**Institute of Technology Tralee**

**Computing Department**

**Structured Programming 2**

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

**Practical 14 – Basic GUI Components**

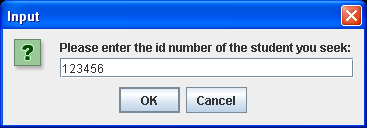
At this stage you have covered all the essentials of structured programming in Java. Using what you have learned so far, you should (in theory!) be able to design, code, debug and test practically any “small-sized” console or GUI input/output based application in Java. In today’s lab, the focus will be on getting you used to working with some other **GUI components** – the good news is that many of them are very similar in their operation and, because they are pre-defined classes, they are ready for us to **re-use** at will.

**Why use GUIs?**

The reality is that nowadays most real-world applications are GUI-based. The main reason is that they are considered more **user-friendly** than an equivalent console-based application. By providing applications with **consistent**, **intuitive** user interface components, we allow users to more easily become familiar with our application so that they can learn to use it more quickly and also use it more productively. Also, most people consider them to be nicer to look at than a console application, so **aesthetics** is important.

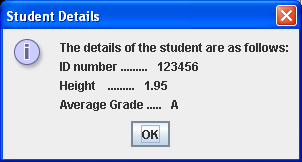
**Simple GUI-based Input/Output with JOptionPane**

As you know, we have already been using some GUI components in this module – in particular the class **JOptionPane**. This class was very useful for us in order to interact with the user in a GUI manner; for prompting for information – with showInputDialog() – and displaying output – with showMessageDialog(). The components used here are normally just called **dialogs**.



The screenshot above shows a typical JOptionPane input dialog. It has a title bar (where the text “Input” appears along with a button allowing you to close the dialog), an icon (the box with the question mark), a label (which contains the prompt message “Please enter ….”), a text-field (where the user-input is typed) and 2 buttons (“OK” and “Cancel”).

So, as simple as the dialog looks, there are quite a few GUI components in it.



The screenshot above shows a typical JOptionPane message dialog. This also has a title bar, an icon (The circle with the “i” in it), a label containing all the text displayed and a button (with “OK” on it).

You may recall that sometimes my message dialog had no icon. This all depended on whether I passed in **JOptionPane.INFORMATION\_MESSAGE** or **JOptionPane.PLAIN\_MESSAGE** as the fourth argument when calling the showMessageDialog() method. The “plain message” would mean **no icon** while the “information message” would mean the “i” icon you see above.

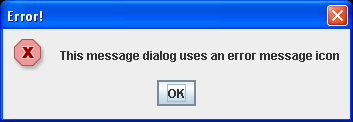
Note that fields such as INFORMATION\_MESSAGE and PLAIN\_MESSAGE are examples of **constants** that have been defined within the JOptionPane class. You can tell they are constants because constants in Java, by **convention**, are always written in **UPPERCASE**.

**Organising your Work**

You should have a folder under X: called SP2Stuff created. This time, create a folder called **Lab14** within SP2Stuff to save your work from this lab session.

**Exercise 1**

There are other possibilities for the message dialog icon. You should now look up the **Java API documentation** and locate the **JOptionPane** class. Then you should write a short program called **Exercise1.java** which simply uses each of the other 3 icons that you have not seen before. The program should run exactly as indicated in the following sample screenshots – so note that the main() of this program contains **only 4 lines of code** in total, with no variables needed at all!



****

****

**Exercise 2**

Have another look at the Java API documentation for the JOptionPane class. You will see that there are not only methods for displaying a message dialog or an input dialog, but there are also methods that allow the programmer to display a **confirmation dialog** to the user. One of these is defined as follows in the API:

|  |  |
| --- | --- |
| static int | [**showConfirmDialog**](http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html#showConfirmDialog(java.awt.Component, java.lang.Object))([Component](http://java.sun.com/javase/6/docs/api/java/awt/Component.html) parentComponent, [Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html) message)            Brings up a dialog with the options *Yes*, *No* and *Cancel*; with the title, **Select an Option**. |

Note that this method expects **2 arguments**. You should just make the first one **null** and the second one will be the message you wish to display on the dialog. For me here it is “Take your pick ….” as indicated in the input screenshot below.

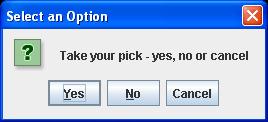
You should now write another short program called **Exercise2.java** that uses the method above to display a confirmation dialog.

When the user interacts with this component at runtime, then there are a number of possible outcomes from the method. Note that the **method returns an integer**. The value of this integer will be one of the following static constants:

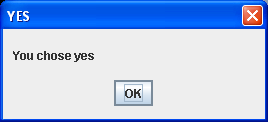
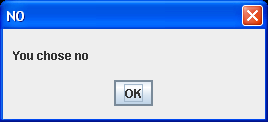
* YES\_OPTION
* NO\_OPTION
* CANCEL\_OPTION
* CLOSED\_OPTION

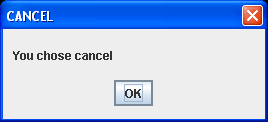
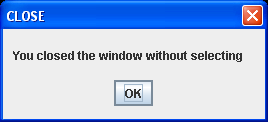
Your program will test against these values and display an appropriate message using a message dialog, indicating the option the user selected, so there will be some decision-making code in your application. Your program should run as indicated in the following sample screenshots:

**Input Window:**



**Possible outputs:**

 ****

**** ****

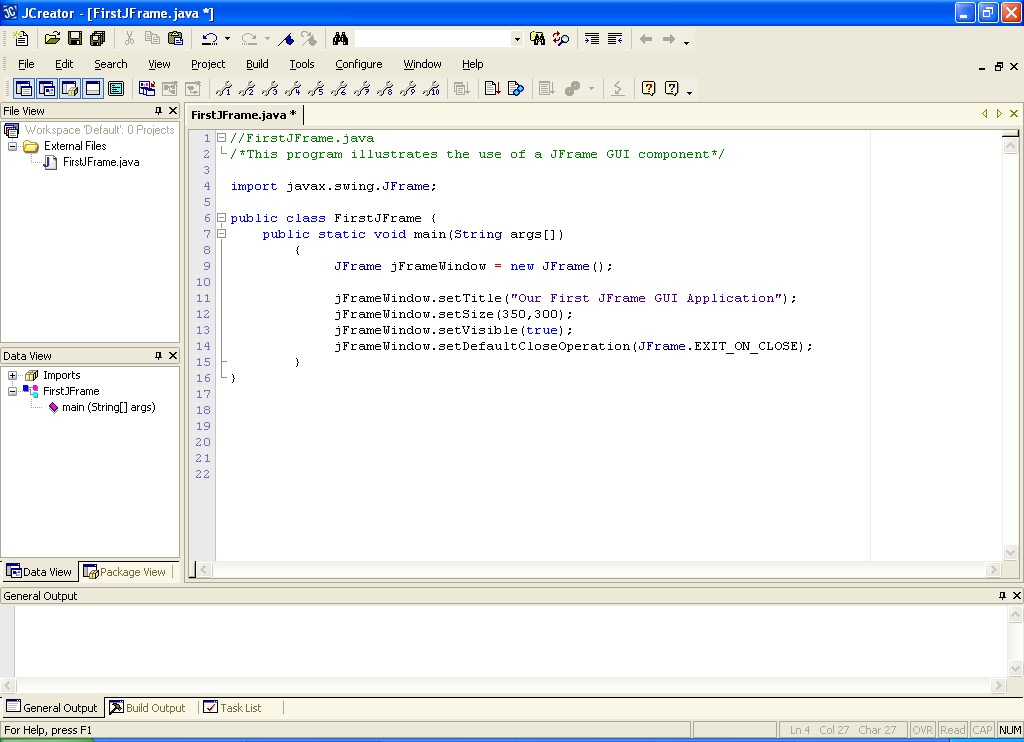
**The Limits of JOptionPane**

As good as JOptionPane is, believe it or not it is quite limited and restricts us as programmers in the sense that the dialogs we have been using were already written by another programmer – their functionality has been decided for us in advance. This was fine for us up until now though and, in the future, we will still continue to use these dialogs, as they **save us a lot of coding effort.** However, most “real-life” GUI applications require more elaborate, **customised** user interfaces. We will now look at some GUI components that allow the developer to create such applications.

**Creating a Simple Window with JFrame**

**Aim:** The next program illustrates how to create our own window-based application using a GUI component called a **JFrame**.

**Java Code:**

****

**Program Analysis:**

● The program begins by importing a class we have not used before. This is called **JFrame**. It belongs to the **javax.swing** package, as do most GUI components we will use.

● The first line of code in main() **creates a JFrame object** with the code **new JFrame()** and then this object is assigned to the reference variable (**object reference**) called **jFrameWindow**. So, if we wish to do anything with our JFrame object in terms of manipulating it, we must do it by using the “remote control” jFrameWindow.

● The next 4 lines of code then manipulate the JFrame object.

The first one uses the method called **setTitle**() to set the text in the title bar of the JFrame window to “Our first …..”.

When a JFrame window is originally created it does have a title bar but it has no title by default. You may have noticed, for example, that the input dialog created from the JOptionPane always had the default text “Input”. This setTitle() method allows us to control the text in the title bar.

● The next line of code uses the **setSize**() method to dictate what the original size of the JFrame window will be when the program runs. In this case the window will be set to 350 pixels in the horizontal direction by 300 pixels in the vertical direction.

When a JFrame window is originally created its default width and height is 0 pixels by 0 pixels so there is actually no window “pane” visible at all, just a title bar! Therefore we must set the dimensions of the window as appropriate.

● The next line of code calls the **setVisible**() method in order to make the JFrame visible to the end-user. This might sound unexpected but, by default, windows are invisible i.e. the window object actually exists but you just cannot see it – it is “hidden” from the user. In order to see the window, you must explicitly call the setVisible() method to achieve this. Passing in the boolean value true makes this JFrame window appear on screen, as required.

● The last line of code uses the **setDefaultCloseOperation**() method to decide what happens when the user clicks on the close button in the title bar of the JFrame window. If you think about what normally happens when you do this in a running application, such as Microsoft Word or JCreator, the application immediately terminates, as long as there are no unsaved files being processed. Otherwise, you normally get a dialog asking whether you wish to save the changes to any open files.

In our case here, we simply wish to terminate (exit) the application when the user clicks on the close button (we have no open files to worry about here). To do this, we just pass in the constant value **JFrame.EXIT\_ON\_CLOSE** to the method. This makes the application terminate immediately, and is used by us in place of **System.exit(0)** for our GUI-based applications.

The default behaviour when closing a JFrame window is to simply **hide the window** – just like calling **setVisible(false)**. This makes the window disappear but the application is still actually running which is very deceptive and can cause problems as the program is still using up CPU time and memory even though we are effectively finished using it.

So you see that, although this program is very short, there is lots to talk about. However, what was said about this program is not wasted because you will see this type of code many times from now on.

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

Click the **New File** icon on the JCreator IDE and save the file as **FirstJFrame.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. Click on the close button in the title bar to terminate the application.

**Exercise 3**

You should now look up the Java API documentation for the JFrame class. In the “**field summary**” section, you will see the reference to the constant **EXIT\_ON\_CLOSE** we used in the last program. In the “**method summary**” section you will see the **setDefaultCloseOperation**() method that we used in the last program. Interestingly though, there is no sign of the **setTitle**(), **setSize**() or **setVisible**() methods in this section despite what you might have expected. So where are these methods?

If you scroll down a little further you will come across a section that says “**Methods inherited from class java.awt.Frame**”. In this section you will see an entry for the method setTitle(). Just below this is a section that says “**Methods inherited from class java.awt.Window**”. In this section you will see entries for the setSize() and setVisible() methods.

Kind of confusing? No doubt it is a little. The thing about Java is that, as an object-oriented programming language, there is lots of **software reuse** going on. So one class, like JFrame, can reuse other classes such as Frame and Window. This is what is happening here with the methods setTitle(), setVisible() and setSize(). All of these methods are actually defined within other classes but, because the JFrame class has itself been defined in such a way to re-use these other classes, it **has access to practically all of the methods** (and fields/constants) defined within those other classes. This concept is called **inheritance** and is a very important OO topic that you will spend a share of time examining in your OOP and Algorithms & Data Structures modules in year 2. For us, it is enough to know that inheritance exists as part of the Java language and that we can simply take advantage of it when coding our GUI applications.

**Exercise 4**

You should now save the FirstJFrame.java program as **Exercise4.java** and make the following modifications to it. Note that you will need to use the **Java API documentation** in order to find some of the methods required.

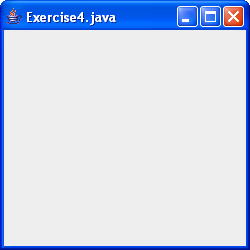
● The text in the title bar should be set to “Exercise4.java”

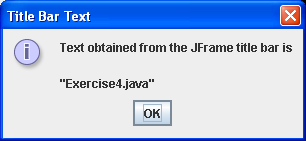
● The size of the window should be set to 250 pixels in the horizontal direction and 250 pixels in the vertical direction.

● The location of the window should be set to the location (150,150) on the screen when it appears.

● The text in the title bar of the window should be obtained from the window and displayed in an information message dialog (have a quick look at the **Frame** superclass for the method needed here)

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





Note that the application here can only terminate when you have first clicked the OK button on the JOptionPane dialog. This is because, as always, the JOptionPane **dialog halts the programs execution until you interact with it** by clicking on the OK button.

**A Word on Instance and Class (static) Methods**

The application above demonstrates many examples of so-called **instance methods**. For example,

setTitle(), setVisible(), setSize() and setDefaultCloseOperation()

are all examples of these instance methods. The reason they are called instance methods is simply because an **instance of a class** (another phrase for an **object**) must exist in order for them to be called. Here the class is JFrame, the instance of the class (the object) is created using **new JFrame()** and the methods are called on this instance through the JFrame object reference **jFrameWindow**.

When an instance method is called, it takes the following general form:

**objectReference.instanceMethodName(arguments)**

You will see many more examples of instance methods as we go on.

The alternative to instance methods are **class methods**. A class method is one which is called directly on the class it belongs to, rather than on a specific instance of that class. We have seen many examples of class methods already. For instance all the methods contained in the **Math** class are class methods. They are often referred to as **static** **methods** also. When we call one of these methods, such as pow(), we do so as follows:

**Math.pow(5,2)** //calculate the value of 5 to the power of 2

Notice how pow() is called directly on the Math class here.

When a class (static) method is called, it takes the following general form:

**ClassName.classMethodName(arguments)**

You will see further examples of class methods as we go on.

**Determining Whether a Method is an Instance or Class Method**

The important thing to realise as a programmer is that, when you are making use of pre-defined methods, **you need to know whether or not the method concerned is an instance or a class method in order to call it correctly**. This is very easy to determine from the **Java API documentation**, however. For example, the entry we had earlier for the showConfirmDialog() method is as follows:

|  |  |
| --- | --- |
| static int | [**showConfirmDialog**](http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html#showConfirmDialog(java.awt.Component, java.lang.Object))([Component](http://java.sun.com/javase/6/docs/api/java/awt/Component.html) parentComponent, [Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html) message)            Brings up a dialog with the options *Yes*, *No* and *Cancel*; with the title, **Select an Option**. |

Note the word **static** on the left – this tells us that the method is a class method and so we call it directly on the class it belongs to, in this case the class JOptionPane.

Consider now the following method from the String class. As there is **no static keyword** on the left here, it must be an instance method. In fact, most of the methods in the String class are instance methods.

|  |  |
| --- | --- |
| boolean | [**equals**](http://java.sun.com/javase/6/docs/api/java/lang/String.html#equals(java.lang.Object))([Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html) anObject)            Compares this string to the specified object. |

One static method from the String class that we have already used is the **format**() method.

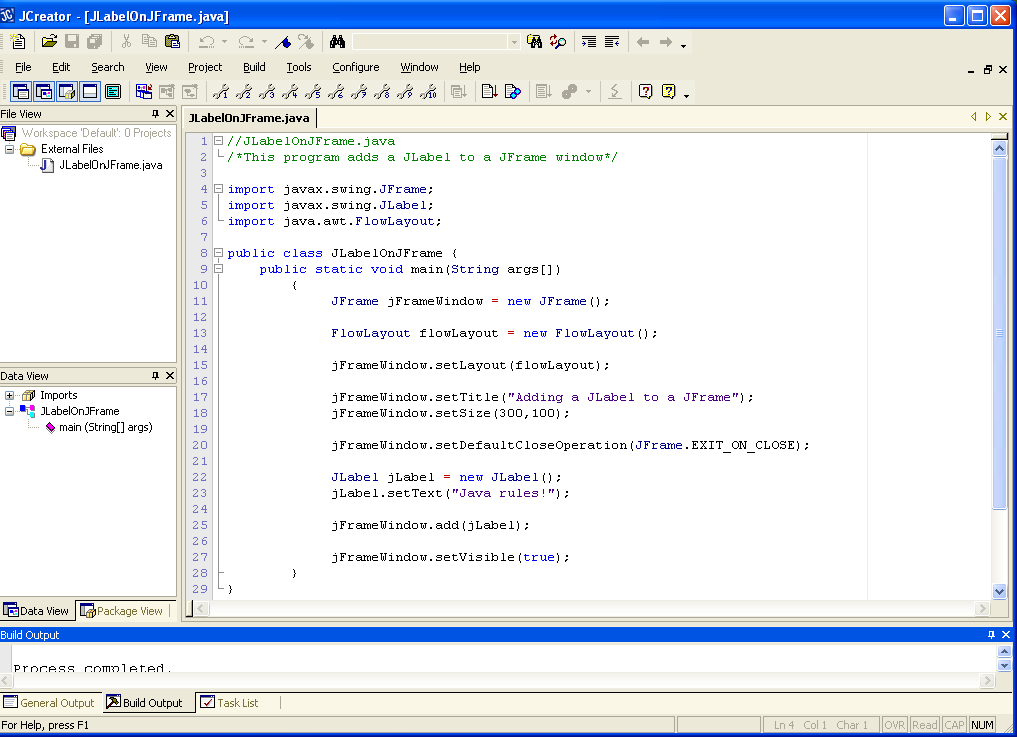
**Adding a JLabel to a JFrame Window**

The original program on JFrame had a window “pane” but it was empty – there were no GUI components added to it. Normally, GUI windows contain several other GUI components such as buttons, text-fields etc. If you think back to the “simple” JOptionPane input dialog, its window contained a text-field, where you inputted your values, and 2 buttons, as well as an icon and a label.

The next example we look at just adds a label to the JFrame window and places some text on the label. Labels are generally used in GUIs for **prompt messages** and for **displaying output**.

**Aim:** This program takes a JFrame window and adds a JLabel GUI component to it. It also uses a layout manager.

**Java Code:**



**Program Analysis:**

● There are several different classes imported at the top of this program. JFrame is there as before but this time there is another GUI class called JLabel. After this there is a class called FlowLayout, which belongs to the package **java.awt**. This is another package that is practically always required when developing GUI applications.

● We begin again by creating a JFrame object. This ultimately becomes our window when the application runs.

● The next line of code creates a **FlowLayout** object. When GUI components are added to the “content pane” of a window, they are added in a certain way by default. A **layout manager** is used by windows such as JFrame to add components to its content pane and to ensure that the components appear in a reasonable manner should the window be resized etc. The default layout manager used by JFrame is called the BorderLayout manager and, although this is a useful layout manager, it is not the best for this particular demo, so I have decided to use a different layout manager called FlowLayout, which we will use with all our GUI applications in this module.

In order to dictate that the layout manager style for the content pane will now be flow-layout rather than the default border-layout, we must set this up as follows:

jFrameWindow.setLayout(flowLayout);

The setLayout() method is called to set the layout manager style for the content pane to flow-layout.

● The next few lines of code should look familiar to you from the last time. The text in the title bar, the size of the window and the default window-closing operation are all set.

● The next line of code creates a JLabel object and the text associated with this label is then set up with the setText() method call.

● The label we just created is then added to the content pane itself with the code

jFrameWindow.add(jLabel);

The add() method is one you will see a lot as we use it whenever we need to add components to a window.

● The last line of code just sets the visibility of the JFrame window to true in order to make the window visible to the end-user. Note that this really **should be the last thing you do** in a GUI application. For example, if you were to make the window visible before adding the JLabel, then the label would not appear on the window at all even though it has been added. To make it appear, you would need to either call yet another method to effectively update the window or else physically move/resize the window via the mouse.

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

Click the **New File** icon on the JCreator IDE and save the file as **JLabelOnJFrame.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. Click on the close button in the title bar to terminate the application.

**Exercise 5**

In the last program there are a few ways in which it might be improved, at least in terms of the number of lines of code it uses. The following lines of code:

jFrameWindow.setTitle("Adding a JLabel to a JFrame");

jLabel.setText("Java rules!");

although perfectly valid, are not necessary in this program. We could set the JFrame title-bar text and the JLabel text using alternative means which can spare us some coding.

Look up the **Java API documentation** for the JFrame and JLabel classes to find out how you can get away without the lines of code above. You should focus your attention on the section of the pages entitled “**Constructor summary**”.

You should then save the JLabelOnJFrame.java program as **Exercise5.java** and see if you can make the necessary modifications. Your program should run exactly the same as before.

**Exercise 6**

Take Exercise5.java now and save it as **Exercise6.java**. Now use the **Java API documentation** to find the methods necessary in order to make the following changes to the existing application:

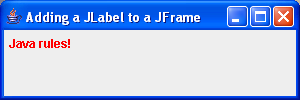
● The colour of the text in the label should be set to red. When you find the correct method for this, it will take the argument **Color.RED**

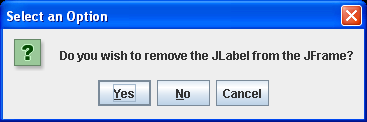
● A **tool-tip** should be added to the label that says “A very important fact!”.

● At the moment the label is centred along the top of the window. You should make it so that it is left-aligned instead. Note that you should look up the **FlowLayout** class to find out how to do this.

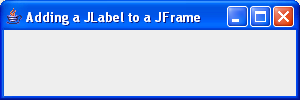
● The user should be presented with a JOptionPane **confirm dialog**. On this they should be asked whether they wish to remove the JLabel from the JFrame. If they select “Yes” then the JLabel should be removed, if they say “No” or just “Cancel”, then it should stay. Note that after removing the label, you also need to call the **repaint**() method on the JFrame window so that the removal actually manifests itself.

The following sample screenshots show how the application should run:

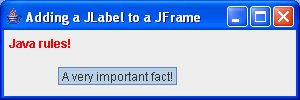




**After selecting the “Yes” option**



**Putting the cursor over the label to get the tool-tip**

****

**The FinalExamTemplateWithDropboxLink.docx File**

You will find this Word document on the X: drive just under the Structured Programming 2 folder. You will have use of this document for your **final exam** in May. On the first page you can see a “template” program that you will **copy** as a basis for your final exam code - the structure of this code probably looks alien right now, but it will become clearer when you have completed the next lab sheet. You can ignore this for now anyway.

After the “template” program, you will see a listing of many of the GUI methods (as well as many other methods you have seen before) that you have used in this lab sheet and will use again over the next two lab sheets. Like with previous CAs, this set of method definitions is available to you in the final exam as a kind of “spell-checker”, to make sure you are coding the names of methods correctly and that you are calling them correctly also. So feel free to have a browse through it over the next few weeks so that you are familiar with its contents – refer to it when you are completing the next two lab sheets if you are having difficulty recalling a method name or how to use it properly.

As with your 2nd Java CA, the file also contains a “dropbox” link that you will use to upload your code in the final exam in May.