Contents

0 Introduction **1**

1 Representing Curves **1**

2 Getting Started on the Application **3**

3 Deciding State in the Application **6**

4 Getting the Curves to Draw: Getting `Curve.java` ready **10**

5 Getting the Curves to draw: Enabling the Panel **12**

6 Creating and Enabling the Color Menus **15**

7 Consructing the Width Menu **18**

8 Graphics2D **21**

9 FileIO for Objects and Serialization **31**

0 Introduction

During this chapter we will create a program named `UniDraw`, which will be a simple drawing program with a color pen whose color and width are controlled by menus. The user will draw by dragging the mouse in the drawing panel, thereby creating a mark on the drawing.

We will be able to save our resulting creations for later viewing or editing. What is new here is that we will learn how to *serialize*, or pack away, objects into files and how to reconstitute them and restore the program's state at the time it was saved. This will be a full-featured event-driven graphical application.

1 Representing Curves

Let us begin by sketching in a bare-bones shell for our program. We will create two classes. We need a means with which to represent a curve in the drawing. To decide how to do this, let's be more specific about the application's functionality.

When the user presses the mouse, that triggers a `MouseEvent`; if the name of the event is `e`, we get the point at which it occurred by calling `e.getPoint()`. So, when the mouse is pressed, we should create a new curve and it should begin at the point we retrieved.

Now the user drags the mouse to draw. As this occurs, the mouse is *polled* and it periodically fires off a `MouseEvent`. Each event can tell us where it occurred; we will need to store all of those points in our curve.

Finally, when he mouse is released, the drag is over. We add the point where it is released, and then we wait for the next pressing of the mouse. Notice that every curve is *guaranteed* to have at least two points.

What does a curve need to know? It needs to know the list of points it accrues during the press/drag/release sequence. It also needs to know, according to our specification, its width and its color.

There is a vary natural way to achieve this. Let us make our curve class extend `ArrayList<Point>`. This class has an `add` method that will easily add points to the class. We will also make the curve capable of drawing itself in a graphical window. So we begin like so. Observe that a curve's color and width will never change, so we make these state variables `final`.

```
import java.awt.Color;
import java.awt.Point;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
}
```

Next we add a constructor to intilaize the state variables.

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.util.ArrayList;
public class Curve extends ArrayList<Point> //implements Serializable
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color = color;
```

```
        this.width = width;
    }
}
```

Finally, we will place a `draw` method in our `Curve` class.

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.util.ArrayList;

public class Curve extends ArrayList<Point> //implements Serializable
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;
    }
    public void draw(Graphics g)
    {
    }
}
```

We now have a starting point for the `Curve` class. Place this code in a file named `Curve.java` and compile it. The remarks we made in this session about the mouse will help us to write the mouse event handlers needed to make the application work.

2 Getting Started on the Application

We will begin with a standard shell for a graphical application.

```
import javax.swing.JFrame;
public class DrawFrame extends JFrame implements Runnable
{
    public DrawFrame()
    {
        super("UniDraw:  Untitled");
    }
    public void run()
    {
```

```
        setSize(600,600);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        DrawFrame df = new DrawFrame();
        javax.swing.SwingUtilities.invokeLater(df);
    }
```

Place this code in a file named `DrawFrame.java`. Compile it and then run it. You should see a window appear on the screen with a title in the title bar.

Now it is time to think about what we want. We need a space for drawing. To do this, we need to subclass `JPanel` and then override `public void paintComponent(Graphics g)` to tell the drawing area how to render itself.

The drawing area must respond to the pressing and releasing of the mouse, so it must be a `MouseListener`. It must respond to the dragging of the mouse, so it must be a `MouseMotionListener`.

Communication will flow in ths way. The user will have selected a background color, a pen color and a width using the menus; these will all be given default values to start. These values need to be known by the application and will be determined by menu choices. The panel will need to have access to these values, so it is a good idea to make the panel an inner class. So our panel will have to extend `JPanel` and implement `MouseListener` and `MouseMotionListener`. We will need to implement the methods in the two interfaces so our class will compile. Now our code will look like this. We will add in the menu items, too, and make the `JMenuBar` a state variable so that we can separate the making of the various menus into separate functions. This will make managing our code simpler. We placed a `paintComponent` method in the draw panel so it would be easy to see that is got placed in the content pane.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.Color;
import java.awt.Graphics;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
```

```
private JMenuBar mbar;
public DrawFrame()
{
    super("UniDraw:  Untitled");
    dp = new DrawPanel();
    mbar = new JMenuBar();
    setJMenuBar(mbar);
}

public void run()
{
    setSize(600,600);
    makeFileMenu();
    makeColorMenu();
    makeBackgroundMenu();
    makeWidthMenu();
    getContentPane().add(dp);
    setVisible(true);

}
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
}
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Pen Color");
    mbar.add(colorMenu);
}
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
}
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
    mbar.add(widthMenu);
}
public static void main(String[] args)
{
    DrawFrame df = new DrawFrame();
    javax.swing.SwingUtilities.invokeLater(df);
}
class DrawPanel extends JPanel
```

```
        implements MouseListener, MouseMotionListener
    {
        public DrawPanel()
        {
            addMouseListener(this);
            addMouseMotionListener(this);
        }

        public void mouseEntered(MouseEvent e){}
        public void mouseExited(MouseEvent e){}
        public void mousePressed(MouseEvent e){}
        public void mouseReleased(MouseEvent e){}
        public void mouseClicked(MouseEvent e){}
        public void mouseMoved(MouseEvent e){}
        public void mouseDragged(MouseEvent e){}
        @Override
        public void paintComponent(Graphics g)
        {
            g.setColor(new Color(0xabcdef));
            g.fillRect(0,0,getWidth(), getHeight());
        }
    }
}
```

If you run this you should see the menus in the menu bar and a Carolina blue square in the content pane.

3 Deciding State in the Application

If we think about the user experience, figuring out state should be fairly easy. Lurking in the future is the issue of what we need to store in a file to reconstitute our drawing session. This informatoin needs to be included in the state variables.

We will begin by focusing on the mechanism of drawing in the draw panel. To draw the panel it seems we need to know the color for the background and the color the pen is now coloring. Hence we create these variables.

```
private Color background;//color of background
private Color foreground;//color of current curve
```

We also need to know the width of the pen. We create that variable too.

```
private int width;
```

Finally, we must deal with the drawing. We will record it as an array list of curves. We declare it as follows.

```
private ArrayList<Curve> drawing;
```

Once we create these variables, they must be initialized in the constructor.

```
public DrawFrame()
{
    super("UniDraw:  New File");
    dp = new DrawPanel();
    mbar = new JMenuBar();
    setJMenuBar(mbar);
    background = new Color(0xabcdef);
    foreground = new Color(0x228800);
    width = 1;
    drawing = new ArrayList<Curve>();
}
```

When we do this, let us update the `paintCompoent` method in the `DrawPanel` as follows. We want the chosen background color to color the background, not just `0xabcdef`.

```
@Override
public void paintComponent(Graphics g)
{
    g.setColor(background);
    g.fillRect(0,0,getWidth(), getHeight());
}
```

Drawing the drawing is simple. Since the `Curve` class has a `draw` method, we just tell each curve to draw itself using a collections `for` loop like so.

```
@Override
public void paintComponent(Graphics g)
{
    g.setColor(background);
    g.fillRect(0,0,getWidth(), getHeight());
    for(Curve c:drawing)
    {
        c.draw(g);
    }
}
```

Finally add this to your imports

```
import java.util.ArrayList;
```

and your progam will compile and run. It, however will not render curves in the drawing because we have not told the curves how to draw themselves. Our call to `s.draw(g)` does nothing. Here is the current state of our `DrawFrame` class.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
    private JMenuBar mbar;
    private Color background;//color of background
    private Color foreground;//color of current curve
    private int width;
    private ArrayList<Curve> drawing;

    public DrawFrame()
    {
        super("UniDraw:  New File");
        dp = new DrawPanel();
        mbar = new JMenuBar();
        setJMenuBar(mbar);
         setJMenuBar(mbar);
        background = new Color(0xabcdef);
        foreground = new Color(0x001a57);
        width = 1;
        drawing = new ArrayList<Curve>();
    }

    public void run()
    {
        setSize(600,600);
        makeFileMenu();
        makeColorMenu();
```

```
    makeBackgroundMenu();
    makeWidthMenu();
    getContentPane().add(dp);
    setVisible(true);

}
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
}
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Pen Color");
    mbar.add(colorMenu);
}
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
}
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
    mbar.add(widthMenu);
}
public static void main(String[] args)
{
    DrawFrame df = new DrawFrame();
    javax.swing.SwingUtilities.invokeLater(df);
}
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
```

```
        public void mouseDragged(MouseEvent e){}
        @Override
        public void paintComponent(Graphics g)
        {
            g.setColor(new Color(0xabcdef));
            g.fillRect(0,0,getWidth(), getHeight());
            for(Curve c: drawing)
            {
                c.draw(g);
            }
        }
    }
}
```

4 Getting the Curves to Draw: Getting `Curve.java` ready

Open the file `Curve.java`; here is its current state

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;

    }
    public void draw(Graphics g)
    {
    }
}
```

Notice that the draw method is unimplemented. Here is what we propose to do. For now we will sidestep the question of controlling width. The default width is 1; we shall begin by getting curves to draw on the screen.

1. Set the pen color to the color of the curve.
2. Connect each dot in the curve to its successor.

Our curve is an array list. This means it can learn its size by calling `size()`. We gain access to the points via the method `Point get(int index)`. Let us declare the following.

```
int n = size();
```

Now supppose `k` iterates through the list. We need to connect `get(k)` to `get(k+1)`. How do we do this? You must ask who is doing it. The pen has that job an it is an object of type `Graphics`, so go to the `Graphics` API page. Here is the method detail for `lineTo`

drawLine

```
public abstract void drawLine(int x1,
                              int y1,
                              int x2,
                              int y2)
```

Draws a line, using the current color, between the points `(x1, y1)` and `(x2, y2)` in this graphics context's coordinate system.

Parameters:

`x1` - the first point's x coordinate.
`y1` - the first point's y coordinate.
`x2` - the second point's x coordinate.
`y2` - the second point's y coordinate.

Now we begin to code our loop to draw the cure. We change the color of the pen to our color and then we connect each dot to its successor. Note the use of the `n -1`. Failure to do that will cause the endpoint to connect to a nonexistent sucessor. The result of that mistak would be an ugly `IndexOutOfBoundsException` being thrown.

```
public void draw(Graphics g)
{
    int n = size();
    g.setColor(color);
    for(int k = 0; k < n - 1; k++)
    {
```

```
        g.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
    }
}
```

Also, notice that there is no provision for connecting points so we had to break out each coördinate separately. Compile and run now. To finish, we must now make the panel's mouse methods work. The action of the mouse is not yet populating our curves or our drawings.

5 Getting the Curves to draw: Enabling the Panel

This is the current state of our `DrawPanel`.

```
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
    public void mouseDragged(MouseEvent e){}
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(new Color(0xabcdef));
        g.fillRect(0,0,getWidth(), getHeight());
        for(Curve c: drawing)
        {
            c.draw(g);
        }
    }
}
```

Let's step through what we had said before about the mouse and carefully outline what is to happen. Note that we will ignore most of the mouse methods.

1. When the user presses the mouse, create a new curve and add the point to it where the press occurred. Also, add the curve to the drawing.
2. When the mouse is dragged, accumulate the points where the mouse events are being fired during the polling process. As each mouse event is fired, repaint the panel so it stays up to date.
3. When the mouse is released, add the point of release to the curve, and repaint.

This means we will use the methods `mousePressed`, `mouseReleased` and `mouseDragged`. The rest get ignored. Let us begin with `mousePressed`. We must make a new curve and then add the point of the press to it. Remember, a mouse event has a `getPoint()` method that does the job.

```
public void mousePressed(MouseEvent e)
{
    Curve c = new Curve(color, width);
    c.add(e.getPoint());
    drawing.add(c);
}
```

When the mouse is released, we get the point, add it to the curve, and repaint.

```
public void mouseReleased(MouseEvent e)
{
    c.add(e.getPoint());
    repaint();
}
```

When the mouse is dragged, we get the point and repaint.

```
public void mouseDragged(MouseEvent e)
{
    c.add(e.getPoint());
    repaint();
}
```

Now compile. You will get this.

```
$ javac *.java
DrawFrame.java:87: error: cannot find symbol
            Curve c = new Curve(color, width);
                                ^
  symbol:   variable color
  location: class DrawFrame.DrawPanel
```

```
DrawFrame.java:92: error: cannot find symbol
            c.add(e.getPoint());
            ^
  symbol:   variable c
  location: class DrawFrame.DrawPanel
DrawFrame.java:97: error: cannot find symbol
            c.add(e.getPoint());
            ^
  symbol:   variable c
  location: class DrawFrame.DrawPanel
3 errors
$
```

That is ugly indeed. What happened? The variable `c` we created to point at the curve is local to `mousePressed`. Lift it up and make it a state variable and all will work. Now your class will look like this.

```
    class DrawPanel extends JPanel
        implements MouseListener, MouseMotionListener
    {
        Curve c;            // c is now a state varible
        public DrawPanel()
        {
            addMouseListener(this);
            addMouseMotionListener(this);
            c = null;  //initialize
        }
        //ignored mouse events
        public void mouseEntered(MouseEvent e){}
        public void mouseExited(MouseEvent e){}
        public void mouseClicked(MouseEvent e){}
        public void mouseMoved(MouseEvent e){}
        //active mouse events
        public void mousePressed(MouseEvent e)
        {
            //remove "Curve" on this first line.
            c = new Curve(foreground, width);
            c.add(e.getPoint());
            drawing.add(c);
        }
        public void mouseReleased(MouseEvent e)
        {
            c.add(e.getPoint());
            repaint();
        }
        public void mouseDragged(MouseEvent e)
```

```
    {
        c.add(e.getPoint());
        repaint();
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(background);
        g.fillRect(0,0,getWidth(), getHeight());
        for(Curve c: drawing)
        {
            c.draw(g);
        }
    }
}
```

6 Creating and Enabling the Color Menus

Next we need to get the background and foreground color menus working. We shall give each menu the Roy G. Biv colors; you will see that other colors caan be added easily; each new color can be added with a single line of code! We will also meet the `JColorChooser` widget, which will allow the user to pick a custom 24-bit color. It has the addional feature of allowing the user to choose *alpha*, or opacity of colors.

Alpha is determined by a numerical scale of 0 through `0xff`. The alpha value of 0 gives a perfectly transparent (non-)color. The alpha value of `0xff` gives a totally opaque color. Intermediate values yield transparaent layers of color. All colors are completely determined by a 32 bit integer, one byte for red, another for green, another for blue, and one for alpha.

Now we resume our main thread. We will create innner classes for the pen color menu items and the background color menu items. We begin with the pen color menu, making each menu item know its color.

```
public class PenColorItem extends JMenuItem
{
    private final Color color;
    public PenColorItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
    }

}
```

We will also attach an action listener which will set the pen color to the menu item's color. Note that the changing of the pen color does not require an update of the `DrawPanel`, so we do not have to repaint the panel.

```
    public class PenColorItem extends JMenuItem
    {
        private final Color color;
        public PenColorItem(Color _color, String colorName)
        {
            super(colorName);
            color = _color;
            addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    foreground = color;
                }
            });
        }
    }
```

Now add these two lines and compile

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

Now let's build the pen color menu. We go into the pen color menu and populate it thusly with a red menu item. We will test this before proceeding with the rest.

```
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new PenColorItem(Color.red, "red"));
}
```

Drag the mouse in the window and see the red color drawn. We will now complete Roy G. Biv; note the hex codes used for indigo and violet.

```
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new PenColorItem(Color.red, "red"));
```

```
    colorMenu.add(new PenColorItem(Color.orange, "orange"));
    colorMenu.add(new PenColorItem(Color.yellow, "yellow"));
    colorMenu.add(new PenColorItem(Color.green, "green"));
    colorMenu.add(new PenColorItem(Color.blue, "blue"));
    colorMenu.add(new PenColorItem(new Color(0x2E0854), "indigo"));
    colorMenu.add(new PenColorItem(new Color(0x7D26CD), "violet"));
}
```

We still have 16,777,209 colors to go. How do we get them? We use a `JColorChooser` to choose a custom color. We will avail ourselves of its static `showDialog` method to do the job. The constructor needs three arguments, a `JComponent`, a message (a string will do), and a default color. We will use our draw frame as the component, `"Choose Pen Color"` as our message, and the `foreground` color as our default color. By default, the color does not change. This is in keeping with minimizing user surprise.

Since there is some *ad hoc* procedure here, we will attach an anonymous inner class as a listener. The `JColorChooser` is goof-proof and requires no exception handling. Begin with an import.

```
import javax.swing.JColorChooser;
```

Now add the listener. at the end of the menu.

```
JMenuItem custom = new JMenuItem("Custom...");
colorMenu.add(custom);
custom.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        foreground = JColorChooser.showDialog(DrawFrame.this,
               "Choose Background Color", background);
    }
});
```

Select the new item from the menu and watch the (very cool) color chooser dialog pop up and offer you all of those choices.

Now wwe show the code for the background color. It is very similar, except that we must repaint when we change its color.

```
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
    backgroundMenu.add(new bgMenuItem(Color.red, "red"));
    backgroundMenu.add(new bgMenuItem(Color.orange, "orange"));
```

```
    backgroundMenu.add(new bgMenuItem(Color.yellow, "yellow"));
    backgroundMenu.add(new bgMenuItem(Color.green, "green"));
    backgroundMenu.add(new bgMenuItem(Color.blue, "blue"));
    backgroundMenu.add(new bgMenuItem(new Color(0x2E0854), "indigo"));
    backgroundMenu.add(new bgMenuItem(new Color(0x7D26CD), "violet"));
    JMenuItem custom = new JMenuItem("Custom...");
    backgroundMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            background = JColorChooser.showDialog(DrawFrame.this,
                   "Choose Background Color", background);
            repaint();
        }
    });
}
```

Now we add the needed inner class to drive this.

```
class BackgroundMenuItem extends JMenuItem
 {
     final Color color;
     public BackgroundMenuItem(Color _color, String colorName)
     {
         super(colorName);
         this.color = _color;
         addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent e)
             {
                 background = color;
                 dp.repaint();
             }
         });
     }
 }
```

7 Consructing the Width Menu

We will construct the width menu and get it to change the state variable `width`, but we will also have to look at how curves draw themselves to get proper performance out of this menu. As of now, our pen only draws a path one pixel wide.

The inner class needed has one small surprise.

```
class WidthMenuItem extends JMenuItem
{
    final int width;
    public WidthMenuItem(int _width)
    {
        super("" + _width);
        width = _width;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                DrawFrame.this.width = width;
            }
        });
    }
}
```

Note the use of `DrawFrame.this` to specify that we want the ennclosing class's `width` variable to have a new value assigned to it.

We then turn to constructing the menu. Begin by doing this import

```
import javax.swing.JOptionPane;
```

Now modify `makeWidthMenu()` as follows.

```
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
    mbar.add(widthMenu);
    widthMenu.add(new WidthMenuItem(1));
    widthMenu.add(new WidthMenuItem(2));
    widthMenu.add(new WidthMenuItem(5));
    widthMenu.add(new WidthMenuItem(10));
    widthMenu.add(new WidthMenuItem(20));
    widthMenu.add(new WidthMenuItem(50));
    JMenuItem custom = new JMenuItem("custom...");
    widthMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            width = JOptionPane.showInputDialog(DrawFrame.this,
            }

        }
    });
}
```

Extreme Danger! What happens if the user types in something that is not a valid integer. Let's see what happens when the user types in `"cows"`. We get this horrid thing.

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException:
    For input string: "cows"
    at java.lang.NumberFormatException.forInputString
          (NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at DrawFrame$3.actionPerformed(DrawFrame.java:117)
               .
               .
          (much horrid stuff)
               .
               .
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:139)
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:97)
```

It is not acceptable for a user goof to produce this. We see that an unhandled `NumberFormatException` has bubbled up and caused havoc. We remedy this by placing an appropriate catch-try sequence in the listener as follows.

```
custom.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        String  buf = JOptionPane.showInputDialog(DrawFrame.this,
           "Specify width");
        try
        {
            width = Integer.parseInt(buf);
        }
        catch(NumberFormatException ex)
        {
            JOptionPane.showMessageDialog(DrawFrame.this,
               "String \"" + buf + "\" is not an integer.");
        }

    }
});
```

No action will be taken in the draw panel, but the user will be alerted to his mistake. The menu does what it is supposed to, but we still have no control of the pen's actual width. For this, we must learn about a new and helpful class.

8 Graphics2D

If you scour the page for `java.awt.Graphics`, you will find no facility for controlling the pen's width. So, when we get desperate what do we do? We look in the ancestor classes. These are barren of anything useful. Where now? We look in some child classes. The `Graphics` class has child class `Graphics 2D`, which will help solve our problem.

If you scroll down on the API page, you will see a section entitled "Default Rendering Attributes. It says this.

`Paint`
: The color of the Component.

`Font`
: The Font of the Component.

`Stroke`
: A square pen with a linewidth of 1, no dashing, miter segment joins and square end caps.

`Transform`
: The getDefaultTransform for the GraphicsConfiguration of the Component.

`Composite`
: The AlphaComposite.SRC_OVER rule.

`Clip`
: No rendering Clip, the output is clipped to the Component.

What is important to us is that the stroke of the pen has a line width of 1. It also says there are "miter segment joins" and "square end caps." We also see that there is a `setStroke` method. Here is its method detail.

`public abstract void setStroke(Stroke s)`
Sets the Stroke for the Graphics2D context.

Parameters:
- the Stroke object to be used to stroke a Shape during the rendering process

See Also:
`BasicStroke, getStroke()`

We see that the classes `Stroke` and `BasicStroke` might also be of importance. We learn that `Stroke` is an interface, and that `BasicStroke` is a class that implements it. We can create a `BasicStroke` and use the `setStroke` method for the `Graphics2D` pen to obtain a pen whose with is larger than 1.

Inside of a `Graphics` lurks a `Graphics2D` You are guaranteed to be able to make the cast

```
(Graphics2D) g
```

if `g` is a `Graphics` object. You always have access to a `Graphics2D` if you are rendering inside of a `Swing` object. This is the type of object the `Swing` paint engine uses.

Now go to the `BasicStroke` constructor list. There is a constructor that will accept a `float`, and therefore which accepts our `width`. We will now use this to upgrade the `draw` method of the `Curve` class. We begin by inserting the needed cast.

```
public void draw(Graphics g)
 {
     Graphics2D g2d = (Graphics2D) g;
     g2d.setColor(color);
     int n = size();
     for(int k = 0; k < n - 1; k++)
     {
         g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
     }
 }
```

Compile and run. You will see that nothing has changed. We will now call `g2d`'s `setStroke` method and feed it a `BasicStroke` with `width`.

```
public void draw(Graphics g)
 {
     Graphics2D g2d = (Graphics2D) g;
     g2d.setStroke(new BasicStroke(width));
     g2d.setColor(color);
     int n = size();
     for(int k = 0; k < n - 1; k++)
     {
         g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
     }
 }
```

Now let's draw in the window. Choose a width of 20. The result looks like this.

The fat curve looks as if it were painted by watery, spattery paint. You can see where we drew a dot; it shows a square pen tip. We are not happy with this result and need to improve it.

How do we improve its appearance? This is where we need to look at the static constants in the `BasicStroke` class.

`BasicStroke.CAP_BUTT`	This ends a path with a semicircle.
`BasicStroke.CAP_ROUND`	This ends a path with a semicircle.
`BasicStroke.CAP_SQUARE`	This ends a path with a half a square.
`BasicStroke.JOIN_SQUARE`	This joins path segments by connecting the outer corners of their wide outlines wiht a straight segment.
`BasicStroke.JOIN_MITER`	This joins path segments by extending their outside edges until they meet.
`BasicStroke.JOIN_ROUND`	This joins path segments by rounding off the corner at a radius half of line width.

Use them in the constructor `BasicStroke(float width, int cap, int join)`.
You will select one `CAP` constant and one `JOIN` constant. We will use `BasicStroke.CAP_ROUND`

and `BasicStroke.JOIN_ROUND`. Here is the new appearance for our `draw` method in the `Curve` class.

```
public void draw(Graphics g)
{
    Graphics2D g2d = (Graphics2D) g;
    g2d.setColor(color);
    int n = size();
    g2d.setStroke(new BasicStroke(width, BasicStroke.CAP_ROUND,
         BasicStroke.JOIN_ROUND));
    for(int k = 0; k < n - 1; k++)
    {
        g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
    }
}
```

Now run again and compile. Notice the spiffy appearance of the fat curve. Also notice that clicking in a spot yields a circular blob of Dook-blue ink. The use of the cap and join parameters cleans things up and gets rid of the splatter.

Let us now collect the current state of the files. First, `Curve.java`.

```
import java.awt.Color;
import java.awt.Point;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.BasicStroke;
import java.util.ArrayList;

public class Curve extends ArrayList<Point>
{
    final Color color;
    final int width;
    public Curve(Color color, int width)
    {
        super();
        this.color= color;
        this.width = width;

    }
    public void draw(Graphics g)
    {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setColor(color);
        int n = size();
        g2d.setStroke(new BasicStroke(width, BasicStroke.CAP_ROUND,
             BasicStroke.JOIN_ROUND));
        for(int k = 0; k < n - 1; k++)
        {
            g2d.drawLine(get(k).x, get(k).y, get(k+1).x, get(k+1).y);
        }
    }
}
```

Next, `DrawFrame.java`.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JColorChooser;
import javax.swing.JOptionPane;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

```
import java.awt.event.MouseMotionListener;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class DrawFrame extends JFrame implements Runnable
{
    private DrawPanel dp;
    private JMenuBar mbar;
    private Color background;//color of background
    private Color foreground;//color of current curve
    private int width;
    private ArrayList<Curve> drawing;

    public DrawFrame()
    {
        super("UniDraw:  New File");
        dp = new DrawPanel();
        mbar = new JMenuBar();
        setJMenuBar(mbar);
         setJMenuBar(mbar);
        background = new Color(0xabcdef);
        foreground = new Color(0x001a57);
        width = 1;
        drawing = new ArrayList<Curve>();
    }

    public void run()
    {
        setSize(600,600);
        makeFileMenu();
        makeColorMenu();
        makeBackgroundMenu();
        makeWidthMenu();
        getContentPane().add(dp);
        setVisible(true);

    }
    public void makeFileMenu()
    {
        JMenu fileMenu = new JMenu("File");
        mbar.add(fileMenu);
    }
```

```
public void makeColorMenu()
{
    JMenu colorMenu = new JMenu("Color");
    mbar.add(colorMenu);
    colorMenu.add(new PenColorItem(Color.red, "red"));
    colorMenu.add(new PenColorItem(Color.orange, "orange"));
    colorMenu.add(new PenColorItem(Color.yellow, "yellow"));
    colorMenu.add(new PenColorItem(Color.green, "green"));
    colorMenu.add(new PenColorItem(Color.blue, "blue"));
    colorMenu.add(new PenColorItem(new Color(0x2E0854), "indigo"));
    colorMenu.add(new PenColorItem(new Color(0x7D26CD), "violet"));
    JMenuItem custom = new JMenuItem("Custom...");
    colorMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            foreground = JColorChooser.showDialog(DrawFrame.this,
                   "Choose Background Color", foreground);
        }
    });
}
public void makeBackgroundMenu()
{
    JMenu backgroundMenu = new JMenu("Background");
    mbar.add(backgroundMenu);
    backgroundMenu.add(new BackgroundMenuItem(Color.red, "red"));
    backgroundMenu.add(new BackgroundMenuItem(Color.orange, "orange"));
    backgroundMenu.add(new BackgroundMenuItem(Color.yellow, "yellow"));
    backgroundMenu.add(new BackgroundMenuItem(Color.green, "green"));
    backgroundMenu.add(new BackgroundMenuItem(Color.blue, "blue"));
    backgroundMenu.add(new BackgroundMenuItem(new Color(0x2E0854), "indigo"));
    backgroundMenu.add(new BackgroundMenuItem(new Color(0x7D26CD), "violet"));
    JMenuItem custom = new JMenuItem("Custom...");
    backgroundMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            background = JColorChooser.showDialog(DrawFrame.this,
                   "Choose Background Color", background);
            repaint();
        }
    });
}
public void makeWidthMenu()
{
    JMenu widthMenu = new JMenu("Width");
```

```
    mbar.add(widthMenu);
    widthMenu.add(new WidthMenuItem(1));
    widthMenu.add(new WidthMenuItem(2));
    widthMenu.add(new WidthMenuItem(5));
    widthMenu.add(new WidthMenuItem(10));
    widthMenu.add(new WidthMenuItem(20));
    widthMenu.add(new WidthMenuItem(50));
    JMenuItem custom = new JMenuItem("custom...");
    widthMenu.add(custom);
    custom.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            String  buf = JOptionPane.showInputDialog(DrawFrame.this,
               "Specify width");
            try
            {
                width = Integer.parseInt(buf);
            }
            catch(NumberFormatException ex)
            {
                JOptionPane.showMessageDialog(DrawFrame.this,
                   "String \"" + buf + "\" is not an integer.");
            }

        }
    });

}
public static void main(String[] args)
{
    DrawFrame df = new DrawFrame();
    javax.swing.SwingUtilities.invokeLater(df);
}
/*****************************************************************
*
*            Width Menu Items
*
*****************************************************************/
class WidthMenuItem extends JMenuItem
{
    final int width;
    public WidthMenuItem(int _width)
    {
        super("" + _width);
        width = _width;
        addActionListener(new ActionListener(){
```

```
            public void actionPerformed(ActionEvent e)
            {
                DrawFrame.this.width = width;
            }
        });
    }
}

/*****************************************************************
*
*            Color Menu Items
*
*****************************************************************/
public class PenColorItem extends JMenuItem
{
    private final Color color;
    public PenColorItem(Color _color, String colorName)
    {
        super(colorName);
        color = _color;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                foreground = color;
            }
        });
    }

}
class BackgroundMenuItem extends JMenuItem
{
    final Color color;
    public BackgroundMenuItem(Color _color, String colorName)
    {
        super(colorName);
        this.color = _color;
        addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                background = color;
                dp.repaint();
            }
        });
    }
}
/*****************************************************************
```

```
*
*           The Draw Panel
*
**********************************************************************/
class DrawPanel extends JPanel
    implements MouseListener, MouseMotionListener
{
    Curve c;            //now a state varible
    public DrawPanel()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        c = null;  //initialize
    }
    //ignored mouse events
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
    //active mouse events
    public void mousePressed(MouseEvent e)
    {
        //remove "Curve" on this first line.
        c = new Curve(foreground, width);
        c.add(e.getPoint());
        drawing.add(c);
    }
    public void mouseReleased(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    public void mouseDragged(MouseEvent e)
    {
        c.add(e.getPoint());
        repaint();
    }
    @Override
    public void paintComponent(Graphics g)
    {
        g.setColor(background);
        g.fillRect(0,0,getWidth(), getHeight());
        for(Curve c: drawing)
        {
            c.draw(g);
        }
```

```
        }
    }
}
```

Programming Exercises

1. Experiment with the other cap and join costants to observe the effects they cause on your drawing.

9 FileIO for Objects and Serialization

Now we well learn how to save our drawings in a file. This will involve several new classes, as well as the familar `File` class, which represents locations in our file system. We will now place two items in our `File` menu, `open` and `save`. Choosing the `open` menu item will open a drawing file and display it to the screen. Note that there are a lot of parallels to the `NitPad` case study here.

Let us begin by adding some features to our code. First of all, we will add a state variable to the `DrawFrame` class.

```
private File currentFile;
```

You also will need this import

```
import java.io.File;
```

In the constructor, do this

```
currentFile = new File("test.unid"); //TODO set this to null when menus work
```

We will begin by getting objects into and out of this fixed file `test.unid`. Later, we will get the full `File` menu working we will choose the current file from the file system using the `JFileChooser` widget.

FileIO will be controlled by listeners attached to items in the `File` menu. Let us begin with the process of saving a drawing.

This is done by creating an `ObjectOutputStream`. You can do this as follows.

```
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream(currentFile);
```

This process, if you check the constructor, can throw an `IOException`. Remember, we will adhere to the convention that we should handle any `FileNotFoundException` in our exception handling code.

You will need the following imports.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
```

What can be stored via an `ObjectOutputStream`? You can store any primitive type, or any object type that implements the interface `java.io.Serializable`. This makes sense, since the process of storing an object in a file in this manner is called *serialization*. The process reduces the object to its bytes and packs it in a non-human-readable binary format.Go to the API page and look up this interface. You will see that this is a "bundler interface" that has no required methods. If you attempt to serialize an object that does not implement this interface, you will be the unhappy recipient of a `NonserializableExcepton`. This is not a runtime exception, but it is a subclass of `IOException`. We can ferret these out by using the `printStackTrace()` method for `IOException`s.

Here we see the current state of `makeFileMenu`.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
}
```

Now let us add the save menu item and attach an action listener shell to it. You can put statemnt to print some string to `stdout` to test that it works. Run that, test it, and then discard it.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem saveItem = new JMenuItem("Save");
    fileMenu.add(saveItem);
    saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            saveWindowToCurrentFile();
        }
    });
}
```

Recalling what we know from `NitPad`, we know that there will be a need to write a file from several places in the menu. Therefore we will calll a helper

method, `private void saveWindowToCurrentFile()`. You should make an empty shell for this method and compile. If you run the program, you will see a save item in the `File` menu.

We will now take care of the implementing our helper method. To get started on this, we open an object output stream to the current file. We will also set up the exception-handling code.

```
private void saveWindowToCurrentFile()
{
    try
    {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(currentFile));
    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Now what do we save? Our rule is *Stuff the State!* Our state variables need to contain all information necessary to reconstruct the fine work of art we are creating. We are going to stuff the state of our app in the file. However, we do not need all of the elements to do this. For example we do not need the draw panel or the current file.

What methods are pertinent. You can write any primitive type. For example,

```
oos.writeInt(5);
```

will write the integer 5 to the object output stream. You can look in the API guide page to see all of the methods for writing primitive data. Any object you write must implement `Serializable`. If the object is a container, such as an `ArrayList<T>`, the type `T` of the entries, must also be serializable. Remember, if an ancestor class implements `Serializable`, you are in business.

We now look at all of our state variables.

```
private DrawPanel dp;
```

```
private JMenuBar mbar;
private Color background;//color of background
private Color foreground;//color of current curve
private int width;
private ArrayList<Curve> drawing;
private File currentFile;
```

We ask, "What is needed ot reconstruct our drawing?" We will not need the draw panel or the menu bar. We ditch them. We do need `foreground`, `background` and `width` to save the state of our session properly. Since `Color` implements `Serializable`, we are OK here. You can see this on the `java.awt.Color` page.

What abut the drawing? It is an `ArrayList` which is serializable. What about the entries? These are of type `Curve`, which is a child class of `ArrayList<Point>`. Finally, note that `java.awt.Point` implements `Serializable`, so we are in the clear. We can safely deflate all of our objects.

Here is our implementation.

```
private void saveWindowToCurrentFile()
{
    try
    {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(currentFile));
        oos.writeObject(background);
        oos.writeObject(foreground);
        oos.writeInt(width);
        oos.writeObject(drawing);
    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Now run the app, make a drawing and save it. Change colors, use curves of various widths and fool around. Remember its appearance, because will will resurrect it. Then look at the file's properties. In Linux, we see this

```
$ ls -l test.unid
```

```
-rw-rw-r-- 1 morrison morrison 774 Jan 22 19:59 test.unid
$
```

Our drawing occupies 774 bytes. If you open it with a text editor, you will see gibberish. So now let's see if we can resurrect our drawing. You can do this in Windoze by right clicking to see file properties.

Let us begin by creating an open item in the menu and attaching an action listener to it. We will create a helper method named `private void readFromCurrentFile()` to do the dirty work.

```
public void makeFileMenu()
{
    JMenu fileMenu = new JMenu("File");
    mbar.add(fileMenu);
    JMenuItem saveItem = new JMenuItem("Save");
    fileMenu.add(saveItem);
    saveItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            saveWindowToCurrentFile();
        }
    });
    JMenuItem openItem = new JMenuItem("Open");
    fileMenu.add(openItem);
    openItem.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e)
        {
            readFromCurrentFile();
        }
    });
}
```

Now we are going to resurrect our drawing. You will not be surprised to learn we use an `ObjectInputStream` to do it. When you serialize, you have a responsibility. You must

- Retrieve the items you stored in the order in which you stored them
- Remember their types. All of this information is lost.
- If you used `writeObject`, you must use `readObject()` and *cast to the correct type*. If you used `writeInt`, you use `readInt()` to retrieve the integer. This same rule applies for all primitive types.
- Handle a `ClassCastException`. This is not a runtime exception but it is most likely a programmer error. Print a stack trace in this `catch` block to see where the error occurred.

Suitably warned, we sally forth with the customary insouciance.

```
private void readFromCurrentFile()
{
    try
    {
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(currentFile));
        background = (Color) ois.readObject();
        foreground = (Color) ois.readObject();
        width = ois.readInt();
        drawing = (ArrayList<Curve>) ois.readObject();
        ois.close();
        dp.repaint();

    }
    catch(FileNotFoundException ex)
    {
        System.err.printf("File %s not found\n",
             currentFile.getAbsolutePath());
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
    catch(ClassNotFoundException ex)
    {
        ex.printStackTrace();
    }
}
```

We have met our responsibilities. Now compile, select the open menu item and your drawing will reappear!

10 Making the `File` Menu

We need to design the file menu with care. Remember: *Data loss to the user is a cardinal sin!* We must be very careful to think through the user experience very carefully to get this complex process right. So let us skech out each item's action prior to coding and determine what helper methods are needed.

New If you have a drawing in the window already, we need to offer to save it. If the user says yes, save the file. In any event, blank the window and set everything to default.

open If you have a drawing in the window already, we need to offer to save it. If the user says yes, save the file. In any event, have the user select a file to open and then place it in the window.

Save If the current file is null, have the user choose a file. If the file is saved, do nothing. If not, write the contents of the window to the current file.

Save As... This will be a menu. It will offer two menu items: to save as a drawing, a `.jpeg` or a `.png`.

Save As → drawing Fire up a file chooser to choose where to save. Then, if the user opts to save, save it there.

Save As → JPEG Fire up a file chooser to choose where to save. Then, if the user opts to save it, save it as a `.jpeg`

Quit Offer to save any window conents, then quit. Cancel if the user declines when asked if he really wants to quit.

Now we examine these menu items. We should write helper methods that are atomic, i.e. these methods accomplish a single task.