**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 3 – GUI Input & Output in Java**

Hopefully you found the review of the Java basics useful to refresh the mind and get the fingers coding again. We begin our examination of new material with a relatively straightforward topic, using some basic Java GUI components for input and output.

**GUI Input and Output vs Text-mode Input and Output**

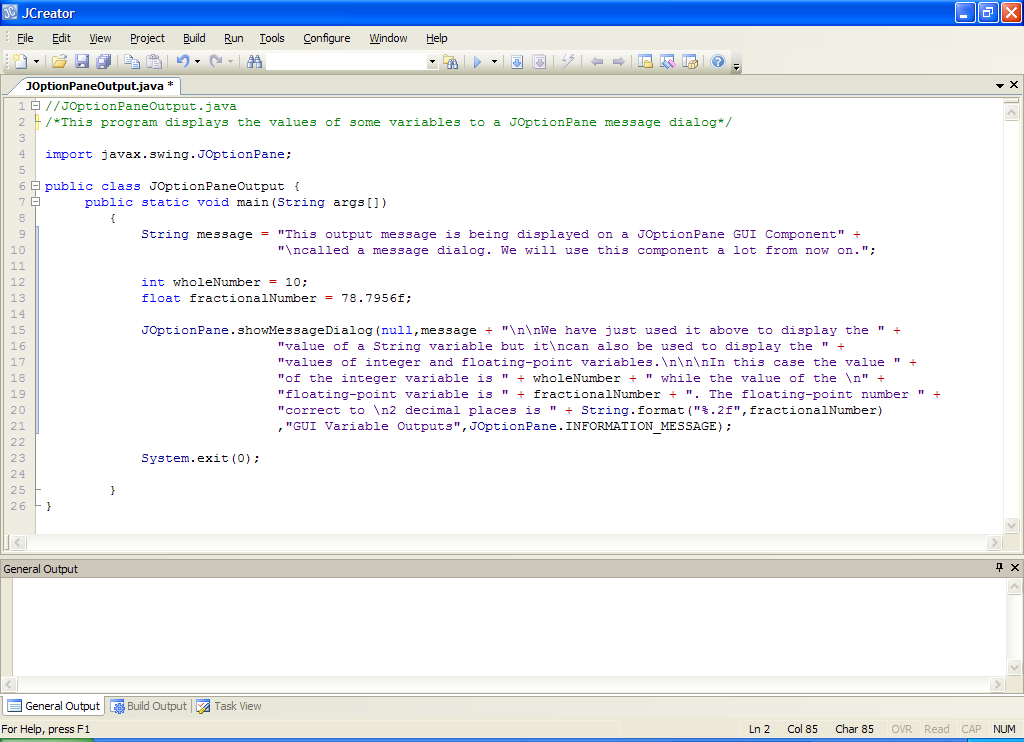
Throughout the entire “Introduction to Programming” module and up to now in this module, we have not considered **graphical user interface** (GUI) programming. There is a reason for this – it is not very easy to do but, like everything else in programming, it’s just all about practice! All our effort so far has involved **console** or **text-mode applications**. This type of programming is **ideal for learning the very basics of programming** as we don’t get bogged down in the detail of GUI code, but the reality is that most applications nowadays are GUI based and so we move on to examining these types of programs. The big difference visually, of course, is that **GUI applications are more colourful, more interactive, often more intuitive**, just plain better looking than a console application!

**Simple GUI Output in Java**

Most of our programs from now on will use graphical components for achieving input and output. Whereas our programs up until now have just used the keyboard, graphical input will also require using the mouse. Also, instead of the output displaying on a text-mode console, it will appear within a GUI window.

**Aim**: To output the values of some variables to a GUI window component

**Java Code**:



**Program Analysis**:

• After the comments there is an **import** statement. This is necessary here because we wish to use a GUI class in this program called **JOptionPane**. It belongs to a package called **javax.swing** so we must import this in order to use JOptionPane – otherwise we get a **syntax error**.

• Within main() we initialise 3 variables, the first a String variable, and then an integer and floating-point variable.

• Beginning on Line 15, the class JOptionPane is used to display the values of the variables. The **showMessageDialog**() method, which belongs to the class JOptionPane, is called to display the values. This method takes **4** **arguments** (pieces of information) here - note that method arguments are **comma separated**.

The **first argument** will always be **null** for us in this module.

The **second argument** is the crucial one – it **contains the information we wish to display**. Hopefully you can see that this argument is identical to the kind of thing you do within a print() or println(). String concatenation is used to join variables and pieces of text as usual. The special character “**\n**” displays a newline. Notice that we can use **String.format**() within the call to messageDialog() just as we have in the past. Here we use it to display the value of the floating-point variable to 2 decimal places.

The **third argument** dictates the text that will go into the **title bar** of the message dialog window when the program runs. Here it will be “GUI Variable Outputs”

The **last argument** isn’t really that important for us. This decides the **icon** that will display in the JOptionPane window. Here it will be an **INFORMATION\_MESSAGE** icon which actually means there will be **a little circular icon with an i in the middle of it to the left of the information being displayed.**

Note that a JOptionPane window will **grow automatically to accommodate the text** you wish to write to it. I deliberately used newlines here so that the text was virtually the same width throughout.

Also note that when a message dialog appears on the screen, it effectively **halts the programs execution** until the user either presses the **OK** button at the bottom of the dialog or else **hits return** on the keyboard.

• Finally, the code

**System.exit(0);**

ensures that the application terminates correctly – it is really a precautionary measure. In Java, it is **only necessary to use it at the end of GUI applications** because text-mode ones call the exit() method automatically when they complete execution. So just remember to always include this statement whenever you use a GUI component in an application.

**Creating a Folder for This Weeks Work**

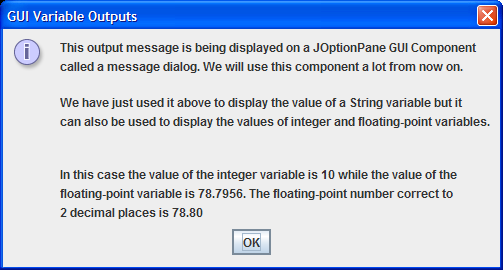
On the X: drive, in your OOP1Stuff folder, you should create a new folder called **Lab3** to store your work for this lab sheet.

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

At this stage everything is set up for you to write your first Java program which uses GUIs. Click the **New File** icon on the JCreator IDE and save the file as **JOptionPaneOutput.java** in your Lab3 folder. Now 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.

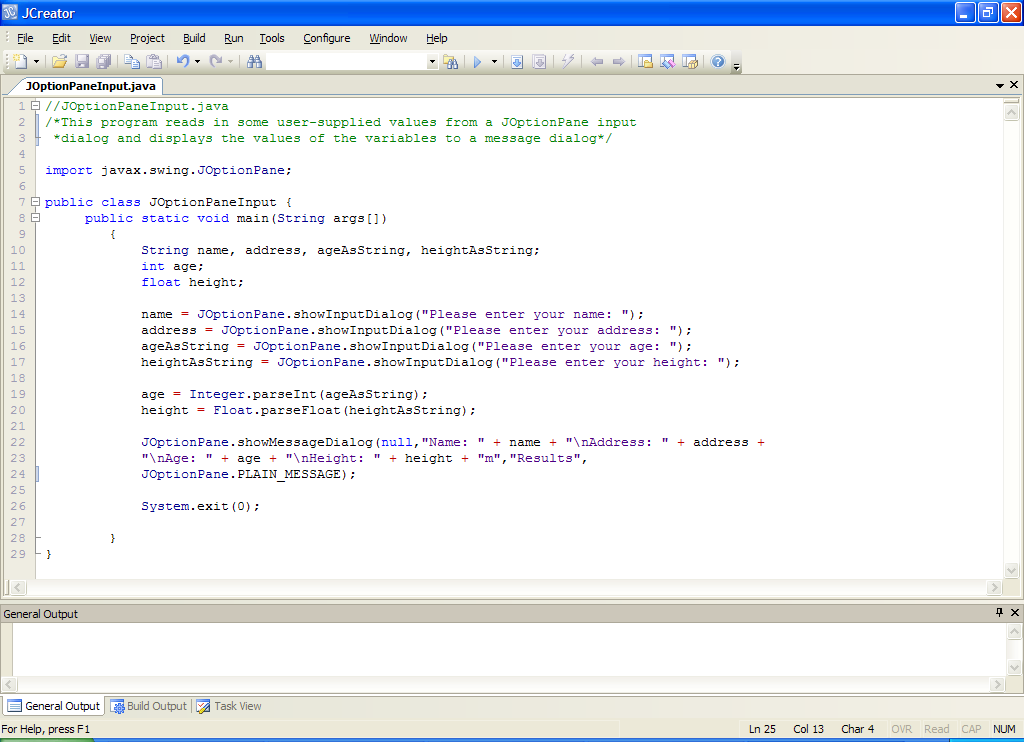
Once you are free from errors, run the program. In this case it will display the message in the JOptionPane window as indicated below:



**GUI Input in Java**

**Aim**: To read in some user-supplied values using a GUI window component

**Java Code**:

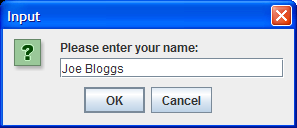


**Program Analysis**:

• After the comments, an import statement occurs so that we can use the class JOptionPane in our program.

• 4 string variables are declared, 1 integer variable and 1 floating-point variable.

• On line 14, a call to the method **showInputDialog**() is made. This method belongs to the class JOptionPane. The method is used to display a **prompt window** for getting user input. The method takes just **1 argument** here – the prompt message you wish the user to see. At runtime, when the user has entered their name and clicked the OK button (or hit return), the value entered will be stored in the variable name. The input dialog appears as follows with a text-field for you to enter your data:



Like the showMessageDialog() method, the call to showInputDialog() **halts the programs execution**. Once you have entered the information, just press **OK** or **hit return** to continue. The word “Input” appears in the title-bar by default and the question-mark icon also appears automatically to the left of the prompt.

• Lines 15-17 work in an identical manner to the previous one. One important point to note about the showInputDialog() method is that it can only read in strings. This means that when you enter a value, **even if it is numeric** in nature, it is always taken in as a String. This is the reason why the variables ageAsString and heightAsString have been declared – just to store the String representations of these numeric inputs.

• On line 19, the code

**age = Integer.parseInt(ageAsString);**

uses the method **parseInt**() to **convert** the string version of the age into an integer version. Actually, this isn’t necessary for this particular application but if you needed to process the value as an integer, this step would be essential – we will see this shortly in a program that uses arithmetic.

• Line 20 works in the same manner, converting the string version of height to a floating-point version through the method **parseFloat**().

• Lines 22-24 contain the code for the showMessageDialog() method and this simply displays the values of the variables to the output dialog window. One slight difference from the last time we used it is that now we have used the **PLAIN\_MESSAGE** icon in the window.

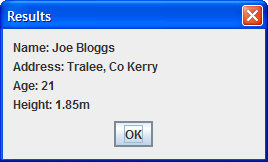
• The exit() method call ensures the program terminates correctly.

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

Click the **New File** icon on the JCreator IDE and save the file as **JOptionPaneInput.java** in your Lab3 folder. Now 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 supply the information required. It would then produce the following message dialog:



**Exercise 1**

Write a Java program that presents the user with a menu of options as follows:

1. Calculate the area of a triangle
2. Calculate the volume of a cone
3. Quit

When the menu appears the user will be asked to enter their preferred choice as a number (1-3) and the program will proceed to carry out the appropriate action. In executing choice 1 the user will be prompted for the lengths of the 3 sides of the triangle - a, b and c. If the user selects choice 2 , then the user will be prompted for the radius, r, and height, h, of the cone. The formulae below should then be used to determine the area or volume as required. All results from the program should be displayed to **3 decimal places**. If the user selects choice 3, then they should receive a farewell message. In the event that the user enters a value outside the 1-3 range then an appropriate **warning message** should be issued. Your program should use a **switch statement** in its coding. You **don’t need to use a loop** here, the menu options will appear only once.

**N.B.** The formulae to be used here are as follows:

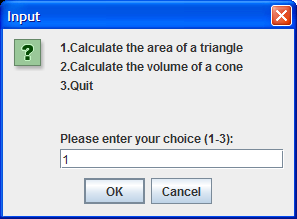
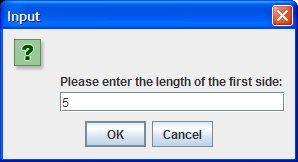
area of triangle = where s =

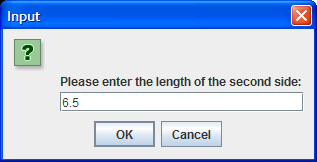
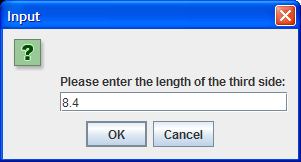
volume of cone =

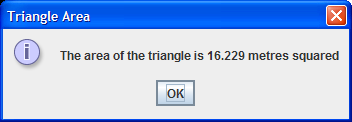
π=3.142 and should be declared as a **constant** in your program

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

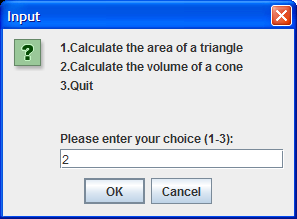
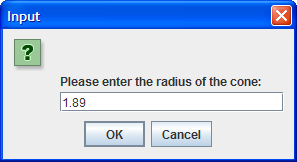
Run 1:

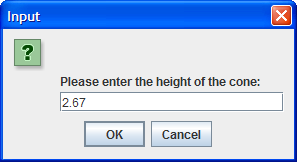
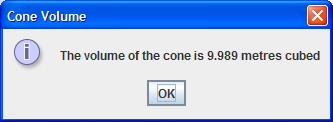
 

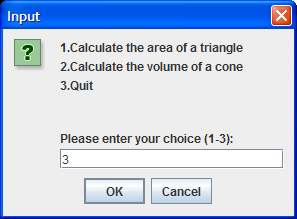
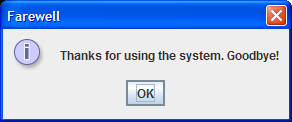


Run 2:

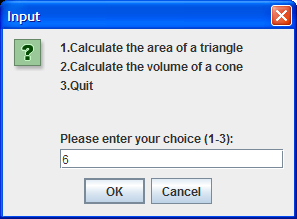
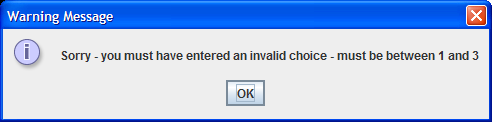
 

Run 3:

Run 4:

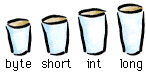
 

**Classes, Objects and Object References in Java**

Recall from your first practical that Java is an **object-oriented** programming language. This is considered a **serious advantage** of Java. Although we have been using some object-oriented features of Java to date, such as packages and classes, we haven’t really gone into any details about object creation in Java. This all changes now since the very next program we shall examine requires a little understanding about objects, classes and object references in Java.

**Ordinary variables and objects are similar in some ways**. They both take up a certain amount of memory. However they differ in that ordinary variables are created from the primitive data types such as int, char, float and double, whereas objects are created from more complex data types called classes.

When you create an ordinary variable and give it a value, that is all there is - a **single value**. Think about an ordinary variable in terms of a cup, holding a single value.

Each of these cups can hold different types of value and each can hold a certain, maximum size of value – therefore the size of the cup is known in advance of putting something into it. long cups hold more than int cups which in turn hold more than short cups etc. At the end of the day though, all these **primitive data type cups can only hold a single value each**.

The situation with objects is quite different. An object is created from a class and a class can be as simple or as complicated as we want it to be. If it is really complicated, then the objects we create from it are equally complicated. **Objects can grow (or shrink) at runtime** also so we **cannot just allocate a set amount of memory for an object** – it must be able to grow. This is one way in which an object is different from an ordinary variable, which has a fixed size.

Also, objects not only have one value associated with them, they can often have 10 or 20 or more values associated with them! Complicated **objects can have hundreds of values** associated with them! In addition to this, **objects will also have at least one method** associated with them. **Ordinary variables only have one value and no methods associated with them**.

In Java, an object is created with the **new** keyword as follows:

***Classname objectName = new Classname(arguments);***

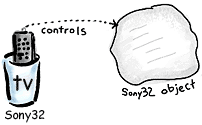
*Classname* is just the name of the class you want to create the object from and *objectName* is the name of the **object reference** associated with the object created. *arguments* has the same meaning as always and refers to the pieces of information that method *Classname()* requires in order to do its job. You’ll have seen this type of code already in relation to the Scanner class i.e.

Scanner input = new Scanner(System.in);

This is where it gets interesting. The object you have just created here **does not have a name** of its own – the ***new Classname(arguments****)* part of the statement **is the object**. In the diagram below, the line of code would be, for example,

Sony32 tv = new Sony32();

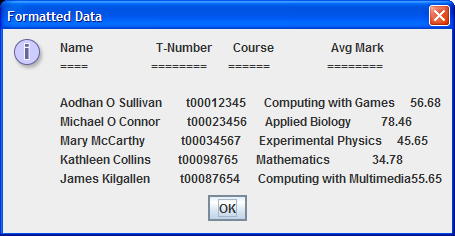
You can think about the object as being a “cloud”, like the Sony32 object cloud on the right below.



When an object is created, we need to be able to refer to it, so that we can process it in some way. This is done by using an **object reference variable**. In the diagram above, the object reference is called **tv** and it is a variable of type **Sony32**, the same type as the object it is referring to. You should think of an object reference variable as being just like a **remote control**. It allows you to control the object it is referring to. So here the remote control is the tv object reference variable, pointing at the Sony32 object. Because it is an object reference **variable**, it can easily be changed and set to control some other object in the future.

**Formatting the JOptionPane Message Dialog**

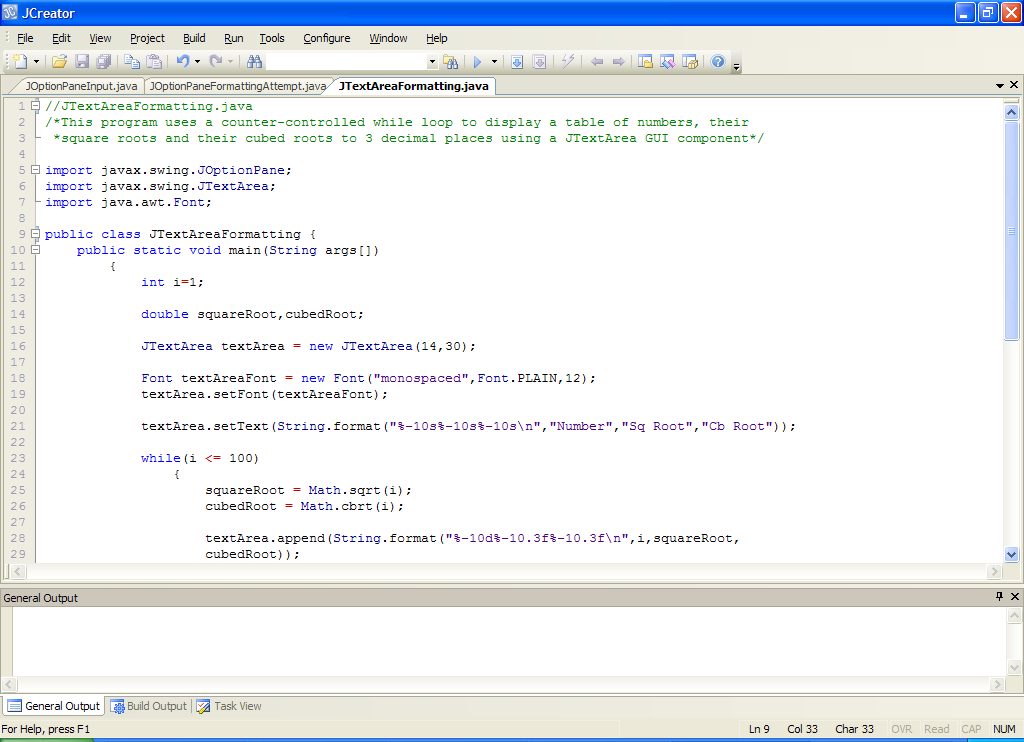
One of the drawbacks of the JOptionPane message dialog component is that formatting it directly is a problem. The main issue here is that the font used by the component (Arial) is not a monospaced font. To make matters worse, it is not possible as a programmer to change the font used by the message dialog component. Therefore, if you try to format data in a nice neat tabular way using **String.format**(), it can render the information badly e.g. when I run the **JOptionPaneFormattingAttempt.java** program, I get the following type of output (you can run this program yourself now if you wish, it is in the LabSheet3 folder):

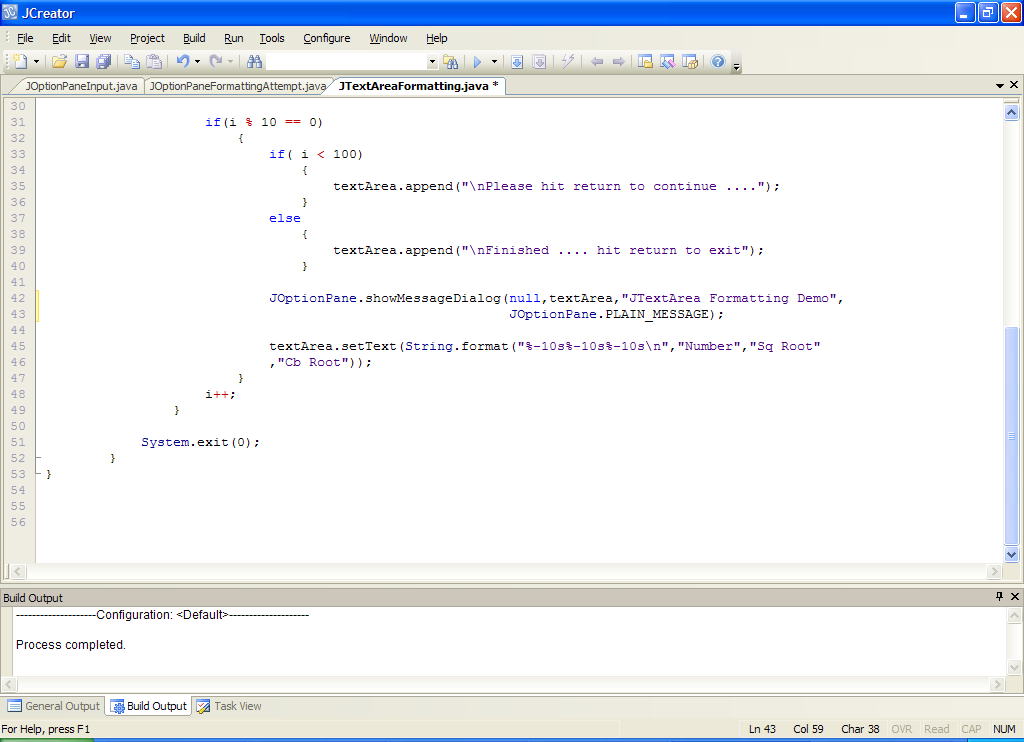


However, there is always a solution to a problem in Java! There is another GUI component called a **JTextArea**, and this component can be manipulated in terms of its font etc. The following program illustrates how the JTextArea can be used to display tabular information within a message dialog.

**Aim:** To write a program which uses a counter-controlled while loop to display all the integers from 1 to 100 along with their square roots and cubed roots in a neat tabular format using a JTextArea in conjunction with a message dialog.

**Java Code:**



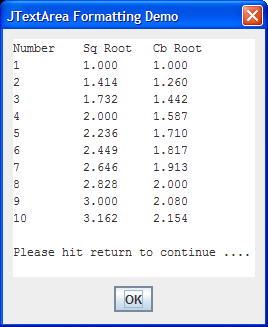


**Analysis of program:**

• The program begins by importing 3 Java classes – JOptionPane as we have done before, but also two other classes. **JTextArea** is, like JOptionPane, a Java GUI component. This component is a **multi-line area that is capable of displaying plain text**.

Note that up until now we have been using the showMessageDialog() method from the JOptionPane class to display our graphical outputs. This technique has been perfect so far because all of our output has been relatively small. An issue arises with this simple message dialog when the amount of output gets larger. The message dialog will actually “grow” to accommodate the amount of output but, if it gets too much, the user will end up **scrolling** down the window to see all the output. Recall what we have said many times before about scrolling – if possible, it should always be avoided.

In this program we are attempting to display 100 lines of output in tabular format and the aim is to break up the output into nice bite-size chunks of 10 lines per screen. This breaking up of the output could be done with the message dialog also (we could have the message dialog appear 10 different times) but using a **text area** gives us a **greater level of control**. Our output window will appear as follows when the application is run



For this, we will use a **JTextArea** in conjunction with the message dialog (the text area is effectively placed onto the message dialog). The beauty of this solution is that **we can use the same text area over and over again** – we don’t need to create 10 of them. Once we have filled the text area with the first chunk of output we just clear it and start over with the next chunk.

As you can see from the code, the JTextArea class belongs to the **same package** as the JOptionPane class – **javax.swing**. In fact, all these predefined Java GUI classes beginning with the letter ‘J’ belong to this package. We will see more of them in future labs.

• The class **Font** belongs to a different package called **java.awt**. There is a specific reason we need this class in the application here. When JTextArea is used to display information, it uses a default font that is “proportional” in nature. This simply means that different characters take up different amount of space width-wise. Since our aim here is to produce a nicely aligned tabular output, we must use a “fixed pitch” font instead where every character takes up the same space width-wise. The one we will use for this purpose is the “**monospaced**” font. You will see this being set later in the code.

• Within the main(), the first thing that happens is that the while loop counter variable, i, is initialised to 1. Recall that, by **convention**, loop counters are often called i. They don’t have to be, of course.

• Two variables called squareRoot and cubedRoot are then declared. These will be used to store the square root and cubed root of all the numbers from 1 to 100 as the loop progresses.

• On line 16, the code

JTextArea textArea = new JTextArea(14,30);

**creates an** **object** of type JTextArea using the method JTextArea(). The initial dimensions of the text area are controlled by the arguments 14 and 30. Here the 14 means the text area will be able to hold 14 lines of output and the 30 means that there will be enough space to accommodate 30 columns of output. These numbers don’t matter that much though as the text-area can “grow” to accommodate the text written to it.

The JTextArea object will be controlled by the object reference variable called **textArea**, whose type is also JTextArea. We will be using this object reference to control the text area object very shortly.

• On line 18, the code

Font textAreaFont = new Font("monospaced",Font.PLAIN,12);

Creates a Font object using the method Font(). The type of the font is set to “monospaced”, the style of the font is set to plain font and the height of the font is set to 12 point. This Font object will be controlled by the object reference variable called **textAreaFont**.

It is our intention here to make the font associated with the text area a “fixed pitch” font so that our output gets aligned nicely in the text area. monospaced font achieves this goal.

• The next line of code

textArea.setFont(textAreaFont);

deals with setting the type of font associated with the text area object to the values just mentioned. This is achieved by calling the method **setFont**() on the text area object reference textArea, and passing in textAreaFont as an argument. This is real OO code in operation here.

Note that this is a perfect example for showing how an object reference variable differs from an ordinary variable. An object reference variable can have methods called on it (those methods associated with the class that its object has been created from) whereas ordinary variables can never have methods called on them – a syntax error would result e.g.

int age = 7;

age.setValue(6); // syntax error here

• The next line of code

textArea.setText(String.format("%-10s%-10s%-10s\n","Number" ,"Sq Root","Cb Root"));

uses the method **setText**() to place some text on the text area. As usual, this is achieved by calling the method on the text area object reference variable, textArea. Recall the **String.format**() method we have used many times before to dictate exactly how values will be formatted. Hopefully you can recall what the format specifiers here are doing. This line of code gives a heading to our table of values as in the screenshot earlier.

• After this we are into the while loop. The loop begins by testing the value of the variable i against the value 100. As long as the value of the loop counter variable remains less than or equal to 100, the loop will keep going. This gives us the 100 iterations we need here. Each time the loop iterates, the value of the loop counter increases by 1 with the code **i++.**

• The next 2 lines of code

squareRoot = Math.sqrt(i);

cubedRoot = Math.cbrt(i);

deal with finding the square root and cubed root of the number i. The methods **sqrt**() and **cbrt**() do this. You met sqrt() before but both are part of the **Math** class. The values returned by these methods are then stored in the appropriate variables. One thing to note here is that the values returned by the Math class methods are of type **double**, so it is vital here that the variables squareRoot and cubedRoot are also of type double rather than of type float. If you were to make them float, then a “**possible loss of precision**” syntax error would occur. This is because a 64-bit double value cannot be squeezed into a 32-bit float variable.

• The code

textArea.append(String.format("%-10d%-10.3f%-10.3f\n",i,squareRoot,cubedRoot));

uses the **append**() method to join onto the end of the text area the number, its square root and its cubed root. This is formatted by String.format() so that it is aligned correctly. Each time the loop iterates it will add another line of output to the text area object. Note again that the **textArea object reference is being used to control the text area object**.

• The code

if(i % 10 == 0)

may look a little odd at first glance. It uses the remainder operator to determine whether or not the value of i is exactly divisible by 10. If it is then i % 10 equals zero. The reason for this if statement is to allow us to control the output display. We want the text area to only display 10 lines of output at a time. So we’ll get the numbers from 1-10, then the number from 11-20 etc. In order to achieve this effect, we must be able to manipulate the text area after every 10 lines of output have been appended to it.

When every 10 lines of output have been appended and, provided the value of i is less than 100, then we also append the message “Please hit return to continue”. If however, the value of i is 100 we naturally don’t want to put out that message – instead we put out the “Finished” message instead. This is achieved with the if-else.

• After the if-else, the showMessageDialog() method is used to display our output as usual. The textArea object reference is passed to the method where we normally pass in text and variables. This has the effect of placing the contents of the entire text area on the output window.

As usual, when the showMessageDialog() is called, the message displays and the program halts until the user either presses the “OK” button or hits return.

• We use this fact in the program to effect in the next line of code

textArea.setText(String.format("%-10s%-10s%-10s\n","Number","Sq Root","Cb Root"));

When the output displays the user sees the 10 lines of output and then the message “Please hit return…” Once they hit return, we wish to clear the contents of the text area for some fresh output. If we don’t do this, the existing output will still be there. The contents of the text area is cleared simply by setting the text of the text area using the **setText**() method. Notice that the text itself is exactly the same as on line 21 – giving us the heading for our table.

• After this the value of the loop counter variable is incremented and the loop reiterates until it reaches the value 101.

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

Take some time to digest the program above as there is a lot in it. When you are done, click the **New File** icon on the JCreator IDE and save the file as **JTextAreaFormatting.java** in your Lab3 folder. Now 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.

**Exercise 2**

Write a Java program that uses a counter-controlled **while** loop to read in the details of exactly 5 students. The details should then be displayed in a neat tabular fashion via the **format**() method and a **JTextArea** as follows. You can take it that the average mark gets displayed to **2 decimal places** here.

