Practical Session 1, Week 4: Freehand Drawing

This practical will take two sessions. You will continue the development of the drawing application by implementing:

- the freehand drawing tool (and its associated slider),
- the colour button in order to draw in different colours,
- the message area for posting messages to the user and
- the clear button for clearing the canvas.

Task 1

In the last practical you have practiced drawing lines on the canvas in order to implement the grid. Freehand drawing in the drawing application can be simply implemented by drawing small filled squares at each position along the path of the mouse while it is being **dragged**.

First, try to draw a single filled rectangle at an arbitrary location on the canvas. The `Graphics` class method that can do this for you is the `fillRect`. For example the following command:

```
...
g.fillRect(20, 30, 200, 100);
...
```

will draw a filled rectangle on the `Graphics` object `g` with its top left corner at point (20, 30), a width of 200 and a height of 100 pixels.

The rectangle's colour can be changed with a prior call to the `setColor` method of the `Graphics` class (see practical 2).

Practice drawing a few rectangles of different colours on canvas. Where would you place the code above so that the rectangles are drawn on the canvas as soon as you run your application?

Task 2

When the freehand drawing tool is selected and the mouse is being dragged we want to draw small squares (i.e. rectangles with equal width and height) for every position of the mouse.

However, before we tie the *mouse drag event* to this action, we need to think of how we will be saving the location, dimension and colour of each square as we draw it. Remember that once things are drawn on the canvas they are *forgotten*. It is our responsibility to *remember* and redraw them when needed.

For now, let us save the details of each square of the freehand trace in arrays. The problem with this is that arrays have limited size and therefore we need to pre-specify a maximum

number of freehand squares (i.e. array elements). You will find out later in this module that more advanced data structures (namely a *linked lists*) would be more appropriate for this purpose.

So, we need a number that specifies the maximum number of freehand squares:

```
...
private final int MAX_FREEHAND_PIXELS = 1000;
...
```

an array that can hold the colour of each square:

```
...
private Color[] freehandColour = new Color[MAX_FREEHAND_PIXELS];
...
```

and an array that can hold the position and size of each square:

```
...
private int[][] fxy = new int[MAX_FREEHAND_PIXELS][3];
...
```

`fxy` is a two-dimensional (2D) array. Think of it as a table that looks like:

	0	1	2
0	100	200	3
1	101	50	3
2	120	51	8
3	130	40	8
...	...	...	...
999	...	...	...

Each row of the array represents one freehand square. Columns 0 and 1 hold the x and y coordinates of the top-left corner of the square and column 2 contains the size of each side (width and height) of the square. In order to access a particular element in this array you need to specify the element's row and column. For example, for the shaded square in the example above:

```
...
fxy[2][1] = 30;
...
```

will replace the contents of the shaded element (i.e. number 51) with number 30.

As squares are drawn on the canvas, we also need to keep track of their count in order to properly index the above arrays, but most importantly to make sure that we don't try to access them beyond their limits:

```
...
private int freehandPixelsCount = 0;
```

```
...
```

Where are you going to place the above declarations? Ask a demonstrator to verify.

Task 3

Write code that will draw the squares that will be specified by the arrays above. Here is the pseudocode to help you:

For every freehand square (here also called pixel) that has been drawn (hint: use the count):
Set the drawing colour to the pixel's corresponding colour.
Draw the pixel.

First of all think: where should this code be placed?

As you draw with the mouse (to be implemented below in this practical) you will be accessing the arrays declared above. However for now, in order to test your code, initialise the above arrays with some data manually, for example in the constructor of the drawing application.

```
...
freehandColour[freehandPixelsCount] = new Color(1.0F, 0.0F, 0.0F);
fxy[freehandPixelsCount][0] = 100;  // x-coordinate
fxy[freehandPixelsCount][1] = 200;  // y-coordinate
fxy[freehandPixelsCount][2] = 10;  // dimension
freehandPixelsCount++;
...
```

If you want to have fun, why don't you put the above in a loop that randomises for colour, position and size values!

Task 4

It is now time to associate the mouse drag event with accessing the arrays defined above.

Recall from your previous practical that the `MouseMotionListener` interface requires the implementation of two methods. Namely, the `mouseMoved` and the `mouseDragged` method.

During the last practical you have written code in the `mouseMoved` method in order to update the mouse coordinates label as the mouse was being moved. Here we shall be writing code in the `mouseDragged` method. Here is a start:

```
...
class CanvasMouseMotionListener implements MouseMotionListener
{
    public void mouseMoved(MouseEvent event)
    {
        ...  // Implemented in the previous practical.
    }
```

```
    public void mouseDragged(MouseEvent event)
    {
        // Your code goes here...

        mouseMoved(event);
        canvas.repaint();
    }
}
...
```

Note that even though *dragging* the mouse also implies *moving* it (as we understand it), in Java one of two things can be happening at any given moment. Therefore, when you are dragging the mouse **only** the `mouseDragged` method is being executed and when you are moving the mouse **only** the `mouseMoved` method is being executed. It should therefore be apparent why we need to call the `mouseMoved` method from within the `mouseDragged` method. Surely, we need to keep track of the mouse's location even when it is being dragged...

If you have placed your freehand trace drawing code in the right place (task 3) then it should also be apparent why you need to call the `repaint` method of the `canvas` object.

In writing your code above, remember that before you start laying down squares on the canvas you need to check whether the freehand drawing tool (radio button) *is selected* and also that you have enough space remaining in order to store your freehand traces. Also, for now, give a default size of 5 pixels to all your freehand squares.

Task 5

You should now be able to drag your mouse and draw freehand traces but there is a small problem that you can realise when you try to draw a *single* freehand pixel by clicking (pressing and releasing) the mouse button at a certain location. The reason for this is that holding down the mouse button *without* moving it does not cause a mouse drag event.

To solve this problem we need to respond to the event associated with the pressing of the mouse button. The occurrence of this event (and other similar events) can be detected using a class that implements the `MouseListener` interface.

Here is the outline of the inner class you need to work on to solve the problem above.

```
...
class CanvasMouseListener implements MouseListener
{
    public void mousePressed(MouseEvent event) {
       ...
    }

    public void mouseReleased(MouseEvent event) {
        ...
    }

    public void mouseClicked(MouseEvent event) {
        ...
```

```
    }

    public void mouseEntered(MouseEvent event) {
        ...
    }

    public void mouseExited(MouseEvent event) {
        ...
    }
}
...
```

Please also remember to associate an instance of this class to the `canvas` object.

Task 6

Let us now use the freehand slider control to change the size of our freehand trace (i.e. the size of the squares drawn by the freehand tool).

To achieve this you need to make use of the `ChangeListener` interface in a similar manner as for the grid checkboxes in practical 2:

```
...
class FreehandSliderListener implements ChangeListener
{
    public void stateChanged(ChangeEvent event)
    {
        ...
    }
}
...
```

and then associate an instance of the class above with the `freehandSizeSlider` object.

By doing this, every time the user changes the slider component the `stateChanged` method above will be called with a `ChangeEvent` instance. It is important to realise here that (just like in the case with the grid checkboxes) the `event` parameter passed to the method does not contain any information about the state of the GUI component (the slider here). In order to find which value the slider is left at after the event you need to make use of the `getValue` method of the `JSlider` class:

```
...
freehandThickness = freehandSizeSlider.getValue();
...
```

Ask yourself now, where should the variable `freehandThickness` be declared and why?

A much quicker (but not so correct) way to make use of the slider to control the size of the freehand trace is to access the value of the slider object every time you need it instead of bothering with an event listener that will maintain a variable in synchrony with the state of the slider at all times.

Even if you choose to use this latter method in your program make sure that you understand why it is not the best?

Task 7

For this task you need to complete the implementation of the colour chooser button. Because the drawing colour will need to be accessible by quite a few methods in the drawing application it needs to be an *instance* variable:

```
...
private Color selectedColour = new Color(0.0F, 0.0F, 0.0F);
...
```

Every time you press the colour chooser button, you need to be able to change this variable to a new colour.

By now you should feel comfortable with the concept of events, event listeners and how to assign event listeners to different GUI components. Thus, from now on only a brief mention of the associated classes will be made for this mechanism. With little investigation you should be able to successfully assign actions to specific events caused by the user's interaction with GUI components.

In order to respond to events caused by pressing buttons (instances of the `JButton` class) you need to use the `ActionListener` interface. Such interface requires the implementation of a single method called `actionPerformed`.

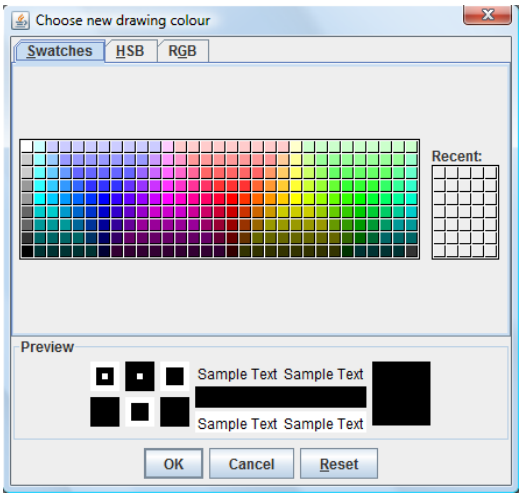
A simple method to request a new colour from the user is perhaps to ask the user to input three real numbers (one for each colour channel – red, green and blue) and instantiate a new colour instance by:

```
...
selectedColour = new Color(red, green, blue);
...
```

However, an even simpler and more elegant way to request a colour from the user is by using a colour chooser dialog window implemented by the `JColorChooser` class. Once you create an instance of this class, use the `showDialog` method to display the dialog:

```
...
JColorChooser colourChooser = new JColorChooser(selectedColour);
Color newColour = colourChooser.showDialog(null, "Choose new
drawing colour", selectedColour);
...
```

The colour chooser dialog looks like this:



Note that the dialog is dismissed by either pressing **OK** or **Cancel**. If **OK** is pressed the `showDialog` method will return an instance of the `Colour` class initialised to the selected colour in the dialog. If the **Cancel** button is pressed then the `showDialog` method will return `null`. Make use of this fact in order to change (or not) the application's drawing colour.

Finally, you need to change the background colour of the colour chooser button (`colourButton`) to show the new drawing colour. Remember that this has already been done once when the button was first created.

Task 8

Here you will implement the mechanism by which you can post messages to the user on the `JTextArea` object in message area. The `JTextArea` class implements a convenient method (called `append`) for appending a string to the existing text that the instance holds.

Write code in order to continuously inform the user (on the message area) of how much *ink* they have left (in terms of freehand squares) as they draw freehand on the canvas.

Task 9

Finally, there is very little to be said (and indeed to be done) in order to complete the implementation of the **Clear Canvas** button. Just as the colour chooser button, the **Clear Canvas** button is an instance of the `JButton` class and therefore can be assigned to the same kind of event listener that responds to the button being pressed.

What code will cause the canvas to be cleared from any freehand traces?

Also, make sure you clear any messages on the message area with this button.