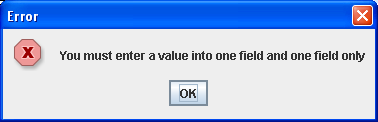
The conversion factor here is **1 kilo = 2.2 pounds**.

Your application should run as indicated in the following sample screenshots:

**Input below was 10 pounds Input below was 10 kilos**

**In the case below, both fields had values when the button was clicked**

**In the case below, both fields were empty when the button was clicked**

