Introduction to Computer Programming

Using the Java Programming Language

9. Graphical User Interface

What is Swing?

JAVA provides a rich set of libraries to create Graphical User Interfaces in a platform independent way. For this course, we will be using the SWING Graphical User Interface controls.

The Swing API is set of extensible GUI Components to ease the developer's life to create JAVA based Front End / GUI Applications. It is built on top of the AWT API and acts as replacement of the AWT API as it has almost every control corresponding to AWT controls.

Swing Features

- **Light Weight** - Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.

- **Rich controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.

- **Highly Customizable** - Swing controls can be customized in a very easy way as visual appearance is independent of internal representation.

- **Pluggable look-and-feel**- SWING based GUI Application look and feel can be changed at run time based on available values.

Our First Window

Let's create our first window! Up until now in the course we have been doing all our input and output in the console window. It's time to start writing graphical user interfaces. Do not worry if you don't understand every line in this code. In the beginning you can merely copy portions of the examples I give you to complete the various tasks. Once we start using more and more features, it will slowly come to you.

First the code, and then I will explain the lines individually.

```
1    import javax.swing.*;

     public class FirstWindow {

           public static void main(String[] args) {
                 FirstWindow w = new FirstWindow();
                 w.setupWindow();
           }

           public void setupWindow () {
4                JFrame f = new JFrame("First");
5                f.setTitle("First Window!");
6                f.setSize(300, 200);
7                f.setLocationRelativeTo(null);
8                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9                f.setVisible(true);

           }
     }
```

See next page for a breakdown of each line of code and it's meaning.

Line 1. We import all of the Swing classes so that they can be used.

Line 4. Create a new JFrame component. A `JFrame` is a top level container for windows. The basic purpose of the container is to hold components for your application.

Line 5. We set the title of the window that will appear on the screen to “First Window!”.

Line 6. We resize the window to be 300 pixels wide and 200 pixels tall.

Line 7. We center the window on the screen.

Line 8. If the close button is clicked on the menu bar (the red dot on an Apple computer), this will close the application. Without it, the window would close but the application would still be running.

Line 9. We set the window to be visible on the screen.

Adding a Button

A button is one component that can be added to your window. There are many different components, but we will take it slow. Buttons are useful and you may want to add one to your program relatively early on in your learning!

First the code, and then I will explain the lines individually.

```
1    import java.awt.*;
2    import javax.swing.*;

     public class ButtonWindow {

           public static void main(String[] args) {
                 ButtonWindow w = new ButtonWindow();
                 w.setupWindow();
           }
           public void setupWindow() {
3                JFrame f = new JFrame("First Window");
4                JPanel p = new JPanel();

5                p.setLayout( new FlowLayout());

6                JButton b1 = new JButton("Button 1");
7                p.add(b1);
8                JButton b2 = new JButton("Button 2");
9                p.add(b2);

10               f.add(p, "Center");

11               f.setSize(300, 300);
12               f.setLocationRelativeTo(null);
13               f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14               f.setVisible(true);
           }

     }
```

See next page for a breakdown of each line of code and it's meaning.

Lines 1 & 2. We import all of the Swing and AWT classes so that they can be used. AWT preceded Swing and is still used for graphical user interfaces.

Line 3. Create a new `JFrame` component. A `JFrame` is a top level container for windows. The basic purpose of the container is to hold components for your application.

Line 4. Create a `JPanel` component inside the `JFrame`. A `JPanel` is just a container that is used to group things together.

Line 5. Set the layout for the panel. There are various layouts and we will go into detail later. For now we use the flow layout that just places components in a “flowing manner inside the window.

Lines 6-9. Create two buttons and add them to the panel that we just created.

Line 10. Add the panel to the frame that we created before. The panel is now “inside” the frame.

Lines 11-14. Same as last example program.

Adding a Button with Listener

When a button is clicked, you need to provide a method to Java that will get called in your program. This is the first time that your programs will not run in a sequential manner. The “listener” will get called by the Java runtime when the user clicks a button.

First the code, and then I will explain the lines individually.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;

4  public class ButtonHandlerWindow implements ActionListener {

5       private JButton b1;
6       private JButton b2;
7       private JButton b3;

8       public void actionPerformed(ActionEvent e) {
9               if (e.getSource() == b1)
10                      System.out.println("Button 1 was pressed");
11              else if (e.getSource() == b2)
12                      System.out.println("Button 2 was pressed");
13              else if (e.getSource() == b3)
14                      System.exit(0);
15      }

16      public void setupWindow() {
17              JFrame f = new JFrame("My Window");
18              JPanel p = new JPanel();

19              p.setLayout( new FlowLayout());

20              b1 = new JButton("Button 1");
21              b1.addActionListener(this);
22              p.add(b1);

23              b2 = new JButton("Button 2");
24              b2.addActionListener(this);
25              p.add(b2);

26              b3 = new JButton("EXIT");
27              b3.addActionListener(this);
28              p.add(b3);

29              f.add(p, "Center");

30              f.setSize(300, 300);
31              f.setLocationRelativeTo(null);
32              f.setVisible(true);
        }


33      public static void main(String[] args) {
34              ButtonHandlerWindow w = new ButtonHandlerWindow();
35              w.setupWindow();
        }
}
```

See next page for a breakdown of each line of code and it’s meaning.

Lines 1, 2 & 3. We import all of the swing and AWT classes so that they can be used. AWT preceded Swing and is still used for graphical user interfaces.

Line 4. We want to implement an interface in our class (the `ActionListener)`. This is what allows us to get notified when certain things happen in our window.

Lines 5, 6, & 7. Declare our buttons each as a `JButton`.

Lines 8 to 15. This is a sample `actionListener. actionPerformed` gets called when certain actions happen in our window. We will attach this to our buttons later in the program.

Line 17. Create a new `JFrame` component. A `JFrame` is a top level container for windows. The basic purpose of the container is to hold components for your application.

Line 18. Create a `JPanel` component inside the `JFrame`. A `JPanel` is just a container that is used to group things together.

Line 19. Set the layout for the panel. There are various layouts and we will go into detail later. For now we use the flow layout that just places components in a "flowing manner inside the window.

Lines 20 to 28. Create three buttons and add them to the panel that we just created. In addition, attach the `ActionListener` using `this` to indicate that it's the current object.

Line 29. Add the panel to the frame that we created before. The panel is now "inside" the frame.

Line 30. Set the dimensions of the window.

Line 31. Tell it where to appear on the screen.

Line 31. Set the window to be visible.

Line 34. Instantiate the object.

Line 35. Call our method that sets everything up.

Using a JLabel

One easy way to display text in a window is to use a JLabel. The use of a JLabel is very similar to the other components we have learned. Here is a sample of the code with explanations of any new lines that have not yet been seen.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JLabelExample implements ActionListener {

       private JButton b1;
       private JButton b2;
1      JLabel l;

       public void actionPerformed(ActionEvent e) {
              if (e.getSource() == b1)
2                     l.setText("New Text!!!");
              else if (e.getSource() == b2)
                     System.exit(0);
       }

       public void setupWindow() {
              JFrame f = new JFrame("First Window");
              JPanel p = new JPanel();

              p.setLayout( new FlowLayout());

              b1 = new JButton("Change Text");
              b1.addActionListener(this);
              p.add(b1);

              b2 = new JButton("EXIT");
              b2.addActionListener(this);
              p.add(b2);

3             l = new JLabel("Original Text");
4             p.add(l);

              f.add(p, "Center");

              f.setSize(300, 300);
              f.setLocationRelativeTo(null);
              f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
              f.setVisible(true);
       }


       public static void main(String[] args) {

              JLabelExample w = new JLabelExample();
              w.setupWindow();
       }

}
```

See next page for a breakdown of each line of code and it's meaning.

Line 1. We declare the variable `l` as a `JLabel` component.

Line 2. We use the method `setText` to set new text into the `JLabel`.

Line 3 & 4. We create a new `JLabel` and add it to the `JPanel`. Remember we are using Flow layout so it will just be placed after the last button.

Drawing Graphics in your Window

It is fun and easy to draw graphics in your window. We will draw directly into a `JPanel` that will be placed inside a `JFrame`. This example shows the basics of drawing a few different shapes. The key concept of drawing is to extend the `JPanel` and create your own `PaintComponent`. Explanations of the new lines are found on the next page.

```
      import java.awt.*;
      import javax.swing.*;

1     public class DrawShapesPanel extends JPanel {

2           public void paintComponent(Graphics g) {
3                 super.paintComponent(g);
4                 g.setColor(Color.RED);
5                 g.fillRect(10, 10, 200, 100);

6                 g.setColor(Color.BLUE);
7                 g.drawRect(250, 10, 100, 150);

8                 g.setColor(Color.GREEN);
9                 g.drawOval(10, 200, 100, 100);

10                g.fillOval(250, 200, 150, 100);

11                g.setColor(Color.BLACK);
12                g.drawLine(0, 175, 600, 175);
            }

            public void setupWindow(DrawShapesPanel p) {
                  JFrame f = new JFrame("Drawing Example");

                  f.setSize(600, 400);
                  f.setLocationRelativeTo(null);
                  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  f.add(p);
                  f.setVisible(true);
            }

13          public static void main(String[] args) {
                  DrawShapesPanel p = new DrawShapesPanel();
                  p.setupWindow(p);

            }
      }
```

See next page for a breakdown of each line of code and it’s meaning.

Line 1. We are creating our own `JPanel` class that extends `JPanel`. Therefore we get everything that `JPanel` has, but we can add methods. We are going to add our own `paintComponent` method later.

Line 2. We add our `paintComponent` method that will get called when the window needs to be painted.

Line 3. Since we are creating our own `paintComponent`, we want to be sure we first call the `paintComponent` that is in the `JPanel` that we are extending. Then we can do anything in addition to that.

Line 4 & 5. Set the color to red and draw a filled rectangle (filled with that color).

Line 6 & 7. Set the color to blue and draw a non-filled rectangle (just the border).

Lines 8 & 9. Set the color to green and draw a non-filled oval. If you make the last two dimensions the same, the oval will be a circle.

Line 10. Draw a filled oval. This will use the last set color (green).

Lines 11 & 12. Set the color to black and draw a line that is horizontal through the mid section of the `JPanel`.

Line 13. Create a new instance of our own `JPanel`. In the past we would just create a new `JPanel`. In this case we want to create our `JPanel` that we are creating in this example since it extends `JPanel`.

Handling a Mouse Click

It is often useful to know when and where a user clicked the mouse in your window. You may want to perform an action when they click on your window (for drawing or for games). A very simple example of handling this event is shown here.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

1  public class MouseClickPanel extends JPanel implements MouseListener
{

2     public MouseClickPanel () {
3           addMouseListener(this);
4     }

5     public void mousePressed(MouseEvent evt) {
6           System.out.println("In Mouse Pressed");
7     }

8     public void mouseClicked(MouseEvent evt) {
9           System.out.println("In Mouse Clicked");
10    }

11    public void mouseReleased(MouseEvent e) {
12          System.out.println("In Mouse Released");
13    }

14    public void mouseEntered(MouseEvent e) {
15          System.out.println("In Mouse Entered");
16    }

17    public void mouseExited(MouseEvent e) {
18          System.out.println("In Mouse Exited");
19    }

      public void setupWindow (MouseClickPanel p) {
            JFrame f = new JFrame("Drawing Example");

            f.setSize(600, 400);
            f.setLocationRelativeTo(null);
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.add(p);
            f.setVisible(true);
      }

      public static void main(String[] args) {
20          MouseClickPanel p = new MouseClickPanel();
            p.setupWindow(p);
      }

}
```

See next page for a breakdown of each line of code and it's meaning.

Line 1. We are creating our own `JPanel` class that extends `JPanel`. Therefore we get everything that `JPanel` has, but we can add methods. We also implement the `MouseListener` interface so that we will get mouse events.

Lines 2, 3, & 4. We implement a constructor that adds the `MouseListener` that we are implementing to the current `JPanel` (this class).

Lines 5, 6 & 7. A method that will get called whenever a mouse button is pressed.

Lines 8, 9 & 10. A method that will get called whenever a mouse button is clicked.

Lines 11, 12, & 13. A method that will get called when the mouse button is released.

Lines 14, 15, & 16. A method that is called whenever the mouse is moved into your window (`JPanel`).

Lines 17, 18, & 19. A method that is called whenever the mouse is moved out of your window (`JPanel`).

Line 20. Create a new instance of our own `JPanel`. In the past we would just create a new `JPanel`. In this case we want to create our `JPanel` that we are creating in this example since it extends `JPanel`. This will help as you add on to this program to draw objects, etc.

Drawing an Object with the Mouse

It's fun to draw an object on the screen where the user clicks the mouse. This will also be necessary when you start to create a game for our final assignment. This small program illustrates how to draw one rectangle wherever the user clicks the mouse.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

1  public class PaintOnClickPanel extends JPanel implements MouseListener {

2      int x=-1, y=-1; //Initialize the x and y location so we know that
                       //the mouse has not been clicked yet while painting.

       public PaintOnClickPanel () {
              addMouseListener(this);
       }

3      public void paintComponent(Graphics g) {
4             super.paintComponent(g);
5             if (x != -1)
6                    g.fillRect(x, y, 20, 20);
7      }

8      public void mouseClicked(MouseEvent evt) {
9             x = evt.getX();
10            y = evt.getY();
11            repaint();
12     }

       public void mousePressed(MouseEvent evt) {
       }

       public void mouseReleased(MouseEvent e) {
       }

       public void mouseEntered(MouseEvent e) {
       }

       public void mouseExited(MouseEvent e) {
       }
       public void setupWindow (PaintOnClickPanel p) {

              JFrame f = new JFrame("Drawing Example");
              f.setSize(600, 400);
              f.setLocationRelativeTo(null);
              f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
              f.add(p);
              f.setVisible(true);
       }

       public static void main(String[] args) {
              PaintOnClickPanel p = new PaintOnClickPanel();
              p.setupWindow(p);
       }
}
```

See next page for a breakdown of each line of code and it's meaning.

Line 1. We are creating our own `JPanel` class that extends `JPanel`. Therefore we get everything that `JPanel` has, but we can add methods. We also implement the `MouseListener` interface so that we will get mouse events.

Line 2. Declare an x and y value for where the user clicks the mouse. We initially set it to -1 indicating that the user did not click the mouse yet.

Line 3. We add our `paintComponent` method that will get called when the window needs to be painted.

Line 4. Since we are creating our own `paintComponent`, we want to be sure we first call the `paintComponent` that is in the `JPanel` that we are extending. Then we can do anything in addition to that.

Line 5 & 6. If the user clicked the mouse, x will not be -1 anymore (it is set later in the `mouseClicked` event). So, draw a filled rectangle of dimensions 20 by 20 at the x and y location.

Lines 8 – 12. A method that gets called when the mouse button is clicked by the user. Grab the x and y location from the `evt` that gets passed to us with the `getX()` and `getY()` methods. Then call the `repaint()` method which will tell the system to call the `paintComponent` method above.

Drawing an Object with the Mouse (and Saving)

Now that you were able to draw a small rectangle where the user clicked, you will notice that each time you click a new location, the old rectangle goes away. This is due to the fact that you have to repaint the screen each time you are drawing a rectangle. To retain all of the previously clicked rectangles, you need to save the x and y locations each time you click. The easiest way to do this is to use an array of x and y locations of the clicks, along with a variable to keep track of how many clicks we received. You can limit your array to a reasonable number and only save that many clicks.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PaintOnClickPanel extends JPanel implements MouseListener {

1       private final int NUMCLICKS = 20;
2       private int [] x = new int[NUMCLICKS];
3       private int [] y = new int[NUMCLICKS];
4       private int numRects = 0;

5       public PaintOnClickPanel () {
                addMouseListener(this);
        }
        public void paintComponent(Graphics g) {
                super.paintComponent(g);
6               if (numRects != 0) {
7                       for (int i=0; i< numRects; i++)
8                               g.fillRect(x[i], y[i], 20, 20);
                }
        }
        public void mouseClicked(MouseEvent evt) {
9               if (numRects < NUMCLICKS) {
10                      x[numRects] = evt.getX();
11                      y[numRects]=evt.getY();
12                      numRects++;
13                      repaint();
                }
        }
        public void mousePressed(MouseEvent evt) {
        }
        public void mouseReleased(MouseEvent e) {
        }
        public void mouseEntered(MouseEvent e) {
        }
        public void mouseExited(MouseEvent e) {
        }
        public void setupWindow(PaintOnClickPanel p) {
                JFrame f = new JFrame("Drawing Example");
                f.setSize(600, 400);
                f.setLocationRelativeTo(null);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.add(p);
                f.setVisible(true);
                p.requestFocusInWindow();

        }
        public static void main(String[] args) {
                PaintOnClickPanel p = new PaintOnClickPanel();
                p.setupWindow(p);
        }
}
```

See next page for a breakdown of each line of code and it's meaning.

Lines 1 – 4. Create arrays for the x and y locations of each mouse click. Declare the array of size NUMCLICKS, which is a constant. This makes it easier to make the array bigger. NumRects will be used to keep track of how many rectangles we have drawn so far.

Line 5. In the constructor for the class, attach the `MouseListener` to the object.

Lines 6 - 8. When painting the panel (during a repaint for instance) loop through the arrays and draw a rectangle at each of the locations that have been clicked so far.

Lines 9-13. When a mouse is clicked, save the x and y location in the array, bump up the number of rectangles that we have drawn, and force a repaint of the screen. Check to make sure we didn't store too many rectangles before doing this.

A Simple Timer

For animation, we must continually re-draw the window moving object positions to make them appear as if they are moving. For this, we need a timer. This code is a very simple timer application that you could incorporate into your code.

```
import java.awt.event.*;
import javax.swing.*;

public class SimpleAnimation implements ActionListener {

      public SimpleAnimation () {
1           Timer tm = new Timer(500, this);
2           tm.start();
      }

3     public void actionPerformed (ActionEvent e) {
4           System.out.println("Tick tick tick...");
5     }

      public static void main (String [] args) {
            SimpleAnimation sa = new SimpleAnimation();

      }
}
```

See next page for a breakdown of each line of code and it’s meaning.

Line 1. Declare a `Timer` variable and set the delay in milliseconds (1/2 a second). Pass it the current object so that it knows which object to call back.

Line 2. Start the timer running. This is being done in the constructor so that it begins right when the class is instantiated.

Lines 3, 4 & 5. Create an `actionPerformed` method to get called when the timer goes off. If you were trying to do animation with graphics, you would make sure you modify some values and then call `repaint()` in this method.

A Simple Background Image

It is nice to have a background in your window other than a solid color. To put an image as your background, we use a drawing method in the Grapics2D package.

```
import java.awt.*;
import javax.swing.*;

public class BackgroundPanel extends JPanel {

1     private Image bg;
2     private int bg_width;
3     private int bg_height;

      public BackgroundPanel () {
4           ImageIcon i = new ImageIcon("src/background.jpg");
5           bg = i.getImage();
6           bg_width = bg.getWidth(null);
7           bg_height = bg.getHeight(null);
      }

8     public int getWidth () {
9           return bg_width;
10    }

11    public int getHeight () {
12          return bg_height;
13    }

14    public void paintComponent(Graphics g) {
15          super.paintComponent(g);
16          Graphics2D g2d = (Graphics2D) g;
17          g2d.drawImage(bg, 0, 0, null);
18    }

      public void setupWindow(BackgroundPanel p) {
19          frame.setSize(p.getWidth(), p.getHeight());
            frame.setLocationRelativeTo(null);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.add(p);
            frame.setVisible(true);
      }

      public static void main(String[] args) {
            JFrame frame = new JFrame("Background");
            BackgroundPanel p = new BackgroundPanel();
            p.setupWindow(p);
      }

}
```

See next page for a breakdown of each line of code and it’s meaning.

Line 1. Declare a variable of type `Image` that will hold our background once we read it in from a file.

Lines 2 & 3. Declare variables to hold the width and height of our background. We will get these values programmatically once we read in our image file.

Lines 4 & 5. In the constructor of the `Background` class read the image in from disk. Note that for Eclipse you must state `src/` before the image. If you are going to run this from the console, do not include `src/` before your image name.

Lines 6 & 7. Once we read in the background, get the width and height of the image so that we can use it when we create the frame.

Lines 8 – 13. Create *getters* to get the value of the background image width and height. We will call these from our `main` method when creating the size of the `frame`.

Line 14. Declare our `paintComponent` method. This will get called when we repaint the screen and when it initially gets painted the first time.

Line 15. Call the super class `paintComponent` to let it do whatever needs to be done.

Line 16. To use methods in the `Graphics2D` class (that extends the `Graphics` class and has a lot of really useful drawing methods) we must type cast the variable `g` from `Graphics` type to `Graphics2D` type.

Line 17. Draw the background image. The 0, 0 are coordinates to draw it starting in the upper left corner.

Line 19. When setting the size of the `JFrame`, make a call to our *getters* to find out the width and height of our image. This way the image will fill the entire window.

A Moving Background

For some games, it is useful to have a background that continually scrolls from left to right or right to left. This is done by integrating a timer into our background panel and redrawing the image 1 pixel different each time, giving it the illusion that it is moving. The tricky part is to have it continually move, requiring us to draw 2 of the same image *connected* together.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MovingBackgroundPanel extends JPanel implements ActionListener {

        private Image bg;           // holds the background image
        private int bg_width;       // width of the background
        private int bg_height;      // height of the background
        private int bg_x=0;         // x location of where to draw background
        private int speed = 8;      // speed to set the timer
        private int direction = -1; // -1 is left, 1 is right...

        public MovingBackgroundPanel () {
                ImageIcon i = new ImageIcon("src/background.jpg");
                bg = i.getImage();
                bg_width = bg.getWidth(null);
                bg_height = bg.getHeight(null);

                // start the timer and set the speed
                Timer tm = new Timer(speed, this);
                tm.start();
        }

        public int getWidth () {
                return bg_width;
        }

        public int getHeight () {
                return bg_height;
        }

        public void actionPerformed (ActionEvent e) {
                // When the timer goes off, we need to scroll the background
                // Once we scroll through the whole width, reset the location back to 0
                if (Math.abs(bg_x) == bg_width)
                        bg_x = 0;
                else
                        bg_x += direction;
                // Force a repaint with the new x location
                repaint();
        }

        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2d = (Graphics2D) g;

                // Draw the image once in the new location,
                // but some of the window will be empty
                g2d.drawImage(bg, bg_x, 0, null);

                // Now draw the background a second time to fill
                //the empty part of the window
                if (direction == -1)
                        g2d.drawImage(bg, bg_width+bg_x, 0, null);
                else
                        g2d.drawImage(bg, bg_x - bg_width, 0, null);
        }
```

```
	public void setupWindow (MovingBackgroundPanel p) {
		frame.setSize(p.getWidth(), p.getHeight());
		frame.setLocationRelativeTo(null);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.add(p);
		frame.setVisible(true);
		panel.requestFocusInWindow();

	}

	public static void main(String[] args) {
	JFrame frame = new JFrame("Background");
	MovingBackgroundPanel p = new MovingBackgroundPanel();
	p.setupWindow(p);

	}
}
```

This code has inline comments and is very similar to the last two lessons. It integrates a timer into the `JPanel`. Every time the timer goes off, it adjusts the `x` coordinate depending on whether it is scrolling left or right. The variable `direction` is positive 1 to scroll right and -1 to scroll left. It then forces a repaint.

The repaint (`paintComponent` method) is slightly tricky as it first paints the image at the new location. However, it must fill the missing part of the window where this image is not present. To do this, it draws the second image to connect to it on the left (if scrolling right) or on the right (if scrolling left). This fills the missing part of the window.

Adding a Menu

A menu is another component that can be added to your window. Most applications have menus if you take a look at the common applications that you use.

First the code, and then I will explain the lines individually.

```
1   import java.awt.event.*;
2   import javax.swing.*;

3   public class FirstMenu implements ActionListener {

4      public static void main(String[] args) {
5            FirstMenu fm = new FirstMenu();
6            fm.setup();
       }

7      public void actionPerformed(ActionEvent e) {
8            System.out.println("Hi there");
9            System.exit(0);
       }

10     public void setup () {
11           JFrame f = new JFrame("Window");
12           JMenuBar menubar = new JMenuBar();
13           JMenu file = new JMenu("File");

14           JMenuItem eMenuItem = new JMenuItem("Exit");
15           eMenuItem.addActionListener(this);

16           file.add(eMenuItem);
17           menubar.add(file);

18           f.setJMenuBar(menubar);

19           f.setTitle("Simple menu");
20           f.setSize(300, 200);
21           f.setLocationRelativeTo(null);
22           f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23           f.setVisible(true);
24     }
25  }
```

See next page for a breakdown of each line of code and it's meaning.

Lines 1 & 2. We import all of the SWING and AWT classes so that they can be used.

Line 3. We "implement" an `ActionListener` which is a way that our program can get called from actions that happen in the Windowing environment.

Lines 4, 5, & 6. Create a new instance of our class and call the `setup` method to do all the work.

Line 7, 8, & 9. This is a very simple `ActionListener`. This method will get called when the user chooses any of the menu items. We only have one menu item, so we can assume it is the "exit" menu that we create.

Line 11. Create a new `JFrame` component. A `JFrame` is a top level container for windows. The basic purpose of the container is to hold components for your application.

Line 12. Create a menubar item since we want to place one in our window.

Line 13. Create a menu item that we will place into our menubar. Normally we would have more than one menu items.

Line 14. A menu object consists of menu items. A menu item is created with the `JMenuItem` class. The string is the menu item that will appear on your window.

Line 15. We add our ActionListener so that whenever the menu item is chosen, it will call our `ActionListener` (above).

Line 16. We add our menu item (exit) to our menu.

Line 17. We add our menu to our menu bar.

Line 18. We put our menubar into the frame.

Lines 19 to 23. Discussed in previous examples.

Adding Sound

Layout Managers – Grid Layout

Laying out your window with buttons and textboxes and images is challenging, but luckily Java includes layout managers to help you. The first layout style we used was a FlowLayout which basically placed everything in the window flowing in the order you added it. This is good for simple windows or parts of a window, but often times we need more functionality.

A grid layout allows you to break your window into regions that are laid out like graph paper. Here is an example with six buttons. The program is on the back of this page so you can see it in it's entirety.

The major components of using the grid layout is in line:

Line 23 Establish that the layout being used in the JPanel is a GridLayout. Specify that the rows and columns in the grid are 2 (rows) by 3 (columns). If you want to have as many rows or columns as necessary you can use 0 for one of the parameters.

There are two constructors for the GridLayout. The one above takes rows and columns as parameters. In addition you can pass rows, columns, as well as a horizontal gap and vertical gap. These gaps put buffers between the rows and columns so that objects are not pushed up touching each other (if desired).

From this point on, when you add new objects to the JPanel, they will get added from left to right, top to bottom.

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyWindow implements ActionListener {

    private JButton b1, b2, b3, b4, b5, b6;

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == b1)
            System.out.println("Button 1 was pressed");
        else if (e.getSource() == b2)
            System.out.println("Button 2 was pressed");
        else if (e.getSource() == b3)
            System.out.println("Button 3 was pressed");
        else if (e.getSource() == b4)
            System.out.println("Button 4 was pressed");
        else if (e.getSource() == b5)
            System.out.println("Button 5 was pressed");
        else if (e.getSource() == b6)
            System.exit(0);
    }

    public void setupWindow() {
        JFrame f = new JFrame("Displaying a grid layout");
        JPanel p = new JPanel();

        p.setLayout(new GridLayout(2,3));

        b1 = new JButton("Button 1");
        b1.addActionListener(this);
        p.add(b1);

        b2 = new JButton("Button 2");
        b2.addActionListener(this);
        p.add(b2);

        b3 = new JButton("Button 3");
        b3.addActionListener(this);
        p.add(b3);

        b4 = new JButton("Button 4");
        b4.addActionListener(this);
        p.add(b4);

        b5 = new JButton("Button 5");
        b5.addActionListener(this);
        p.add(b5);

        b6 = new JButton("EXIT");
        b6.addActionListener(this);
        p.add(b6);

        f.getContentPane().add(p, "Center");

        f.setSize(300, 300);
        f.setLocationRelativeTo(null);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }


    public static void main(String[] args)  {
        MyWindow fw = new MyWindow();
        fw.setupWindow();
    }
}
```

More Layout Managers – Card Layout

Laying out your window with buttons and textboxes and images is challenging, but luckily Java includes layout managers to help you. The first layout style we used was a FlowLayout which basically placed everything in the window flowing in the order you added it. This is good for simple windows or parts of a window, but often times we need more functionality.

The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. It may also be controlled based on levels in a game to be able to swap out different JPanels.

The code is fully commented. The tricky part of this code is to create a high level JPanel (called cards in this program) and then create multiple JPanels that will be attached to it. In this program I called them card1 and card2. You add the sub JPanels to the parent JPanel and then switch between them using the code:

```
    CardLayout cl = (CardLayout)(cards.getLayout());
    cl.show(cards, "C1");
```

I named the card1 and card2 JPanels as C1 and C2 in the code when I added them to the parent JPanel. This makes it easier to show the correct card.

When you create the parent JPanel, you state that the layout being used is CardLayout as seen in this line:

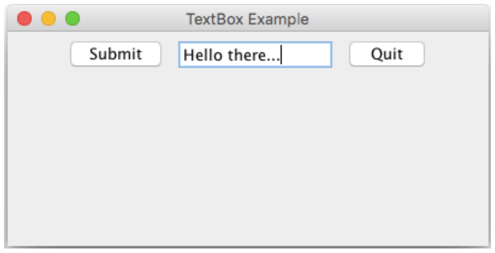
```
    cards = new JPanel(new CardLayout());
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CardLayoutDemo implements ActionListener {

        JPanel cards;    //The top level container to hold all of the 'cards'

        public static void main(String[] args) {
                CardLayoutDemo cl = new CardLayoutDemo();
                cl.setup();
        }

        public void actionPerformed(ActionEvent e) {
                //Check to see what menu item the user chose
                if (e.getActionCommand().equals("Go to Card1")) {
                    CardLayout cl = (CardLayout)(cards.getLayout());
                    cl.show(cards, "C1");
                }
                else if (e.getActionCommand().equals("Go to Card2")) {
                    CardLayout cl = (CardLayout)(cards.getLayout());
                    cl.show(cards, "C2");
                }
                // Assume they chose exit if no others...
                else
                        System.exit(0);
        }

        public void setup () {
                JFrame f = new JFrame("Card Layout");
                JPanel card1 = new JPanel();
                JPanel card2 = new JPanel();
                JMenuBar menubar = new JMenuBar();
                JMenu file = new JMenu("File");

                // Declare the 3 menu items.
                JMenuItem eMenuItem = new JMenuItem("Exit");
                JMenuItem eMenuItem2 = new JMenuItem("Go to Card1");
                JMenuItem eMenuItem3 = new JMenuItem("Go to Card2");

                eMenuItem.addActionListener(this);
                eMenuItem2.addActionListener(this);
                eMenuItem3.addActionListener(this);

                // Add the 3 menu items to the file menu
                file.add(eMenuItem2);
                file.add(eMenuItem3);
                file.add(eMenuItem);   // Exit Menu item

                // Add the file menu to the menubar
                menubar.add(file);

                f.setJMenuBar(menubar);

                // Create the high level container for the cards
                cards = new JPanel(new CardLayout());
                //Create a button, one for each card so we can see the difference
                JButton Card1Button = new JButton("This is Card1");
                card1.add(Card1Button);
                JButton Card2Button = new JButton("This is Card2");
                card2.add(Card2Button);

                // Add the cards to the high level card container
                cards.add(card1, "C1");
                cards.add(card2, "C2");

                // Add the card layout to the frame.
                f.add(cards, BorderLayout.CENTER);

                f.setTitle("Simple menu");
                f.setSize(500, 300);
                f.setLocationRelativeTo(null);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setVisible(true);
        }
}
```

Adding a Text Field

It is very common to need input from the user in your Windows based programs. To do this, we use a `JTextField`. Normally we will put a Submit button or something to tell the program that the user has entered text into the fields.

The example provided on the next page creates a `JFrame` containing a `JPanel`. Within the `JPanel` there are three objects: two `JButtons` (submit and exit) as well as a `JTextField`. When the submit button is clicked, the `actionPerformed` method is invoked. Within this method we check to see which button was clicked by using the `getSource` method within the `ActionEvent`. If the Submit button was clicked we utilize the `getText` method within the `JTextField` to see the text that was in the box.

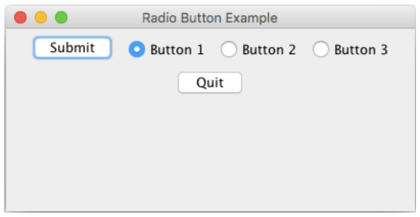


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextFieldExample implements ActionListener {

      JTextField textField;
      JButton quitButton;
      JButton submitButton;
      JPanel myPanel;
      JFrame myFrame;

      public void actionPerformed(ActionEvent e) {
            if (e.getSource() == quitButton)
                  System.exit(0);
            else if (e.getSource() == submitButton)
                  System.out.println("Text: " + textField.getText());

      }

      private void setupWindow() {

            myFrame = new JFrame ();
            myPanel = new JPanel();
            textField = new JTextField(10);

            myPanel.setLayout(new FlowLayout());

            quitButton = new JButton("Quit");
            quitButton.addActionListener(this);

            submitButton = new JButton("Submit");
            submitButton.addActionListener(this);

            myPanel.add(submitButton);
            myPanel.add(textField);
            myPanel.add(quitButton);

            myFrame.add(myPanel, "Center");

            myFrame.setTitle("TextBox Example");
            myFrame.setSize(400, 200);
            myFrame.setLocationRelativeTo(null);
            myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            myFrame.setVisible(true);
      }

      public static void main(String[] args) {

            TextFieldExample ex = new TextFieldExample();
            ex.setupWindow();

      }
}
```

Adding a Radio Button

Another commonly used input method is the use of a radio button. A radio button is a group of buttons that only allow you to select one choice. This is similar to a multiple choice test where only one answer is valid.

The example provided on the next page creates a `JFrame` containing a `JPanel`. Within the `JPanel` there are five objects: two `JButtons` (submit and exit) as well as three `JRadioButtons` that are put into a `ButtonGroup`. When the submit button is clicked, the `actionPerformed` method is invoked. Within this method we check to see which button was clicked by using the `getSource` method within the `ActionEvent`. If the Submit button was clicked we first determine which of the `JRadioButtons` is selected in the `ButtonGroup`, and then we use the `getActionCommand` that we had setup for the `JRadioButtons` during initialization of the buttons.

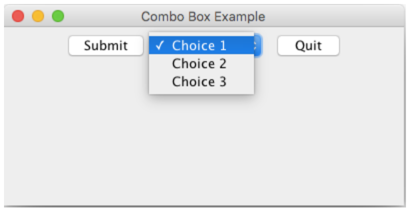


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButtonExample implements ActionListener {
        ButtonGroup group;
        JButton quitButton;
        JButton submitButton;
        JPanel myPanel;
        JFrame myFrame;


        public void actionPerformed(ActionEvent e) {
                if (e.getSource() == quitButton)
                        System.exit(0);
                else if (e.getSource() == submitButton)
                        System.out.println("Selected Radio Button: " +
                                        group.getSelection().getActionCommand());
        }


        private void setupWindow() {
                myFrame = new JFrame ();
                myPanel = new JPanel();
                group = new ButtonGroup();

                JRadioButton button1 = new JRadioButton("Button 1");
                button1.setActionCommand("button1");
                button1.setSelected(true);    // Set one button as selected by default.
                JRadioButton button2 = new JRadioButton("Button 2");
                button2.setActionCommand("button2");
                JRadioButton button3 = new JRadioButton("Button 3");
                button3.setActionCommand("button3");

                group.add(button1);
                group.add(button2);
                group.add(button3);

                myPanel.setLayout(new FlowLayout());

                quitButton = new JButton("Quit");
                quitButton.addActionListener(this);

                submitButton = new JButton("Submit");
                submitButton.addActionListener(this);

                myPanel.add(submitButton);
                myPanel.add(button1);
                myPanel.add(button2);
                myPanel.add(button3);
                myPanel.add(quitButton);

                myFrame.add(myPanel, "Center");

                myFrame.setTitle("Radio Button Example");
                myFrame.setSize(400, 200);
                myFrame.setLocationRelativeTo(null);
                myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                myFrame.setVisible(true);
        }


        public static void main(String[] args) {
                RadioButtonExample ex = new RadioButtonExample();
                ex.setupWindow();

        }
}
```

Adding a Combo Box

Another commonly used input method is the use of a combo box (often called a dropdown list).

The example provided on the next page creates a `JFrame` containing a `JPanel`. Within the `JPanel` there are three objects: two `JButtons` (submit and exit) as well as a `JComboBox`.When the submit button is clicked, the `actionPerformed` method is invoked. Within this method we check to see which selection was made in the `JComboBox` by using the `getSelectedItem` method.



```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBoxExample implements ActionListener {

      JComboBox<String> cb;
      JButton quitButton;
      JButton submitButton;
      JPanel myPanel;
      JFrame myFrame;

      public void actionPerformed(ActionEvent e) {
            if (e.getSource() == quitButton)
                  System.exit(0);
            else if (e.getSource() == submitButton)
                  System.out.println("Selected Choice in Combo Box: "+
                                      cb.getSelectedItem());

      }

      private void setupWindow() {

            myFrame = new JFrame ();
            myPanel = new JPanel();

            String[] choices = {"Choice 1", "Choice 2", "Choice 3"};
            cb = new JComboBox<String>(choices);

            myPanel.setLayout(new FlowLayout());

            quitButton = new JButton("Quit");
            quitButton.addActionListener(this);

            submitButton = new JButton("Submit");
            submitButton.addActionListener(this);

            myPanel.add(submitButton);
            myPanel.add(cb);
            myPanel.add(quitButton);

            myFrame.add(myPanel, "Center");

            myFrame.setTitle("Combo Box Example");
            myFrame.setSize(400, 200);
            myFrame.setLocationRelativeTo(null);
            myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            myFrame.setVisible(true);
      }

      public static void main(String[] args) {

            ComboBoxExample ex = new ComboBoxExample();
            ex.setupWindow();

      }
}
```