TCP2201 Object Oriented Analysis and Design
Lab 3 – Java Swing applications

Lab outcomes

By the end of today's lab, you should be able to

- Make use of Java GUI Swing classes and interface objects
- Make use of Java event handlers (ActionListener) to handle interactivity from user interaction on GUI components and containers.
- combine `java.util` functions into a working application

Java Swing

Swing applications require the Swing package. So always "`import javax.swing.*`" into your code to load it. The code snippets below assume that the class already extends a `JFrame` and has a `JPanel` loaded onto the `JFrame`, i.e. as below. Copy the code below into a file then compile and execute the code. You will see an empty frame with no content.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class SomethingWindow extends JFrame
{
    public SomethingWindow()
    {
        super("This is my application");
        JPanel jp=new JPanel(new FlowLayout());
        add(jp);
        setSize(320,150)
        setVisible(true);
    }
    public static void main(String[] args)
    {
        new SomethingWindow();
    }
}
```

Terminating the application

You can end a Java GUI application either by creating a dedicated component (e.g. close/exit button, exit drop-down menu, etc.) or by clicking the [X] titlebar controls of the application window. Both require slightly different ways of ending the application. For a component that initiates a close, you can capture the event (e.g. `actionPerformed`) and then call

`System.exit(0)` to terminate the application. To handle the window closing event add the following line of code `setDefaultCloseOperation(EXIT_ON_CLOSE);`

Creating a JButton

```
        JButton aButton = new JButton("Push me");
        jp.add(aButton);
```

or

```
        JButton bButton;
        bButton = new JButton();
        bButton.setText("Click me");
        jp.add(bButton);
```

or

```
        jp.add(new JButton("Nothing")); // anonymous obj. style
```

The same applies to JLabel, JTextField, JCheckBox etc.

Creating multiple JButtons using a loop

If you need buttons with consecutive numbers or letters, you could use a loop, for example:

```
        for (int x=0;x<4;x++){
             JButton btnArr = new JButton("Button "+x);
                                         // all buttons here named btnArr
             jp.add(btnArr);} // This jp comes from JPanel above
```

or

```
        JButton[] btn = new JButton[4];
        for (int x=0;x<4;x++){
             btn[x] = new JButton("Buttons "+x);
                                         // buttons named btn[0],btn[1] etc.
             jp.add(btn[x]);}
```

NOTE: The same code above applies to JLabel, JTextField, JCheckBox etc.

Creating and using JOptionPane

JOptionPane is one of the containers found in Swing. It has 4 methods
`showConfirmDialog` → pop up with OK/No/Cancel buttons to ask question
`showMessageDialog` → pop up with message and OK button for information purposes
`showInputDialog` → pop up prompt for user input – saved as string
`showOptionDialog` → pop up combo of previous three methods.

Each method is overloaded with different options, for example
`showMessageDialog(<parentobj>,<message>,<titlebar>,<icontype>);`

For more details see http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

Common methods in Swing components

Some methods that Swing components have in common include:
`setText` and `getText` → change or retrieve the text seen on a component
`addListener` → create a event source which will trigger a listener object (e.g. `addActionListener`)
`setEnable(<boolean>)` → enable/disable a component's methods

Java event management

Event management requires the `java.awt.event` package imported in.

Listeners
`addActionListeners` are for components like buttons and labels. `addItemListeners` are for components that provide choices like checkboxes. `addAdjustmenetListener`/ `addChangeListener` are components like scrollbars that have ranges (i.e. min to max)

Listeners are examples of the **Observer Design Pattern** that will be covered again in the design patterns part of this course.

Adding listeners
Example: First you need to *implement* the correct interface object. If you are going to have only one button in your window, you could make the same class act as the listener by The writing something like:

```
public class SomethingWindow extends JFrame implements
   ActionListener{...
```

However, most of the time, you will have multiple buttons which do different things, so you could have a different class act as the listener:

```
public class AButtonActionListener implements ActionListener{...
```

Since most of the time the individual `ActionListener` classes are never used by any other class than the `JFrame` for which it is designed, it is a common practice to make these classes private inner classes, for example:

```
public class SomethingWindow extends JFrame {
  private class AButtonActionListener implements ActionListener {
     ...
```

Then for each component you want to 'listen' to, add the corresponding listener. Example: To add an `ActionListener` to the buttons declared above.

```
        aButton.addActionListener(new AButtonActionListener);
```

Handling events

Each listener will trigger event handlers. The `ActionListener` will trigger an `actionPerformed` method which must be overridden. Example: If you only had to handle one (1) button, then the event handler would be something like this

```
public void actionPerformed(ActionEvent evt){
     JOptionPane.showMessageDialog(null,"A button was pushed!");
}
```

If you had multiple buttons in a container and all of them have `ActionListeners` attatch a different `ActionListener` to each button:

```
      aButton.addActionListener(new AButtonActionListener());
      bButton.addActionListener(new BButtonActionListener());
```

If the buttons were declared by a loop and share the same variable ID, then make use of the content on/in the component to figure out which button was pressed

```
public void actionPerformed(ActionEvent evt){
     String btn=evt.getActionCommand();  // grab text from button
     JOptionPane.showMessageDialog(null,btn+" was pressed!");
}
```

The same concepts shown above apply to ItemListener, AdjustmentListener and ChangeListener. Check javadoc for more information how to implement these interface objects

EXAMPLE

Here is an example of a Java program that puts up 3 buttons, and does something different for each button. This example is included in the zip file for this tutorial.

- The first button will put up a new window with the MMU logo.
- The second button will play a sound.
- The third button will allow the user to select a file using the standard file dialog box.

ButtonExample.java

```
import javax.swing.*;
import java.awt.*;

public class ButtonExample extends JFrame
{

    public ButtonExample()
    {   super("Button Example");
        JPanel p = new JPanel();
        add(p);
        setSize(250,80);
        JButton pb = new JButton("Picture");
        JButton sb = new JButton("Sound");
        JButton fb = new JButton("File");
        p.add(pb);
        p.add(sb);
        p.add(fb);
        fb.addActionListener(new FileActionListener());
        pb.addActionListener(new PictureActionListener());
        sb.addActionListener(new SoundActionListener());
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

}
```

SoundActionListener.java

```
import java.awt.event.*;
import java.awt.Toolkit;

public class SoundActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

PictureActionListener.java

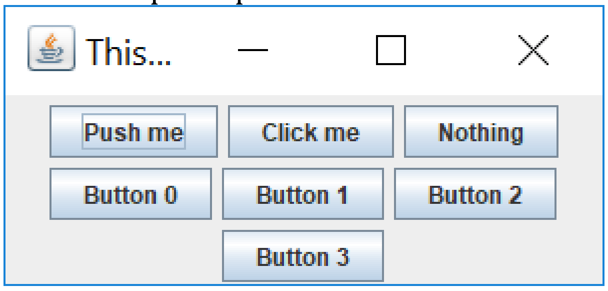
```
import java.awt.event.*;
import javax.swing.*;
import javax.imageio.ImageIO;
import java.io.IOException;

public class PictureActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JDialog dialog = new JDialog();
        dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
        try
        { dialog.add(new JLabel(new
ImageIcon(ImageIO.read(getClass().getResourceAsStream("mmulogo.png")))));
        }
        catch (IOException ex)
        { dialog.add(new JLabel("Couldn't load the image"));
        }
        dialog.pack();
        dialog.setLocationByPlatform(true);
        dialog.setVisible(true);
    }
}
```

FileActionListener.java

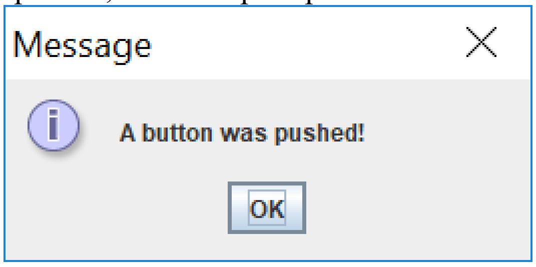
```
import java.awt.event.*;
import javax.swing.JFileChooser;

public class FileActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JFileChooser fc = new JFileChooser();
        int r = fc.showOpenDialog(null);
    }
}
```

EXERCISE 1

Create a Java Swing application that puts up this window:

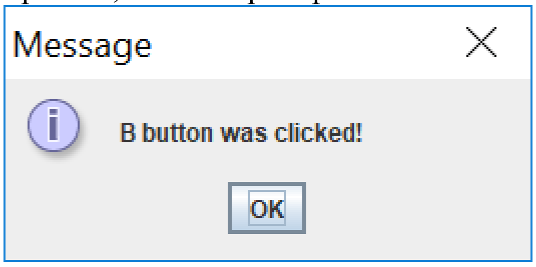


When the “Push me” button is pushed, it should put up this window:



A button was pushed!

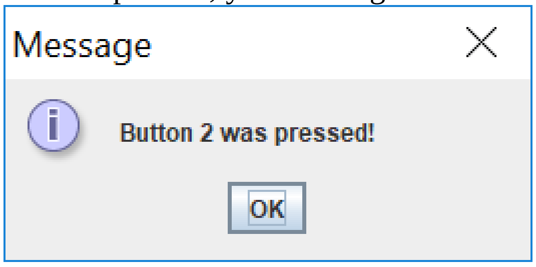
When the “Click me” button is pushed, it should put up this window:



B button was clicked!

When the “Nothing” button is pushed, it does nothing. This illustrates that when you add an anonymous object style button, it is hard to attach an ActionListener to it, so it is hard to ghet that button to do anything.

When any of the numbered buttons are pressed, it should put up a window like this, with the correct number, for example, if button 2 were pressed, you would get:



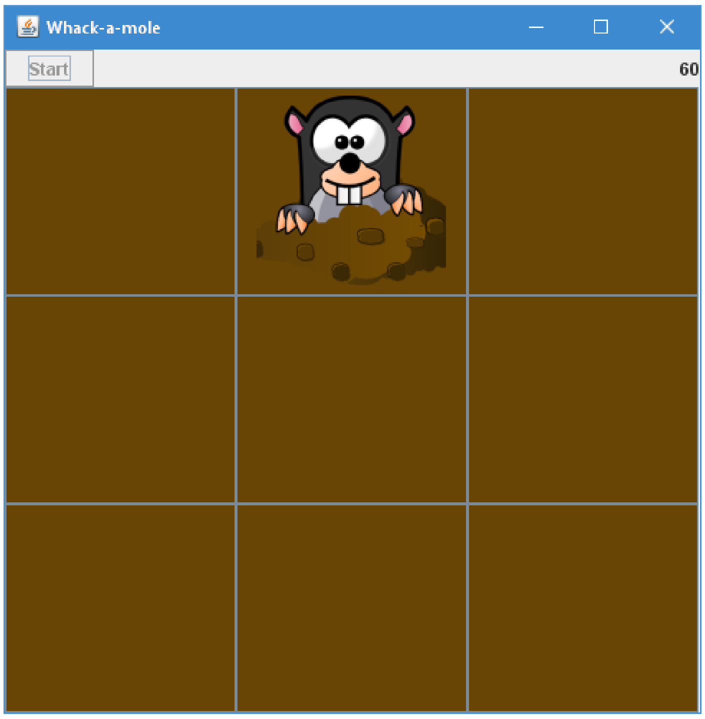
Button 2 was pressed!

All the code for this exercise was provided in the explanation section above. You just need to put it all together into one program.

EXERCISE 2

Using the following requirements, build a single player "*Whack-a-mole*" type game as a Java Swing application. You are required to

- Inherit from a JFrame class for your main application container
- Add a JPanel to house the start button (and optionally a timer) using Border layout manager. *Note: There are two classes called Timer in the standard Java library. I suggest you use javax.swing.Timer as you're doing a Swing application.*
- Add *another* JPanel with the Grid layout manager to get a 3×3 arrangement to the JFrame
- Add nine (9) components as into the JPanel (you can use either JButton or JLabel or any other component you prefer) that represent each grid section on the board
- Add the JPanels to the JFrame subclass
- Implement the correct mouse listener(s) to listen to component activity (button presses and timer starts)
- Add correct listeners to your components that trigger when the mouse is clicked on/in the component and keep track of the successful 'hits'
- Create a timer to change to background colour (or image) of the component when required
- Notify when the game ends and the number of moles hit/whacked
- OPTIONAL: If time permits, add a listener to make use of the number pad on the keyboard to 'whack' a mole. To do this, look up "KeyListener".



An example of how the program could possibly run will be included as a pre-compiled class to you. To execute the program, run the included *Mole.jar* using the Java runtime.

[Disclaimer: Although every precaution has been taken to test for invalid inputs and logical errors, the included class file has not been exhaustively checked for correct operation. Any problems that arise from running the program will not be handled or entertained]

APPENDIX

Useful methods to assist in the exercise:

The following method resizes any image (supported in Java) to a desired size without having to edit the file

```
private ImageIcon loadImage(String path){
        Image image = new ImageIcon(this.getClass().getResource(path)).getImage();
        Image scaledImage = image.getScaledInstance(132, 132,  java.awt.Image.SCALE_SMOOTH);
        return new ImageIcon(scaledImage);
    }
//*************************
```

The following statement creates a Random object that will return a pseudo-random value from 0-16 when called

```
Random rnd = new Random(System.currentTimeMillis());
int newVal = rnd.nextInt(16);
//*************************
```

To play an audio file when an action is performed you need the sound objects from Swing

```
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.Clip;
import javax.sound.sampled.AudioSystem;

public void playSound(String soundName)
 {
   try
   {
    AudioInputStream audioInputStream = AudioSystem.getAudioInputStream(new
File(soundName).getAbsoluteFile( ));
    Clip clip = AudioSystem.getClip( );
    clip.open(audioInputStream);
    clip.start( );
   }
   catch(Exception ex)
   {
     System.out.println("Error with playing sound.");
     ex.printStackTrace( );
   }
 }
//*************************
```

To quickly empty out an array, you can use the functions from the Array object class (req. Array package)

```
Arrays.fill(<arrayname>,<fill value>);
//*************************
```

To change colour of Java components

```
Button.setBackground(Color.GREEN);
Button.setBackground(new Color(4,5,6));
```