# COM6516: Object oriented programming and software design: Practical session 6

This exercise introduces 2D graphics and graphical user interfaces (GUIs) in Java. You should follow up some of this material in your own time, and refer to an Oracle java tutorial listed below and Core Java chapters 7—9. As you work through this material, you will see how the concepts of inheritance, abstract classes, and interfaces turn out to be very useful. Take time to work through this material, aiming to understand what we do at each stage. Here are some useful links –
http://download.oracle.com/javase/tutorial/2d/overview/index.html
http://download.oracle.com/javase/8/docs/api/

**Task 1 – Using a frame**
A top-level window in Java is called a Frame. `Swing` has a class called `JFrame` (which extends the `AWT Frame` class). We can create one by defining a `SimpleFrame` class, as included in the lab material. Run this as an example. Note that it imports all the `Swing` classes; it also extends `JFrame` and so inherits all of its functionality.

`SimpleFrame` sets the title and size of the window that is created. The `main` method creates a `SimpleFrame` object `frm`, and defines what should happen when the window is closed, and sets the window to be visible – you should find the tiny window appeared at the top left corner of the screen.

`JFrame` objects inherit many methods from the superclasses – take a look at the Java API for details. Experiment with the `setLocation` and `setBounds` methods (inherited from `java.awt.Window`) to reposition and resize your `SimpleFrame`.

Construct and show a frame that is 1/4 the size of the screen and has the same height/width ratio as the screen. Modify the code to position the frame in different corners of the screen. Use `SimpleFrame.java` as a starting point. You can use the following code to measure the screen height and width, and the `setBounds` method to set the corners of the `JFrame`.

```
import java.awt.Dimension;
import java.awt.Toolkit;
...
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
double width=screenSize.getWidth();
double height=screenSize.getHeight();
```

**Task 2 – Drawing a frame**
Frames are intended to be containers for other objects including text, graphics (images, lines, etc.), and GUI buttons. This is done using panels, which are also containers. To experiment with panels, use `SimpleFrame2.java` from the lab material.

This class creates a frame, with a title and size as before. However, the extra code creates a new `MyPanel` object (`MyPanel.java` is also included in the lab material), and adds it to the `Container`. Note that `Container` is part of the `AWT`.

Take some time to look at the `MyPanel.java` code. The constructor simply sets the Panel's background colour to white. To actually draw anything on the Panel we need to create a class that extends `JPanel`, and we need to provide a `paintComponent` method (overriding the default method in `JComponent`). We do not need to call this method explicitly because it is called automatically every time the window is drawn or redrawn. This method must call the `paintComponent` method of the superclass, and then implement any display actions associated with our `MyPanel` class.

The argument passed to `paintComponent` is a reference to an object of the `java.awt.Graphics` class. It provides methods that enable items to be drawn on our `JPanel`.

The `Graphics` class has a range of methods for drawing lines and text on a panel; for example

```
    drawRect(int x, int y, int width, int height)
```

draws a rectangle and

```
    drawString(String str, int x, int y)
```

draws a string. Experiment with these, and with the `setColor` method. To figure out how this works, look at the `setBackground` method in the `MyPanel` class and the API documentation for `Color` objects (note: US spelling).

**Task 3 – Using the `Java2D` library**

The `Java2D` library is a powerful set of graphics classes that allow much greater control over how graphics are displayed in windows. Drawing shapes using the `Java2D` library requires a `Graphics2D` object. These extend the abstract `Graphics` class, and so can be easily be obtained by casting the `Graphics` object we have already used in the `paintComponent` method:

```
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
    }
```

We can now implement `Java2D` operations on the object `g2`. Modify `paintComponent` method as follows:

```
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(10));
        g2.setPaint(new Color(128,0,0));
        Rectangle2D.Double s = new Rectangle2D.Double(20.0,20.0,100.0,50.0);
        g2.draw(s);
    }
```

This uses the `setStroke` method to set the line thickness in pixels, sets a drawing colour using `setPaint`, and then creates and draws a `Rectangle2D.Double` object. All shapes and lines in `Java2D` are drawn using floating point co-ordinates rather than integers. `Rectangle2D.Double` and `Rectangle2D.Float` are subclasses of the abstract superclass `Rectangle2D`. The `Java2D` classes for drawing shapes and lines implement the `Shape` interface, and you should look at the Java API to see what methods and attributes that specifies.

Experiment with `Line2D`, and `Ellipse2D` objects, changing the colour of the line using `g2.setPaint(Color)` and changing the fill using `g2.fill(s)`. Take a look at `Polygon's` API documentation and experiment with it.

Since `Rectangle2D`, `Ellipse2D`, and `Polygon` objects all implement the `Shape` interface, we can code an example of polymorphism using our knowledge of `Java2D`. Modify your `paintComponent` method to include the following code (or something like it):

```
Shape[] shapeList = new Shape[3];
shapeList[0] = new Rectangle2D.Double(20.0, 20.0, 40.0, 70.0);
shapeList[1] = new Ellipse2D.Double(20.0, 20.0, 40.0, 70.0);
int[] x = {10, 50, 70};
int[] y = {20, 70, 20};
shapeList[2] = new Polygon(x, y, 3);

for (int i = 0; i < 3; i++) {
    g2.draw(shapeList[i]);
}
```

Take a careful look at this code and make sure that you understand why we have been able to declare an array of Shape that points to several different types of object.

**Task 4 – Using buttons**

The next step is to add functionality to our windows. As with 2D graphics, most of the hard work is already done by the Java API, and all we have to do is to make use of existing classes and interfaces. Use `SimpleFrameWithQuitButton.java` from the lab material; it adds two more tasks to the examples we have looked at so far:

- adding a button to a `JPanel`;
- detecting when this button has been clicked and performing an appropriate action.

Take a look at the code for the class to make sure you understand how it works, and note the following points:

- This class extends `JFrame` and implements the `java.awt.event.ActionListener` interface;
- Implementing this interface will allow our program to detect if the button has been clicked;
- This interface requires us to implement the `ActionPerformed` method, which will be invoked when an event is detected.

This code creates a `JButton quitButton` labelled *'Quit'*. The button is then added to the panel, and the panel is added to the window's content pane. The other important part of this constructor is to link `quitButton` to the `ActionListener`.

A class that implements `ActionListener` is designated as an event listener. Events are transmitted from event sources (such as buttons) to event listeners. Any object may be designated as an event listener, (there can be more than one listener for a source) — this is called an event delegation model. A listener object implements a listener interface. Event sources have methods to register listener objects, and when an event occurs, an event object is sent from the event source to each registered listener. The listener objects react to the event using the information in the event object.

When the action event occurs (the button is pressed), the system creates a new instance of the `ActionEvent` class, which is passed to the `actionPerformed` method, which finds out what object has produced the event, and stores the reference to this object in the variable source. If this reference is the same as `quitButton` then the program halts.

**Task 5 – More buttons**

For the `SimpleFrameWithQuitButton` class, add more buttons, and take a different action when each button is pressed. For example, you could label an additional button `Button 1`, and display a message 'You have pressed Button 1'. You could also implement a method to create each button:

```
JPanel p = new JPanel();
quitButton = makeJButton("Quit", this);
p.add(quitButton);
firstButton = makeJButton("Button 1", this);
p.add(firstButton);
...
private JButton makeJButton(String s, ActionListener a) {
    JButton b = new JButton(s);
    b.addActionListener(a);
    return b;
}
```

**Task 6 – Adding text and image to a frame**

Work through the Java tutorial on using buttons, check boxes and radio Buttons — http://download.oracle.com/javase/tutorial/uiswing/components/button.html

Write a program code `FrameWithTextAndImage.java` that takes the string '*To be or not to be'* and display it in each of the 4 corners of a `JPanel`. Now modify your code so that the string is produced in a different font in each corner. Now use a different colour is used for each piece of text.

Create a frame that contains a GIF image file. The frame should be the same size as the GIF. The lab material contains an example file, `globe.gif`. (Hint: try using an `ImageIcon` to paint the GIF onto the frame. A `JLabel` can be used as a container for `ImageIcon`.)