**Week 4 – An Interactive Drawing Application in Java**

**Learning Outcomes**

By the end this lesson, you will:

* Create interactive applications that adhere to the Model-View-Controller pattern.
* Write code that responds to a variety of events, including clicking the mouse on a frame or panel, moving or dragging the mouse, and typing a key on the keyboard.
* Implement animation using a timer and a corresponding event handler.
* Build a menu system that enables the user to trigger a variety of actions in a familiar way.
* Create multiple, intuitive ways for a user to perform a particular task.
* Design and implement a controller class that outputs data to a text file.
* Design and implement a controller class that inputs data from a text file and builds a collection of objects from the read data.
* Describe how Java achieves speedier input and output through its hierarchy of input and output classes.

We will achieve these learning outcomes by writing an example together in stages from start to finish. Each stage will prepare you to achieve one or more of these objectives.

**LineDrawer Example**

In this example, we create an application that enables the user to draw a series of connected lines by clicking on the application with the mouse.

The new skills we’ll learn in this example are

1. How to respond to mouse events.
2. How to set up a menu system.

The skills we’ll review in this example are

1. How to define a model class.
2. How to create a view class.
3. How to separate model from view.
4. How to create a JFrame that holds a JPanel in which we do some drawing.
5. How to respond to ActionEvents.

**The Model Class – Point**

The drawing the user will create will consist of Point objects. Lines will join the various Point objects. Each Point will be represented on the display as a little circle.

The Point class will have just an x value and a y value, which will represent its location. The get and set functions don’t need to do anything special to access and change the values of these parameters.

Here is the Point class:

class Point {

private int x;

private int y;

public int getX() {

return x;

}

public int getY() {

return y;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public Point() {

x = 0;

y = 0;

}

public Point(int x, int y) {

setX(x);

setY(y);

}

public String toString() {

return String.format("%d %d", x, y);

}

}

**Implementing a list of points**

The main function in the main class will create the list of points. This list of points will be passed to the JFrame class, which, in turn, will pass the list of points to the panel, so that the panel can draw them and the lines between them.

public class LineDrawer {

public static void main(String[] args) {

ArrayList<Point> points = new ArrayList<Point>();

}

}

We won’t add the points yet. That’s the job of the code that handles the user clicking on the panel.

**The View**

The View will be implemented as a JFrame called LineFrame. To create the LineFrame, we will pass the list of points. The LineFrame will contain an instance of LinePanel. The LinePanel, because it will be drawing the points and the lines between them, will need to have the list of points shared with it.

Here is the code for LineFrame:

class LineFrame extends JFrame {

private LinePanel pan;

private ArrayList<Point> points;

public LineFrame(ArrayList<Point> points) {

this.points = points;

setupUI();

}

public void setupUI() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

pan = new LinePanel(points);

Container c = getContentPane();

c.setLayout(new BorderLayout());

c.add(pan, BorderLayout.CENTER);

}

}

Notice how LineFrame has a private data member called points that keeps track of the Point objects. In this example, both the frame and the panel inside it will keep track of the points, because we’re later going to be adding a menu to the frame that will need to access the points.

Here is the code for the LinePanel. Notice how its paintComponent function plots each Point as a filled oval and draws a line between them. It can do this because it has access to the list of Point objects.

class LinePanel extends JPanel {

private ArrayList<Point> points;

public LinePanel(ArrayList<Point> points) {

this.points = points;

}

public void paintComponent(Graphics g) {

super.paintComponent(g);

Point prevPoint = null;

for (Point p : points) {

g.fillOval(p.getX(),p.getY(),5,5);

if (prevPoint != null) {

g.drawLine(prevPoint.getX(),prevPoint.getY(),

p.getX(),p.getY());

}

prevPoint = p;

}

}

}

**Adding Mouse Handling**

To respond to mouse events, we need a class to implement MouseListener. MouseListener is an interface defined in java.awt.event that provides stubs for five functions:

* public void mouseEntered(MouseEvent e);
* public void mouseExited(MouseEvent e);
* public void mouseClicked(MouseEvent e);
* public void mousePressed(MouseEvent e);
* public void mouseReleased(MouseEvent e);

We could create a separate, standalone class that implements MouseListener, or we could have our LinePanel be its own MouseListener by having it implement MouseListener. We then have to tell the LinePanel to add itself as its MouseListener.

Because a MouseListener implementer must provide implementations for all five functions, some or most of the functions will likely by empty. In this particular case, we have to provide an implementation for just mouseClicked. When the mouse is clicked, we want to add a new Point to the list at the location where the mouse was clicked. That location can be identified using the MouseEvent e, which has functions getX() and getY() to return where the mouse was pressed.

Here is how the code for LinePanel looks now that it has been equipped with the ability to listen for mouse events:

class LinePanel extends JPanel implements MouseListener {

private ArrayList<Point> points;

public LinePanel(ArrayList<Point> points) {

this.points = points;

addMouseListener(this);

}

public void mousePressed(MouseEvent e) {

}

public void mouseClicked(MouseEvent e) {

points.add(new Point(e.getX(),e.getY()));

repaint();

}

public void mouseReleased(MouseEvent e) {

}

public void mouseEntered(MouseEvent e) {

}

public void mouseExited(MouseEvent e) {

}

public void paintComponent(Graphics g) {

super.paintComponent(g);

Point prevPoint = null;

for (Point p : points) {

g.fillOval(p.getX(),p.getY(),5,5);

if (prevPoint != null) {

g.drawLine(prevPoint.getX(),prevPoint.getY(),

p.getX(),p.getY());

}

prevPoint = p;

}

}

}

Note that there are alternatives to how we handled mouse events. We could have declared a separate class that implements MouseListener. Or, we could have created a class that extends MouseAdapter. An Adapter class is one that implements an interface by providing blank implementations for every one of its functions. Any class that extends MouseAdapter then has to override only the function or functions it cares about, rather than providing blank implementations for every function like we did in the LinePanel above.

**Implementing a Menu**

We conclude this example by adding a menu system to LineFrame.

There is an easy-to-understand hierarchy of classes that we use to create menus:

* JMenuBar is the menu that goes at the top of the frame.
* JMenu items are the main ones that hang off the JMenuBar
* JMenuItems are the sub-items that belong to each JMenu.

You create the JMenuBar. You create JMenu items that have captions (like File, Edit, etc.). You create JMenuItem objects that have captions (like Open, Close, Save, etc.) and add them to the JMenu’s. You associate ActionListener objects with each JMenuItem to describe what should happen when the JMenuItem is clicked. You add each JMenuItem to the JMenu. You add the JMenu to the JMenuBar. You then tell the frame to set its JMenuBar using setJMenuBar.

Here’s the LineFrame code, now with a setupMenu function that creates a menu that has a File and Edit menu. Under File, there is an Exit option that closes the application. Under Edit, there is a Clear option that tells the list of points to clear itself, and then tells the LineFrame to repaint. In telling the LineFrame to repaint, we automatically tell the LinePanel to repaint. Because the LinePanel and the LineFrame refer to the same list of points, which was just cleared, the LinePanel will display no points or lines, because there are none to display, since the list is now empty. Note that the menu handlers were implemented using anonymous inner classes that implement ActionListener.

class LineFrame extends JFrame {

private LinePanel pan;

private ArrayList<Point> points;

public LineFrame(ArrayList<Point> points) {

this.points = points;

setupUI();

}

public void setupMenu() {

JMenuBar mbar = new JMenuBar();

JMenu mnuFile = new JMenu("File");

JMenuItem miExit = new JMenuItem("Exit");

miExit.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.exit(0);

}

});

mnuFile.add(miExit);

mbar.add(mnuFile);

JMenu mnuEdit = new JMenu("Edit");

JMenuItem miClear = new JMenuItem("Clear");

miClear.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

points.clear();

repaint();

}

});

mnuEdit.add(miClear);

mbar.add(mnuEdit);

setJMenuBar(mbar);

}

public void setupUI() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setupMenu();

pan = new LinePanel(points);

Container c = getContentPane();

c.setLayout(new BorderLayout());

c.add(pan, BorderLayout.CENTER);

}

}

public class LineDrawer {

public static void main(String[] args) {

ArrayList<Point> points = new ArrayList<Point>();

LineFrame lf = new LineFrame(points);

lf.setBounds(10,10,400,400);

lf.setVisible(true);

}

}

**Working with Text Files**

**Saving and Loading Text Files**

We now add the ability to save and to load Circle data from a file.

The fancy word for reading and writing data objects from / to external media is called *serialization*. Java and C# and other modern languages provide a *serialization interface* that automates some of the grunt work associated with doing this. Still, if all you’re after is reading and writing text files, there isn’t a lot of grunt work anyway, and so just reading and writing the data “by hand” isn’t too burdensome.

To read data from a text file:

* Create a File object, feeding the file’s path to the File class constructor
* Attach a Scanner object to it.
* Use Scanner’s readLine and hasNextLine functions to read the file from beginning to end.

To write a text file,

* Create a File object, feeding the file’s path to the File class constructor
* Create a FileWriter to access the File.
* Create a BufferedWriter to dump data to the FileWriter efficiently (in blocks rather than as individual characters)
* Use BufferedWriter’s write and newLine functions to commit the data to the file.

Here’s the code for a new class called CircleIO that attempts to read and write Circle data. You can think of this as a controller, with the model being Circle and the view – the way the model is shown to the world – being the file on disk.

**class** CircleIO {

**public** **static** **boolean** saveToFile(Circle c) {

JFileChooser jfc = **new** JFileChooser();

File file;

**try** {

**if** (jfc.showSaveDialog(**null**) == JFileChooser.*APPROVE\_OPTION*) {

file = jfc.getSelectedFile();

BufferedWriter bw = **new** BufferedWriter(**new** FileWriter(file));

bw.write(c.toString());

bw.newLine();

bw.close();

}

**return** **true**;

} **catch** (Exception e) {

**return** **false**;

}

}

**public** **static** Circle loadFromFile() {

JFileChooser jfc = **new** JFileChooser();

File file;

String line;

String[] parts;

Circle c = **null**;

**try** {

**if** (jfc.showOpenDialog(**null**) == JFileChooser.*APPROVE\_OPTION*) {

file = jfc.getSelectedFile();

Scanner sc = **new** Scanner(file);

**while** (sc.hasNextLine()) {

line = sc.nextLine().trim();

parts = line.split(" ");

c = **new** Circle(Integer.*parseInt*(parts[0]),

Integer.*parseInt*(parts[1]),

Integer.*parseInt*(parts[2]),

**new** Color(Integer.*parseInt*(parts[3]),

Integer.*parseInt*(parts[4]),

Integer.*parseInt*(parts[5])));

}

}

**return** c;

} **catch** (Exception e) {

**return** **null**;

}

}

}

**HANDLING KEY EVENTS**

So far, we have learned how to work with Action Events that arise when a user clicks on a button and Mouse Events when a user clicks on a panel. It turns out you can also respond to key events.

To detect and respond to a key event, you have to implement the KeyListener interface. KeyListener has three functions:

public void keyPressed(KeyEvent e)

public void keyReleased(KeyEvent e)

public void keyTyped(KeyEvent e)

The KeyEvent object helps us identify which key was pressed, typed, or released. We can identify the key that was used using the KeyEvent class’s getKeyCode() function and comparing its return value with any of what are called virtual key codes. Virtual key codes begin with VK\_ and end with an identification of the key that was pressed. For example, VK\_A represents the A key, and VK\_ENTER represents the enter key.

In this next example, we add a JTextField to enable the user to set the size of the points in the drawing. They can finalize their selection either by clicking the “Set” button or by typing ENTER when they are typing in the text field.

We’ve bolded the affected code. Notice how the frame itself serves as the KeyListener. It can because it implements the KeyListener interface.

class LineFrame extends JFrame **implements KeyListener** {

private LinePanel pan;

private ArrayList<Point> points;

**private JTextField txtSize;**

**public void keyPressed(KeyEvent e) {**

**if (e.getKeyCode()== KeyEvent.VK\_ENTER) {**

**pan.setPointSize(Integer.parseInt(txtSize.getText()));**

**}**

**}**

**public void keyTyped(KeyEvent e) {**

**}**

**public void keyReleased(KeyEvent e) {**

**}**

public LineFrame(ArrayList<Point> points) {

this.points = points;

setupUI();

}

public void setupMenu() {

JMenuBar mbar = new JMenuBar();

JMenu mnuFile = new JMenu("File");

JMenuItem miExit = new JMenuItem("Exit");

miExit.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.exit(0);

}

});

mnuFile.add(miExit);

mbar.add(mnuFile);

JMenu mnuEdit = new JMenu("Edit");

JMenuItem miClear = new JMenuItem("Clear");

miClear.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

points.clear();

repaint();

}

});

mnuEdit.add(miClear);

mbar.add(mnuEdit);

setJMenuBar(mbar);

}

public void setupUI() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setupMenu();

pan = new LinePanel(points);

Container c = getContentPane();

c.setLayout(new BorderLayout());

c.add(pan, BorderLayout.CENTER);

**JPanel entryPanel = new JPanel();**

**entryPanel.setLayout(new FlowLayout());**

**JLabel lblSize = new JLabel("Point size:");**

**txtSize = new JTextField(5);**

**txtSize.addKeyListener(this);**

**JButton btnSize = new JButton("Set");**

**btnSize.addActionListener(new ActionListener() {**

**public void actionPerformed(ActionEvent e) {**

**int size = Integer.parseInt(txtSize.getText());**

**pan.setPointSize(size);**

**}**

**}**

**);**

**entryPanel.add(lblSize);**

**entryPanel.add(txtSize);**

**entryPanel.add(btnSize);**

**c.add(entryPanel,BorderLayout.SOUTH);**

}

}

**A VERSION OF LINEDRAWER WITH FILE INPUT AND OUTPUT and HANDLING OF KEY EVENTS**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import java.util.ArrayList;

import java.io.\*;

import java.util.Scanner;

class PointIO {

public static boolean writeFile(ArrayList<Point> points, String fname) {

return writeFile(points, new File(fname));

}

public static ArrayList<Point> readFile(String fname) {

return readFile(new File(fname));

}

public static boolean writeFile(ArrayList<Point> points, File file) {

try {

FileWriter fw = new FileWriter(file);

BufferedWriter bw = new BufferedWriter(fw);

for (Point p : points) {

bw.write(p.toString());

bw.newLine();

}

bw.close();

return true;

} catch (Exception e) {

return false;

}

}

public static ArrayList<Point> readFile(File file) {

ArrayList<Point> result;

Scanner sc;

try {

sc = new Scanner(file);

result = new ArrayList<Point>();

String line;

String[] parts;

Point p;

while (sc.hasNextLine()) {

line = sc.nextLine();

parts = line.split(" ");

p = new

Point(Integer.parseInt(parts[0]),Integer.parseInt(parts[1]));

result.add(p);

}

return result;

} catch (Exception ex) {

return null;

}

}

}

class Point {

private int x;

private int y;

public int getX() {

return x;

}

public int getY() {

return y;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public Point() {

x = 0;

y = 0;

}

public Point(int x, int y) {

setX(x);

setY(y);

}

public String toString() {

return String.format("%d %d", x, y);

}

}

class LinePanel extends JPanel implements MouseListener {

private ArrayList<Point> points;

private int pointSize;

public LinePanel(ArrayList<Point> points) {

this.points = points;

pointSize = 5;

addMouseListener(this);

}

public void setPointSize(int size) {

pointSize = Math.abs(size);

if (size == 0) {

pointSize = 5;

}

}

public void mousePressed(MouseEvent e) {

}

public void mouseClicked(MouseEvent e) {

points.add(new Point(e.getX(),e.getY()));

repaint();

}

public void mouseReleased(MouseEvent e) {

}

public void mouseEntered(MouseEvent e) {

}

public void mouseExited(MouseEvent e) {

}

public void paintComponent(Graphics g) {

super.paintComponent(g);

Point prevPoint = null;

for (Point p : points) {

g.fillOval(p.getX(),p.getY(),pointSize,pointSize);

if (prevPoint != null) {

g.drawLine(prevPoint.getX(),prevPoint.getY(),p.getX(),p.getY());

}

prevPoint = p;

}

}

}

class LineFrame extends JFrame implements KeyListener {

private LinePanel pan;

private ArrayList<Point> points;

private JTextField txtSize;

public void keyPressed(KeyEvent e) {

if (e.getKeyCode()== KeyEvent.VK\_ENTER) {

pan.setPointSize(Integer.parseInt(txtSize.getText()));

}

}

public void keyTyped(KeyEvent e) {

}

public void keyReleased(KeyEvent e) {

}

public LineFrame(ArrayList<Point> points) {

this.points = points;

setupUI();

}

public void setupMenu() {

JMenuBar mbar = new JMenuBar();

JMenu mnuFile = new JMenu("File");

JMenuItem miSave = new JMenuItem("Save");

miSave.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

if (PointIO.writeFile(points,"points.txt")) {

JOptionPane.showMessageDialog(null,"Points saved.");

} else {

JOptionPane.showMessageDialog(null,"Points could not be

saved.");

}

}

});

mnuFile.add(miSave);

JMenuItem miOpen = new JMenuItem("Open");

miOpen.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

ArrayList<Point> readPoints;

readPoints = PointIO.readFile("points.txt");

if (readPoints != null) {

points.clear();

for (Point p: readPoints) {

points.add(p);

}

repaint();

} else {

JOptionPane.showMessageDialog(null,"File could not be

read.");

}

}

});

mnuFile.add(miOpen);

JMenuItem miExit = new JMenuItem("Exit");

miExit.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.exit(0);

}

});

mnuFile.add(miExit);

mbar.add(mnuFile);

JMenu mnuEdit = new JMenu("Edit");

JMenuItem miClear = new JMenuItem("Clear");

miClear.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

points.clear();

repaint();

}

});

mnuEdit.add(miClear);

mbar.add(mnuEdit);

setJMenuBar(mbar);

}

public void setupUI() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setupMenu();

pan = new LinePanel(points);

Container c = getContentPane();

c.setLayout(new BorderLayout());

c.add(pan, BorderLayout.CENTER);

JPanel entryPanel = new JPanel();

entryPanel.setLayout(new FlowLayout());

JLabel lblSize = new JLabel("Point size:");

txtSize = new JTextField(5);

txtSize.addKeyListener(this);

JButton btnSize = new JButton("Set");

btnSize.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

int size = Integer.parseInt(txtSize.getText());

pan.setPointSize(size);

}

}

);

entryPanel.add(lblSize);

entryPanel.add(txtSize);

entryPanel.add(btnSize);

c.add(entryPanel,BorderLayout.SOUTH);

}

}

public class LineDrawer {

public static void main(String[] args) {

ArrayList<Point> points = new ArrayList<Point>();

LineFrame lf = new LineFrame(points);

lf.setBounds(10,10,400,400);

lf.setVisible(true);

}

}

**Handling MouseMotion Events**

In addition to MouseEvents, Java can respond to MouseMotion events like moving and dragging. To respond to such events, you need a class that implements MouseMotionListener. MouseMotionListener objects need to implement two function:

public void mouseMoved(MouseEvent e)

public void mouseDragged(MouseEvent e)

The most important piece of data in these cases comes from e through its getX() and getY() functions, which together reveal the whereabouts of the mouse.

Here’s code that will display the coordinates of the mouse as you move around. We’ve bolded the affected code.

**class** LinePanel **extends** JPanel **implements** MouseListener, **MouseMotionListener** {

**private** ArrayList<Point> points;

**private** **int** pointSize;

**private** String positionString;

**public** LinePanel(ArrayList<Point> points) {

**this**.points = points;

pointSize = 5;

positionString = "";

addMouseListener(**this**);

addMouseMotionListener(**this**);

}

**public** **void** setPointSize(**int** size) {

pointSize = Math.*abs*(size);

**if** (size == 0) {

pointSize = 5;

}

}

**public void mouseMoved(MouseEvent e) {**

**positionString = String.*format*("(%d, %d)",e.getX(),e.getY());**

**repaint();**

**}**

**public void mouseDragged(MouseEvent e) {**

**}**

**public** **void** mousePressed(MouseEvent e) {

}

**public** **void** mouseClicked(MouseEvent e) {

points.add(**new** Point(e.getX(),e.getY()));

repaint();

}

**public** **void** mouseReleased(MouseEvent e) {

}

**public** **void** mouseEntered(MouseEvent e) {

}

**public** **void** mouseExited(MouseEvent e) {

}

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

Point prevPoint = **null**;

**for** (Point p : points) {

g.fillOval(p.getX(),p.getY(),pointSize,pointSize);

**if** (prevPoint != **null**) {

g.drawLine(prevPoint.getX(),prevPoint.getY(), p.getX(),p.getY());

}

prevPoint = p;

}

**g.drawString(positionString, 300, 300);**

}

}

**Timer Events and Basic Animation**

We do one more thing to this example: we add a timer that, every second, will cause the points to move.

There are a few different times in Java. The one we will use is javax.swing.Timer.

To create one, do this:

Timer tim = new Timer(interval, thelistenerobject)

Here, thelistenerobject is an object of a class that implements ActionListener. This ActionListener implementer will fire whenever the interval (expressed in milliseconds) elapses. In our example, we will write an actionPerformed function that will change the x and y coordinates of each point and then tell the frame to repaint.

Here is the updated code:

**class** LinePanel **extends** JPanel **implements** MouseListener, MouseMotionListener, ActionListener {

**private** ArrayList<Point> points;

**private** Timer tim;

**private** **int** pointSize;

**private** String positionString;

**public** LinePanel(ArrayList<Point> points) {

**this**.points = points;

pointSize = 5;

positionString = "";

**tim = new Timer(1000,this);**

addMouseListener(**this**);

addMouseMotionListener(**this**);

**tim.start();**

}

**public void actionPerformed(ActionEvent e) {**

**if (points != null) {**

**for (Point p : points) {**

**p.setX(p.getX()+5);**

**p.setY(p.getY()+5);;**

**}**

**}**

**repaint();**

**}**

**public** **void** setPointSize(**int** size) {

pointSize = Math.*abs*(size);

**if** (size == 0) {

pointSize = 5;

}

}

**public** **void** mouseMoved(MouseEvent e) {

positionString = String.*format*("(%d, %d)",e.getX(),e.getY());

repaint();

}

**public** **void** mouseDragged(MouseEvent e) {

}

**public** **void** mousePressed(MouseEvent e) {

}

**public** **void** mouseClicked(MouseEvent e) {

points.add(**new** Point(e.getX(),e.getY()));

repaint();

}

**public** **void** mouseReleased(MouseEvent e) {

}

**public** **void** mouseEntered(MouseEvent e) {

}

**public** **void** mouseExited(MouseEvent e) {

}

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

Point prevPoint = **null**;

**for** (Point p : points) {

g.fillOval(p.getX(),p.getY(),pointSize,pointSize);

**if** (prevPoint != **null**) {

g.drawLine(prevPoint.getX(),prevPoint.getY(),p.getX(),p.getY());

}

prevPoint = p;

}

g.drawString(positionString, 300, 300);

}

}