Practical session 2, Week 3: Updating the Mouse Coordinates and Drawing the Grid

Before starting this practical you need to become completely confident that you understand all of the code in the version 1 of the drawing application that accompanies this practical sheet. The project is compressed in a file called *da1.zip*.

This version extends the solution to practical 1 with the implementation of all the GUI components in the control panel and `JTextArea` component in the message area.

It is particularly important that you understand why an **inner class** called `Canvas` (that extends/inherits from the `JPanel` class) is now used to create the `canvas` object.

Please ask a demonstrator to explain if you have any difficulties in understanding the code at hand before you start this practical.

In the current state of the drawing application, some of the GUI components are in place but they result produce no actions.

In this and the next few practical sessions you will be writing the code that will be executed when you use a GUI component (such as when pressing a button, selecting a checkbox etc.)

In this practical you will write code that responds to the mouse motion event in order to change/update a label (at the top of the control panel) with the mouse coordinates. In addition, you will also be implementing the grid control of the drawing application.

Task 1

Notice that in the **drawing position** panel (which is in the **drawing tools** panel) there is a label (an object created from the `JLabel` class). The object's name is `coordinatesLabel`.

You can change the label string using its `setText()` method. For example:

```
...
coordinatesLabel.setText("some text");
...
```

Try initialising the label with some text after it is created in the drawing application's constructor and run your program to confirm.

Task 2

Now, you need to keep changing the above label in order to display the coordinates of the mouse on the canvas. To achieve this you need to respond to the event that the mouse issues every time it moves on the canvas object. This event is called a **mouse motion event**.

In order to achieve this, you first need to create a class that **implements** the `mouseMotionListener` interface. Remember that implementing an interface is like making a "promise" that you will define some behaviour. For the `mouseMotionListener` interface this behaviour is two methods called `mouseMoved` and `mouseDragged`. Both methods take a `MouseEvent` object as a parameter. Let's call this class `CanvasMouseMotionListener`. It's (partial) implementation should look like this:

```
class CanvasMouseMotionListener implements MouseMotionListener
{
    public void mouseMoved(MouseEvent event)
    {
        ;
    }

    public void mouseDragged(MouseEvent event)
    {
        ;
    }
}
```

As you will see below, some code in this class will need to access at least one attribute of the `DrawingApplication` class. For this to be possible (according to variable scope rules) the code above must reside within the body of the `DrawingApplication` class. Defining a class within a class is perfectly possible (and indeed preferable in this case) in Java. The enclosed class is called an **inner class**.

When you assign an object of the `CanvasMouseMotionListener` class to a GUI component, such as the canvas panel for example, every time the mouse is moved on the canvas the `mouseMoved` method above will be called. Similarly every time the mouse is dragged (moved while one of its buttons is pressed) the `mouseDragged` method will be called.

So, where do you think you will need to add code in the class above in order to change the `coordinatesLabel` text to show the mouse coordinates?

Note that the `MouseEvent` object (called `event`) that is passed to the above methods has, among other, two methods called `getX()` and `getY()` that return the x- and y-coordinates (as integers) of the mouse location when the event occurred. Please visit the `MouseEvent` class' API to confirm this.

Once you complete the implementation of the `CanvasMouseMotionListener` class you need to create an instance from it and assign it to the canvas object. This is done by:

```
...
CanvasMouseMotionListener listenerObj = new CanvasMouseMotionListener();
canvas.addMouseMotionListener(listenerObj);
...
```

or, in one line by:

```
...
canvas.addMouseMotionListener(new CanvasMouseMotionListener());
...
```

Where do you thing either of the above code segments should be placed?

Task 3

This and the following tasks will guide you through the implementation of the grid controls in the control panel.

In a drawing application the grid is comprised of regularly spaced vertical and horizontal straight lines that serve as guides for aligning objects on the canvas.

Recall that the `JPanel` class implements a method called `paintComponent`. This method is automatically called whenever the contents of the panel object need to be redrawn, for example when the frame containing the panel first appears, when the panel is resized etc. The `paintComponent` method takes as a parameter an instance of the `Graphics` class. It is this `Graphics` object that you will be using to draw objects on the `canvas`.

It is a good idea at this point to visit the API of the `Graphics` class in order to experience the variety of methods with which it allows you do perform graphic operations.

Recall that the `Canvas` class is an inner class that extends the `JPanel` class. This extension is done in the drawing application for the sole purpose of extending the functionality of the `JPanel`'s `paintComponent` method.

Please verify at this point that the `draw` method is used to extend the function of the `JPanel`'s `paintComponent` method in the `Canvas` class.

Task 4

Let's try to use the `draw` method to draw lines on the canvas.

Recall that the `drawLine` method of the Graphics class is used to draw a line between two points. For example the command:

```
...
g.drawLine(20, 50, 100, 210);
...
```

(where "little" `g` is an instance of the Graphics class) will draw a line between the points (20, 50) and (100, 210).

Try this and verify that you can draw a line on the canvas by running your program.

You can change the drawing colour of the `Graphics` object by preceding any drawing command with the `setColor` method. For example the command:

```
...
g.setColor(new Color(1.0F, 0.0F, 0.0F));
...
```

will change the drawing colour to red. Recall from your poster practical in CSC-10024 that a colour can be specified by the amount (ranging from 0.0 to 1.0) of the red, green and blue components that make it. In the above example the resulting colour is pure red since there is 1.0 of red and 0.0 of green and blue.

Experiment by drawing lines of different colours on the canvas area.

Task 5

You are now expected to write code in order to fill the canvas on the screen with horizontal and vertical gridlines with a spacing of 10 pixels. This will be the fine grid.

It will be useful to know that the width and height of a `JPanel` object (and therefore a `Canvas` object) can be obtained using their `getWidth` and `getHeight` methods respectively. You will need these dimensions in order to draw gridlines that extent to the ends of the canvas.

Note that you are expected to use *efficient coding* to produce the grid lines. Do not write one line of code for every grid line!

Also, set the colour of the fine grid to (0.8F, 0.8F, 0.8F), i.e. a light shade of grey.

In a separate segment of code (after the above) draw another grid at 50 pixels spacing. This will be the coarse grid. Set the colour of this grid to a darker grey shade, say (0.6F, 0.6F, 0.6F).

Run your program intermittently to check your progress. The final result should look like the canvas area shown in figure 1, below.

Task 6

You now need to link the drawing of each grid (fine and coarse) to the corresponding checkboxes in the control panel.

In other words, every time the `draw` method is called it needs to check the state of each of the two grid control checkboxes and then draw, or not, the corresponding grid.

In order to interrogate the state of a `JCheckBox` object you need to use it's `isSelected` method. This method returns a `Boolean` value (`true` if the checkbox is checked and `false` otherwise). Here is an example:

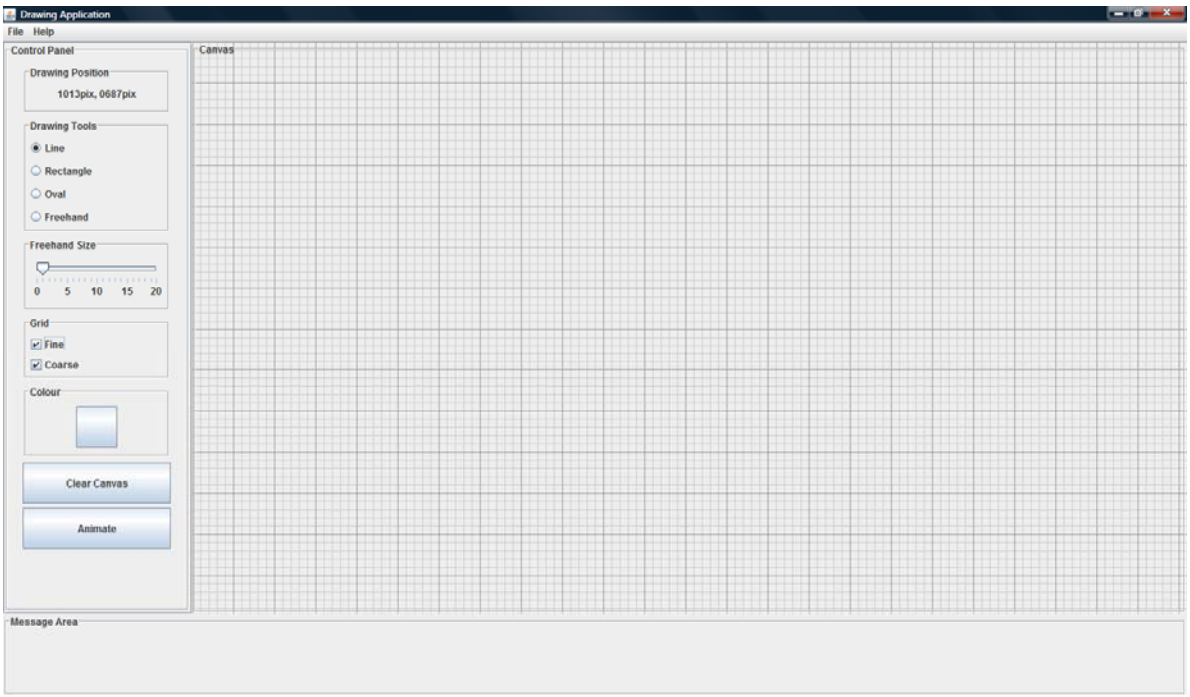

Figure 1

```
...
if(aCheckBox.isSelected()) {
    // Code to execute if the checkbox is checked.
} else {
    // Code to execute if the checkbox is not checked.
}
...
```

where `aCheckBox` is an instance of the `JCheckBox` class in the example above.

Use the above to control the grid with the checkboxes in the Grid panel but before you start read the task below...

Task 7

Recall that the `draw` method is called whenever the `paintComponent` method (of the `canvas` object) is used and the `paintComponent` method is called when there is a change in the size or visibility of the `canvas`. Redrawing the grid whenever the `canvas` is resized or is made to reappear (after being occluded for example) is desirable, but it is also expected that the grid is updated right when the state of the checkboxes is changed.

As things are, when you successfully finish the above task, regardless of when you change the state of the grid checkboxes, the grid will be updated only when an event occurs that causes the canvas to be repainted (i.e. when the `paintComponent` method is called). Verify this by running your code after finishing the above task.

So now you are looking for an **event listener** that listens to **change events**. You would expect that for this listener interface you have to implement a method that will be called each time there is a change to the **state** of the component the listener is associated with.

And you are very fortunate because such a listener does exist! The scenario is similar to that of the `MouseMotionListener` above.

Use the following information to cause the grid to be updated every time the state of either grid checkbox is changed:

- The event listener you need to implement here is the `ChangeListener`.
- There is only one method that you **must** implement for this interface. This is called `stateChanged` and it takes an instance of the `ChangeEvent` class (the event) as the only parameter.
- As mentioned several times above, the `paintComponent` method of a component is automatically called when the component is changed (resized, maximised etc.) however, you can also *request* a repaint whenever you want in your code by calling the `repaint` method of the component.
- Finally, use the `addChangeListener` method of a checkbox to assign an instance of the class implementing the `ChangeListener` to the checkbox.