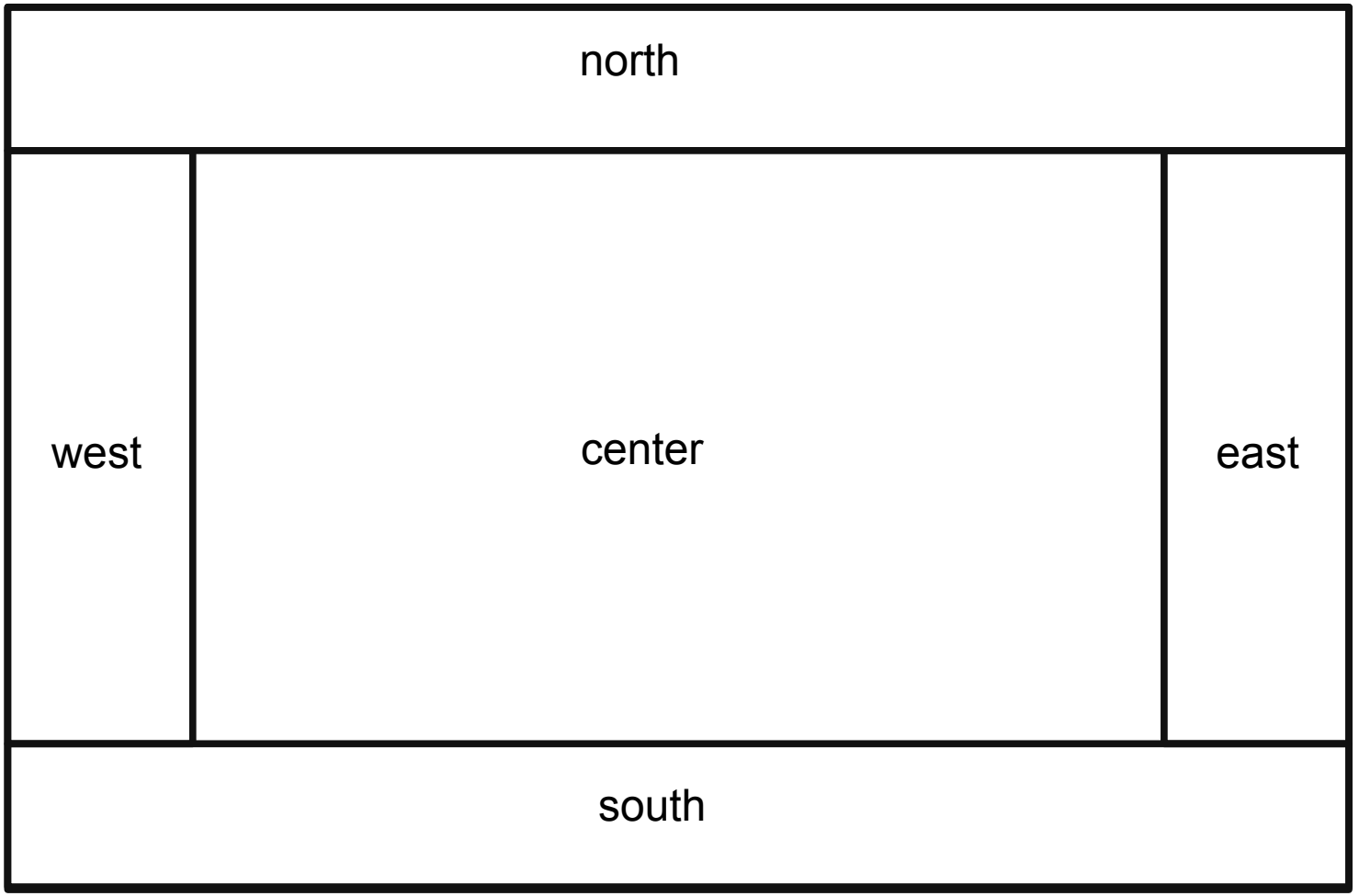
- Swing is the collection of classes used to create windows (not MS Windows, but Graphical User Interface windows) based Java applications
- Swing is built on top of the Abstract Windowing Toolkit (AWT)
  - AWT components are platform specific
    - Components are “heavyweight” components
  - Swing components are platform independent
    - Components are “lighweight” all Java components
- Note: Swing is not thread safe
  - From Java documentation for Jbutton
    - “Warning: Swing is not thread safe. For more information see Swing's Threading Policy.”

- Swing components (a small subset)
  - If you pull up the Java documentation, go to the J's, you will see many of the Swing components starting with J
    - There are non Swing classes that also start with J
  - JComponent
    - Base class for the Swing components, except for the top level containers
  - JFrame
    - Basically a Swing window
  - JButton
    - Swing button
  - JPanel
    - A swing container to put other components in
  - JCheckBox
    - A Swing check box component

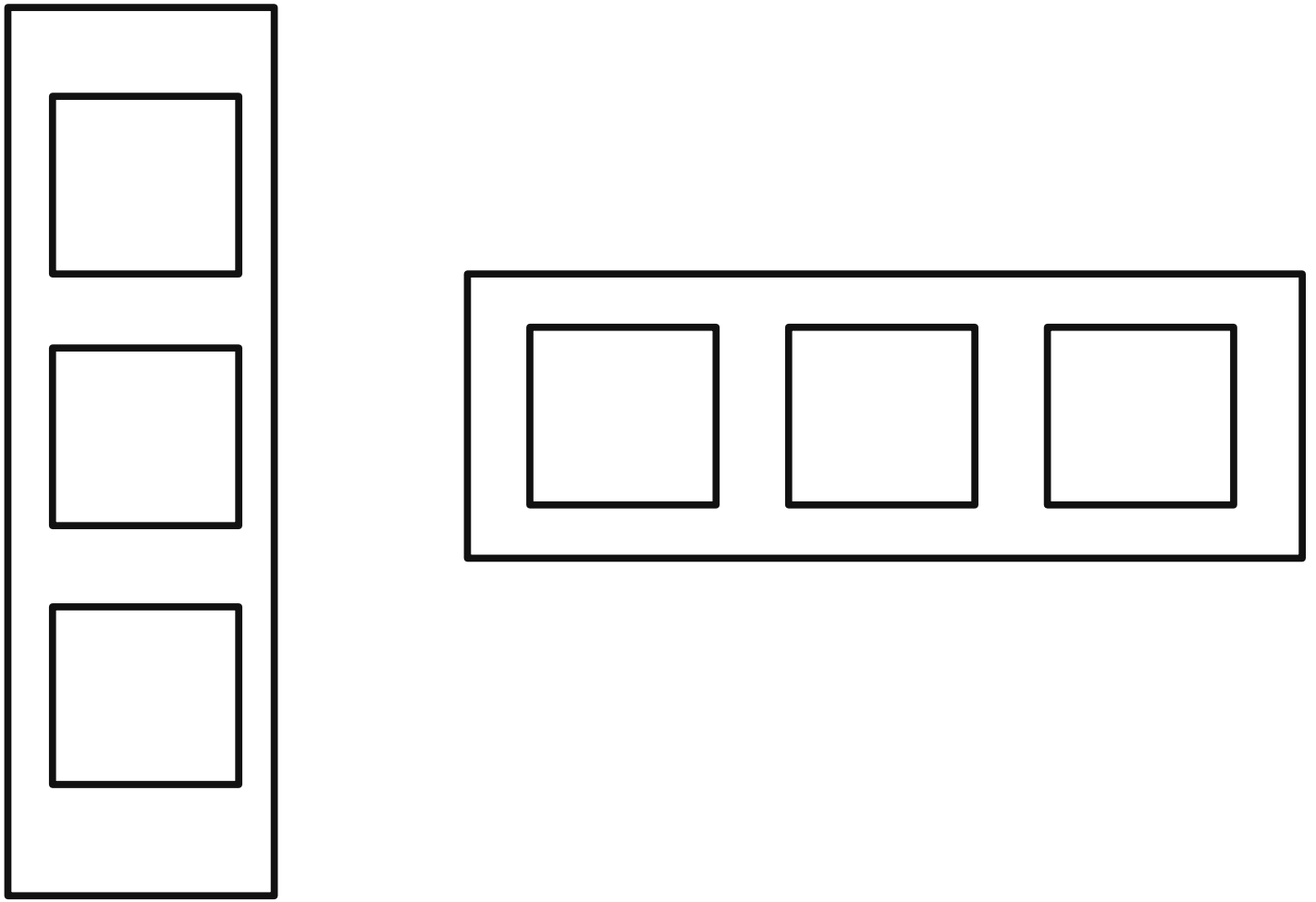
- Swing components (cont)
  - JLabel
    - Swing label component
  - JMenu
    - Swing menu
  - JMenuBar
    - Swing menu bar
  - JMenuItem
    - Swing menu item
  - JScrollPane
    - Swing scroll pane

- JPanel
  - These are containers to hold components
  - Typically
    - Create a JPanel
    - Specify a layout manager for the panel
    - Add components to the panel
    - Add the panel to a container
- Layout managers handle how components are arranged within a container
  - BorderLayout
  - BoxLayout
  - GridLayout
  - GridBagLayout
  - And many more

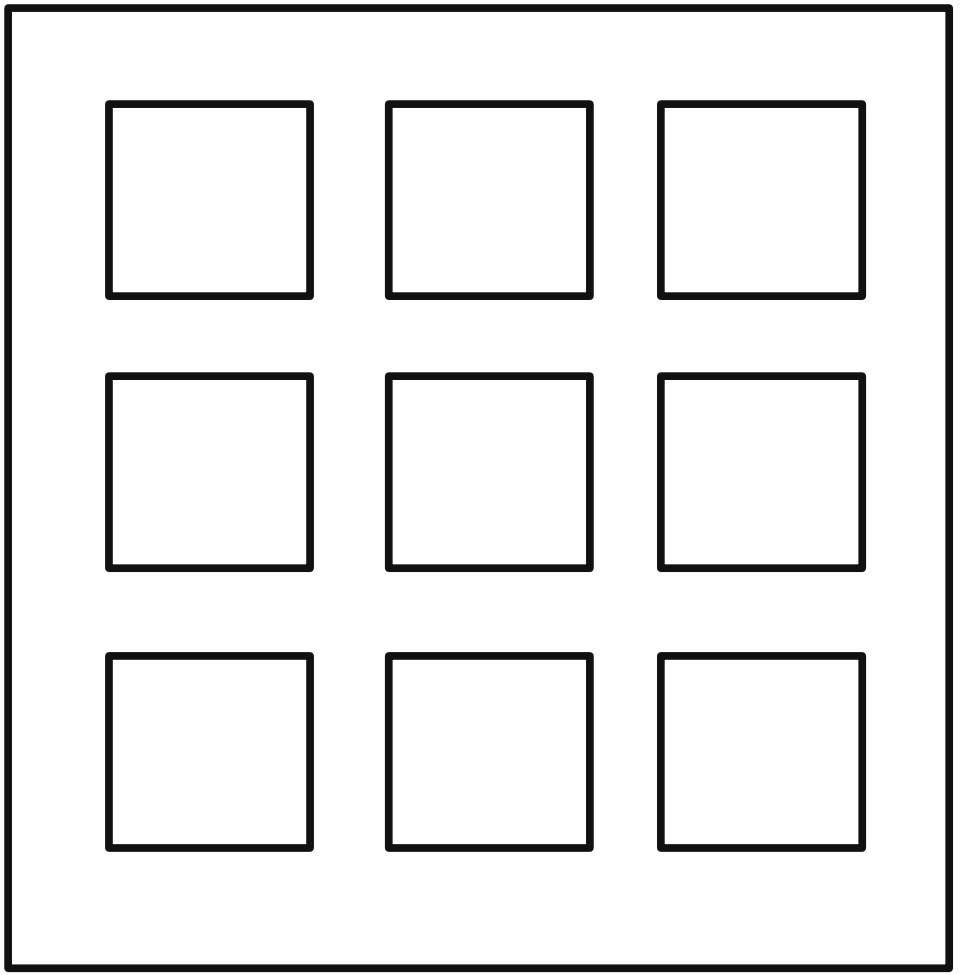
- BorderLayout
  - Lays out up to five components in a container as below



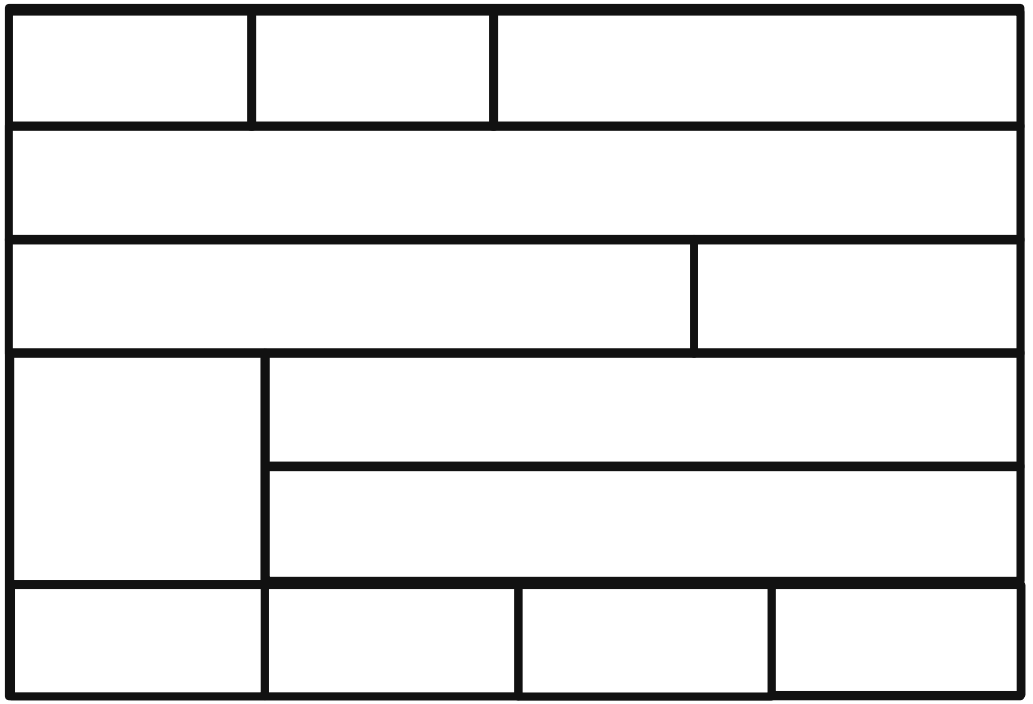
- BoxLayout
  - Lays out components vertically or horizontally



- GridLayout
  - Lays out components in a grid



- GridBagLayout
  - Lays out components vertically and horizontally



- JFrame
  - This is the “window”
  - Typically has one or two components
    - MenuBar
      - Assigned via setMenuBar()
      - At top of the window
      - Can put the MenuBar in the component that is assigned as the ContentPane and leave this “empty”
    - ContentPane
      - Assigned via setContentPane()
      - Whatever component is defined as the ContentPane typically has most of the functionality
        - Non MenuBar functionality
  - Can set the minimum and preferred sizes
    - Can't resize smaller than the minimum size
    - The preferred size should be the size that the window initially starts at when it shows up on the screen

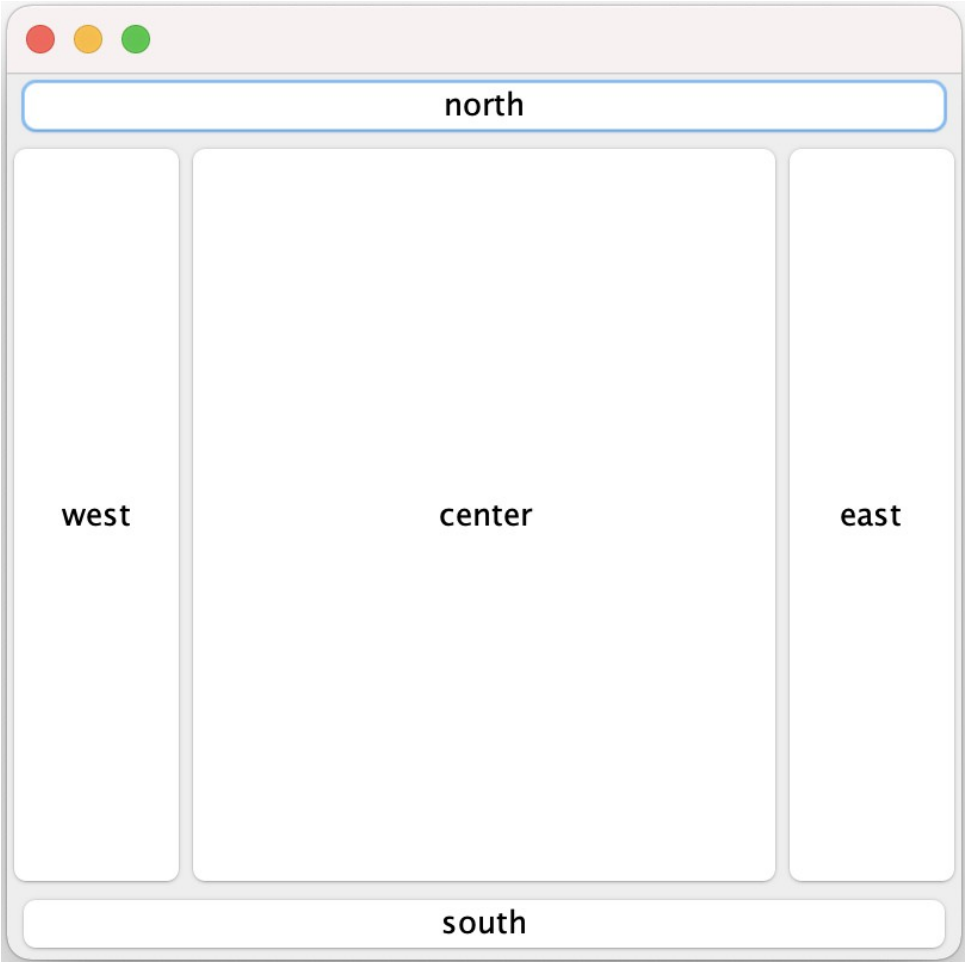
- Simple example (example1a.java)
  - We want to put a window on the screen, and then put five buttons on the window
    - First we need a JFrame, which will be the window
    - Once we have created the JFrame
      - Set its size
      - Have the program exit if the JFrame is closed
      - Set the JFrame's content pane
      - Validate it
      - Make it visible
    - In the JFrame we will have a JPanel, which will contain the five buttons
      - Once we create the JPanel, we need to specify how to layout the components that we add to it
        - We use a BorderLayout, which puts one component on the east, west, north, south borders and one in the center
        - There are many ways to layout a JPanel
      - We create the five JButtons and add them to the JPanel
        - One at each of the “five” borders

- Simple example (cont)
  - Once we have the JFrame, JPanel, and five JButtons created
    - We want to know when the JButtons are pushed
    - We define and add an ActionListener to each of the five buttons
      - See ActionListener interface in Java documentation
    - The action listeners actionPerformed method will be called when the buttons are pushed
- JButton
  - A push button
  - When we create it we can specify the text on the button
  - We can use the addActionListener(ActionListener x) method to specify what class will be used/called when the button is pushed
  - An ActionListener must implement the actionPerformced(ActionEvent e) method

- Simple example (cont)
- For this example, we will create an inner class, ButtonActionListener, that implements the ActionListener interface
  - Since we want to share this ActionListner with all five buttons, we include a JButton field that specifies which button it is for

```
// action listener for the buttons
static class ButtonActionListener implements java.awt.event.ActionListener
{
    // the button associated with the action listener, so that we can
    // share this one class with multiple buttons
    private javax.swing.JButton b;

    ButtonActionListener(javax.swing.JButton b)
    {
        this.b = b;
    }

    public void actionPerformed(java.awt.event.ActionEvent e)
    {
        System.out.println("action performed on " + b.getText() + " button");
    }
}
```

- Simple example (cont)
  - Here's what it looks like
  - If we change the size of the window, the buttons expand to fill the available space
    - West, center, and east buttons expand in north and south directions
    - North, center, and south buttons expand in east and west directions



- Simple example (cont)
- Here's the code that creates the window, sets the size of the window, and specifies to exit the program when the window is closed
- We also create the five buttons, with the text written on the button to be where it will be layed out by the BorderLayout manager

```
// create the frame (or window)
javax.swing.JFrame f = new javax.swing.JFrame();

// set the preferred and minimum sizes of the frame
f.setPreferredSize(new java.awt.Dimension(width, height));
f.setMinimumSize(new java.awt.Dimension(width, height));

// specify that the program will exit when the frame is closed
f.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);

// create the five buttons
javax.swing.JButton westButton = new javax.swing.JButton("west");
javax.swing.JButton eastButton = new javax.swing.JButton("east");
javax.swing.JButton northButton = new javax.swing.JButton("north");
javax.swing.JButton southButton = new javax.swing.JButton("south");
javax.swing.JButton centerButton = new javax.swing.JButton("center");
```

- Simple example (cont)
- Here's the code that creates the JPanel, sets the layout for the panel, and adds the buttons to the panel, specifying where they should be positioned by the BorderLayout manager

```
// create the panel with a border layout
javax.swing.JPanel mainPanel = new javax.swing.JPanel();
mainPanel.setLayout(new java.awt.BorderLayout());

// add the five buttons to the panel
mainPanel.add(centerButton, java.awt.BorderLayout.CENTER);
mainPanel.add(westButton, java.awt.BorderLayout.WEST);
mainPanel.add(eastButton, java.awt.BorderLayout.EAST);
mainPanel.add(northButton, java.awt.BorderLayout.NORTH);
mainPanel.add(southButton, java.awt.BorderLayout.SOUTH);
```

- Simple example (cont)
- Here's the code that creates the ButtonActionListeners and the code that adds the appropriate ButtonActionListener to the action listeners for each of the five buttons

```
// create the action listeners for the buttons
ButtonActionListener centerButtonActionListener = new ButtonActionListener(centerButton);
ButtonActionListener westButtonActionListener = new ButtonActionListener(westButton);
ButtonActionListener eastButtonActionListener = new ButtonActionListener(eastButton);
ButtonActionListener northButtonActionListener = new ButtonActionListener(northButton);
ButtonActionListener southButtonActionListener = new ButtonActionListener(southButton);

// add the action listeners to the buttons
centerButton.addActionListener(centerButtonActionListener);
westButton.addActionListener(westButtonActionListener);
eastButton.addActionListener(eastButtonActionListener);
northButton.addActionListener(northButtonActionListener);
southButton.addActionListener(southButtonActionListener);
```

- Simple example (cont)
- And finally we add the panel to the frame, validate it (layout the components), and set it visible
  - At this point the window will show up on the desktop

```
// add the panel to the window
f.setContentPane(mainPanel);

// validate the window and put it on the screen
f.validate();
f.setVisible(true);
```

- Simple example (example2a.java)
  - Let's get rid of the button on the top of the window, and add a menu bar where it is
- We create a JmenuBar and add it to the JPanel as the north component
- We create two JMenus, one for “File” and one for “Edit” and add them to the MenuBar
- We create two JMenuItems, an “Exit” item to be added to the “File” menu and a “Color” item to be added to the “Edit” menu
  - The “Exit” menu item simply exits the program
  - The “Color” menu item opens a JOptionPane that tells us that this has not been implemented yet

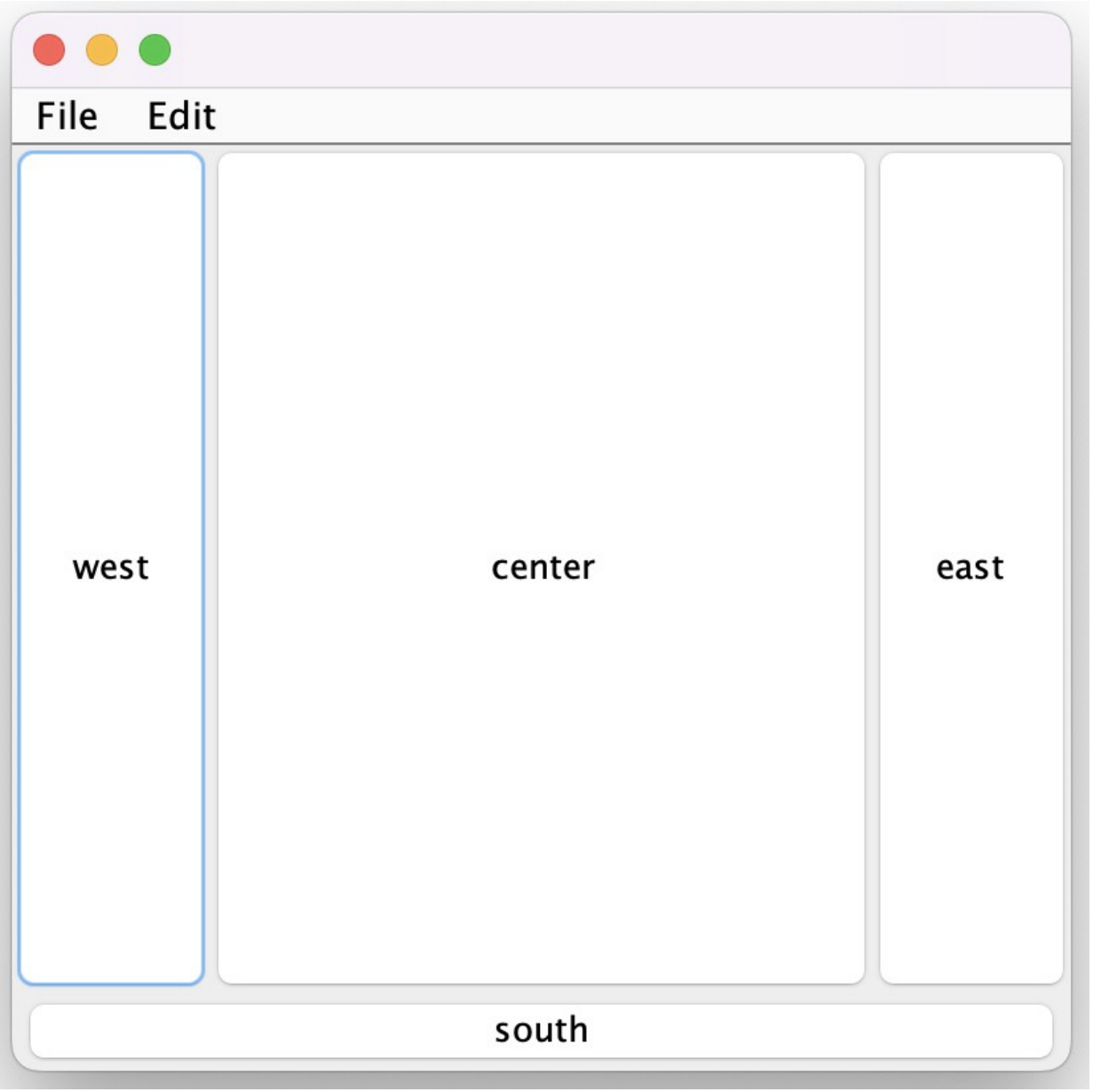
- example2a.java (cont)
  - Create the JMenuBar
  - Create the two JMenus
  - Create the two JMenuItems
  - Add the two Menus to the JMenuBar
  - Add the two JMenuItems to the JMenus

```
// create the menu bar
javax.swing.JMenuBar menuBar = new javax.swing.JMenuBar();

// create the two menus
javax.swing.JMenu fileMenu = new javax.swing.JMenu("File");
javax.swing.JMenu editMenu = new javax.swing.JMenu("Edit");

// create the menu items for the two menus
javax.swing.JMenuItem fileExit = new javax.swing.JMenuItem("Exit");
javax.swing.JMenuItem editColor = new javax.swing.JMenuItem("Color");

// add the two menus to the menu bar
menuBar.add(fileMenu);
menuBar.add(editMenu);

// add the two menu items to the two menus
fileMenu.add(fileExit);
editMenu.add(editColor);
```

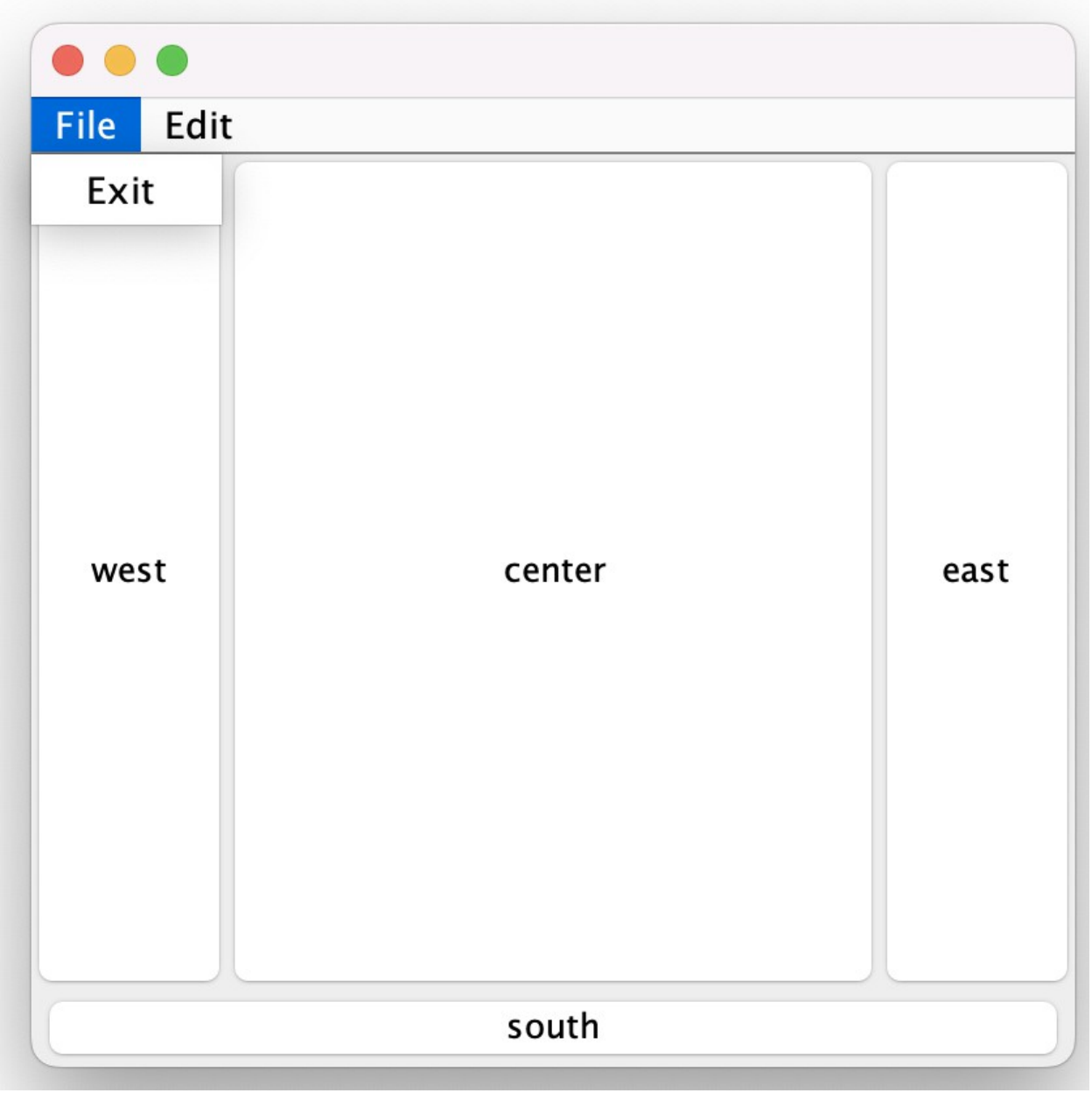
- example2a.java (cont)
  - We create an ActionListener to handle clicks on the JMenuItems
    - The JMenuItem field lets us know which JMenuItem is the source

```
// action listener for the menu items
static class MenuItemActionListener implements java.awt.event.ActionListener
{
    // the menu item associated with the action listener, so that we can
    // share this one class with multiple menu items
    private javax.swing.JMenuItem m;

    MenuItemActionListener(javax.swing.JMenuItem m)
    {
        this.m = m;
    }

    public void actionPerformed(java.awt.event.ActionEvent e)
    {
        System.out.println("action performed on " + m.getText() + " menu item");

        // if exit is selected from the file menu, exit the program
        if( m.getText().toLowerCase().equals("exit") )
        {
            System.exit(0);
        }

        // if color is selected from the edit menu, put a popup on the screen
        // saying something
        if( m.getText().toLowerCase().equals("color") )
        {
            Object[] options = {"OK"};
            javax.swing.JOptionPane.showOptionDialog(null, "This is unimplemented,\nclick OK to continue",
                    "Warning", javax.swing.JOptionPane.DEFAULT_OPTION,
                    javax.swing.JOptionPane.WARNING_MESSAGE, null, options, options[0]);
        }
    }
}
```

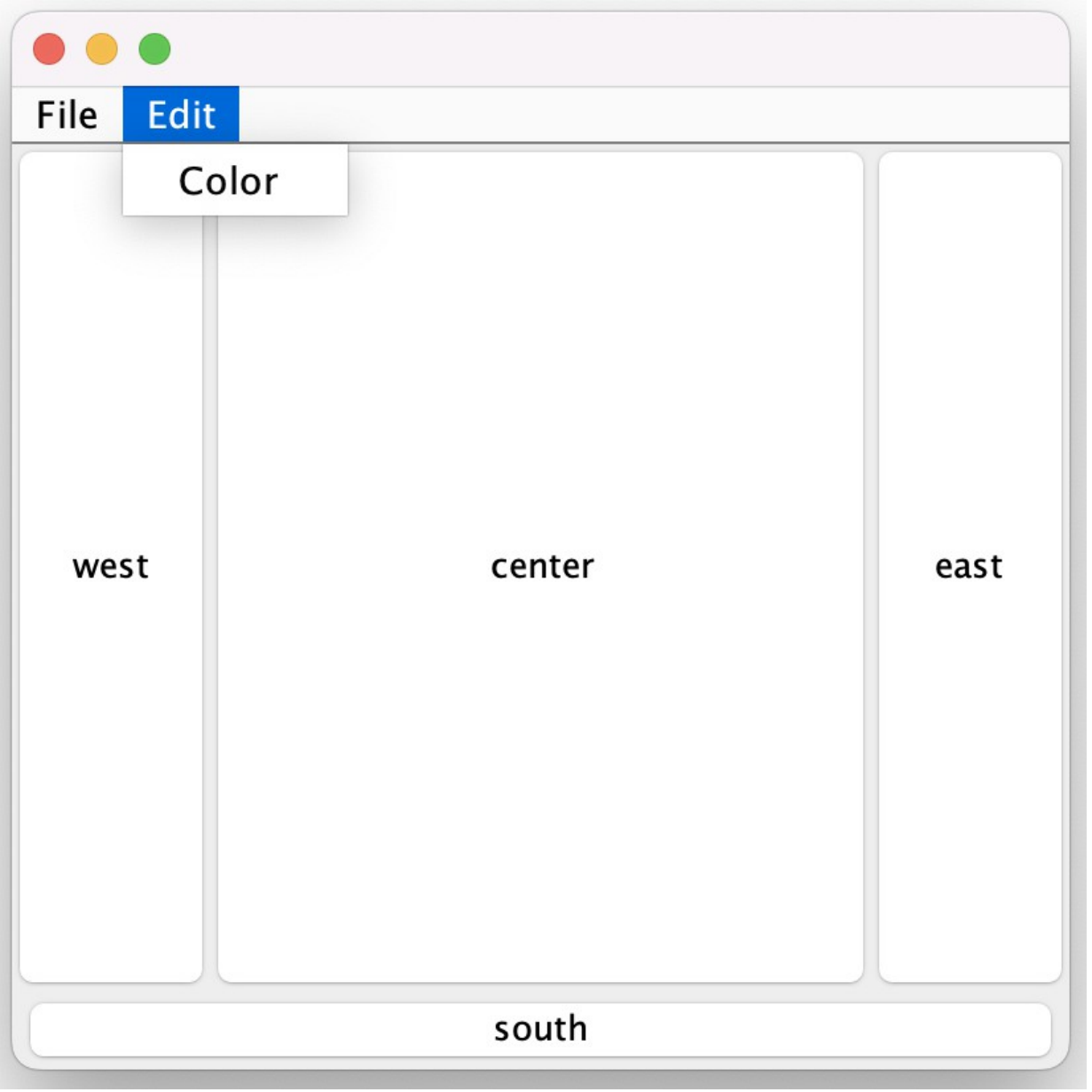
- example2a.java (cont)
  - Here's the new window



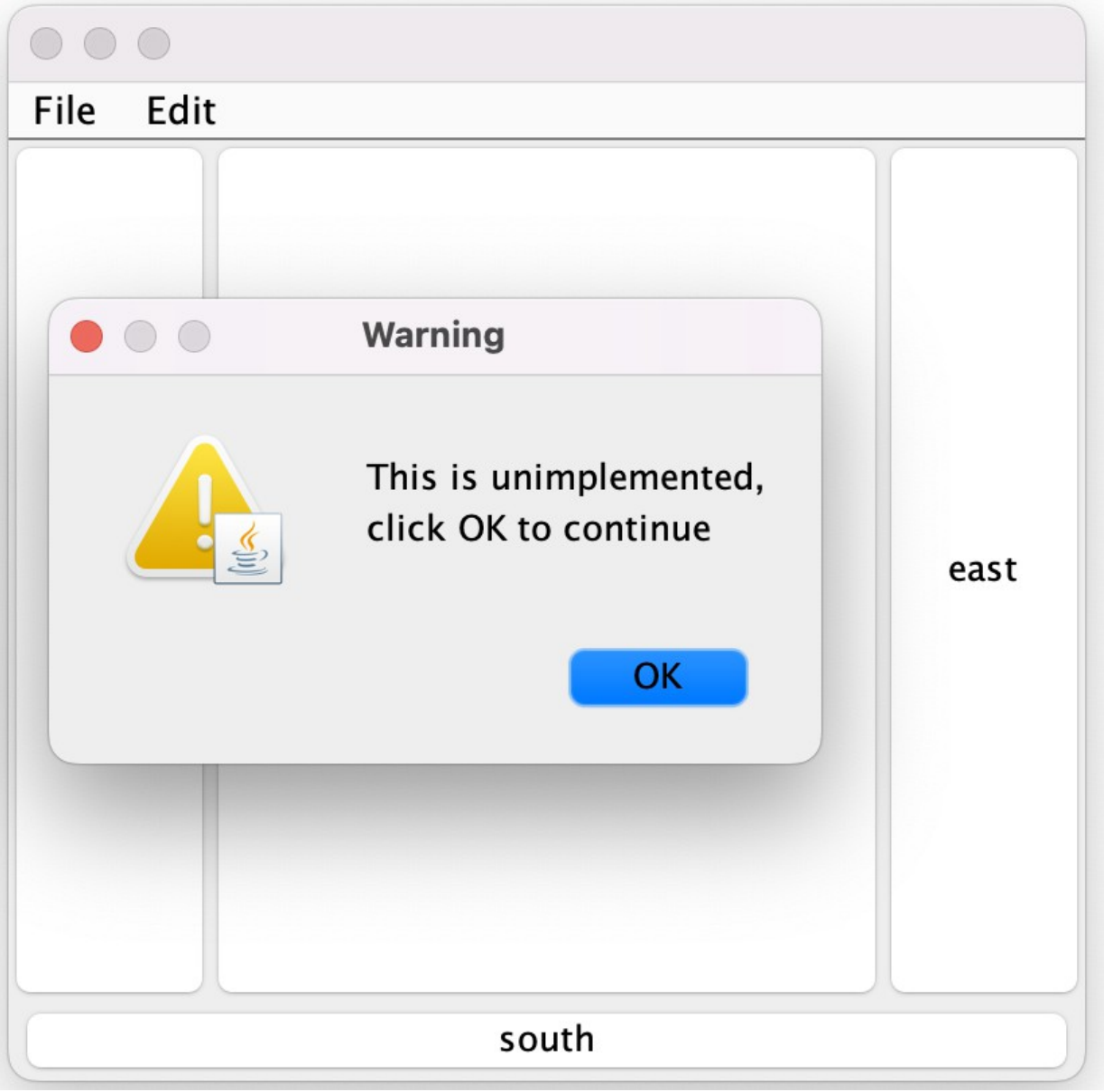
- example2a.java (cont)
  - Here's the new window with “File” selected



- example2a.java (cont)
  - Here's the new window with “Edit” selected



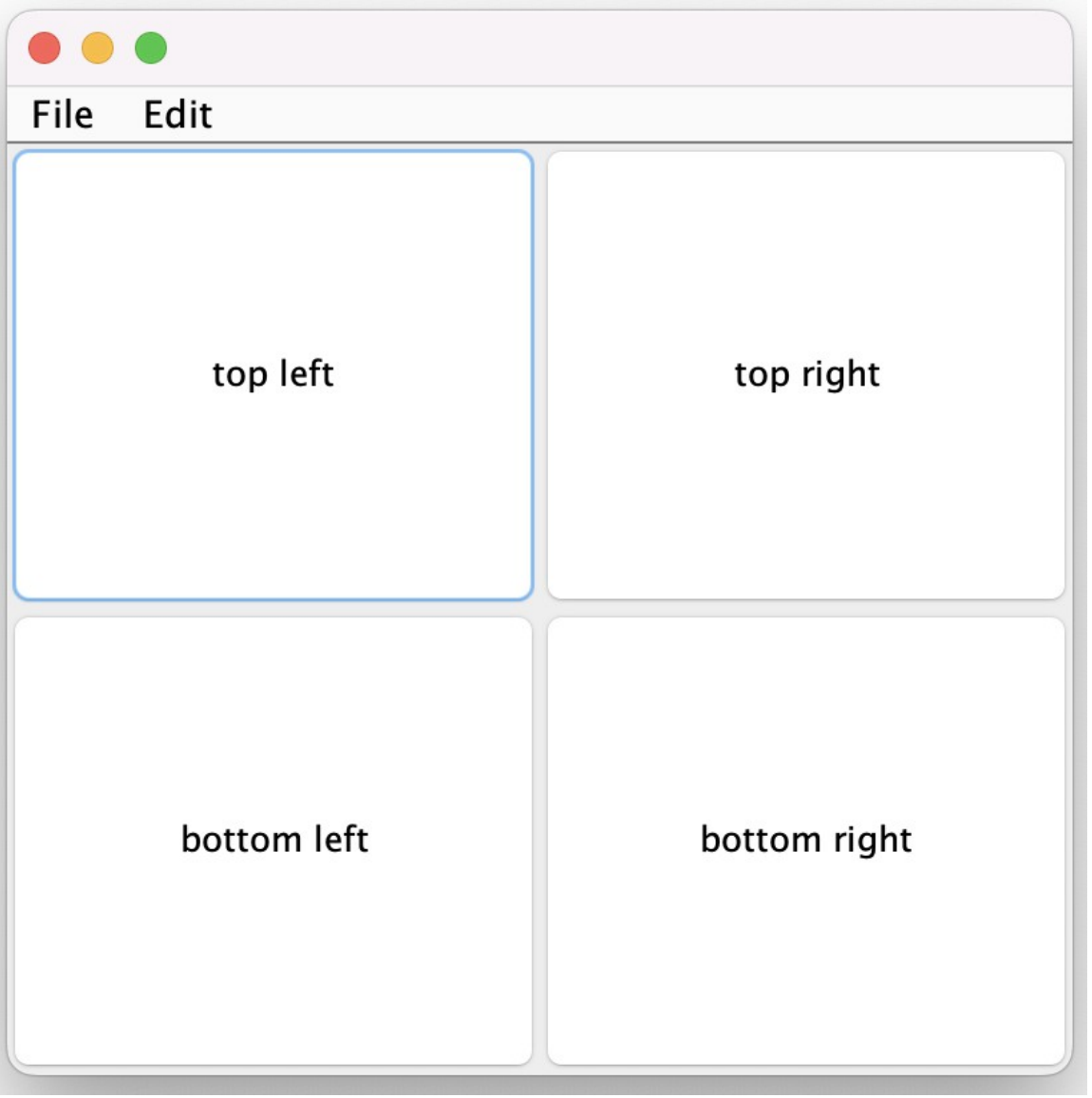
- example2a.java (cont)
  - Here's the new window after “Color” is selected and thje JOptionPane is visible



- Simple example (example3a.java)
  - Let's put the four buttons in the center in a 2 x 2 grid
  - The GridLayout has the components fill the space, so the buttons are taking up the available space
- Create the JPanel, set the layout to a 2 x 2 GridLayout, and then add the four buttons to the JPanel
- Put the MenuBar at the north location and the panel of four buttons in the center

```
// create the panel to hold the four buttons in a 2 x 2 grid and add the buttons
javax.swing.JPanel buttonPanel = new javax.swing.JPanel();
buttonPanel.setLayout(new java.awt.GridLayout(2, 2));
buttonPanel.add(topLeftButton);
buttonPanel.add(topRightButton);
buttonPanel.add(bottomLeftButton);
buttonPanel.add(bottomRightButton);

// create the panel
javax.swing.JPanel mainPanel = new javax.swing.JPanel();
mainPanel.setLayout(new java.awt.BorderLayout());
mainPanel.add(menuBar, java.awt.BorderLayout.NORTH);
mainPanel.add(buttonPanel, java.awt.BorderLayout.CENTER);
```

- example3a.java (cont)
  - Here's what the new window looks like
  - If we make the window larger, the buttons fill the available space

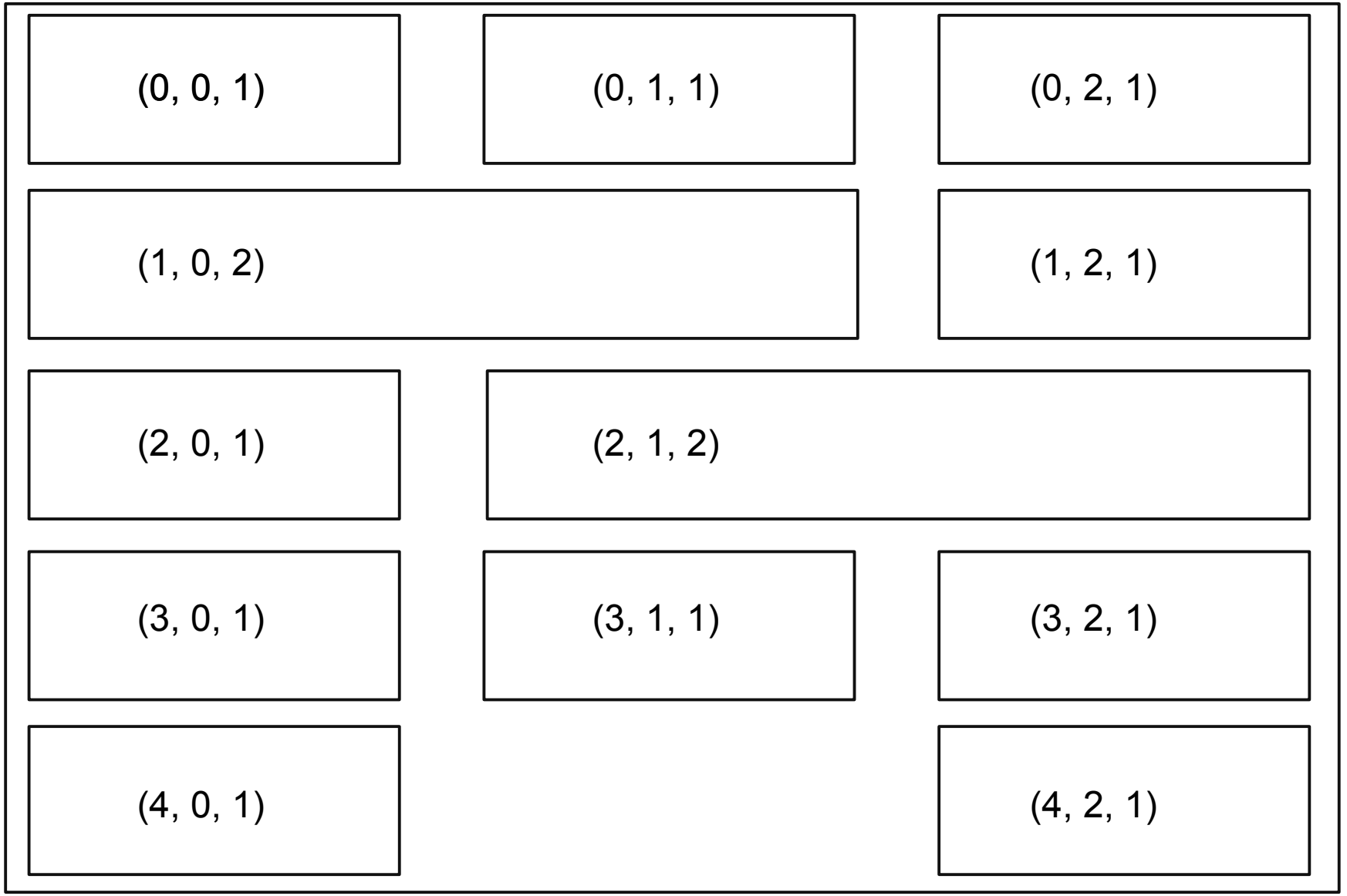


- We are going to use the GridBagLayout for lab 12 and program 8
- Conceptually the GridBagLayout wants to lay out the components, in our case buttons and labels, in a grid
  - We can think of the grid as having rows and columns of components
  - Some components can have a height that is larger than a single row
  - Some components can have a width greater than a single column
- Once we instantiate the GridBagLayout, we instantiate a GridBagConstraints object to specify constraints for the components that are layed out by the GridBagLayout

- The GridBagConstraints tell the GridBagLayout where to position the components and how many grid cells to take up in both the x and y directions
- I got by with using the following constraints in example4b.java
  - Set weightx and weighty to 1
    - This basically tells the GridBagLayout to spread out the components instead of clumping them together in the middle of the container
    - We only need to set these once
  - Set the gridx and gridy
    - For each component, tell it where to put the left corner of the component in the grid (upper left of the container is (0,0))
  - Set the gridwidth and gridheight
    - For each component set the number of grid cells the component uses

- GridBagConstraints (cont)
  - example4.java (cont)
  - For the gridheight, I never changed the value from the default of 1, since I didn't make any tall buttons
  - For the gridwidth I used either 1, 2, or java.awt.GridBagConstraints.REMAINDER
    - java.awt.GridBagConstraints.REMAINDER tells the layout manager to finish off the row with the component
      - If you use this, you can get away without setting the gridy for each component
  - And I set fill to java.awt.GridBagConstraints.HORIZONTAL, which tells the layout manager to expand the components in size to fill in all of the cells that they are supposed to fill
    - I didn't set the VERTICAL one, since I don't have any tall buttons

- GridBagConstraints (cont)
  - example4.java (cont)
  - We want a collection of buttons as follows (labels are (row, col, width))



- Create the panel, the grid bag, set the panel's layout and create the grid bag constratints

```
// create the panel to hold the four buttons
javax.swing.JPanel buttonPanel = new javax.swing.JPanel();
java.awt.GridBagLayout gridBagLayout = new java.awt.GridBagLayout();
buttonPanel.setLayout(gridBagLayout);
java.awt.GridBagConstraints buttonPanelConstraints = new java.awt.GridBagConstraints();
```

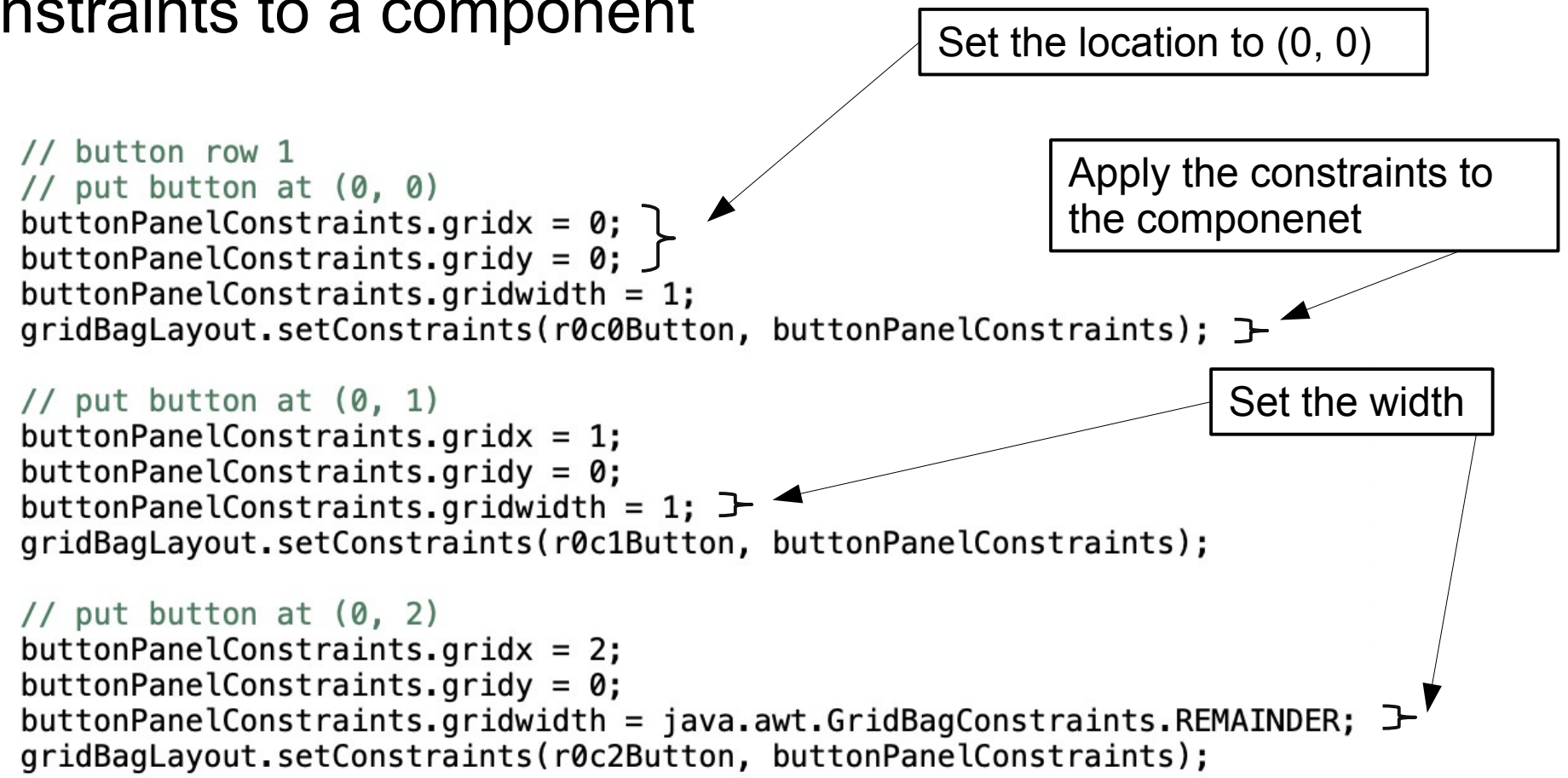
- Create the buttons

```
// create the buttons
javax.swing.JButton r0c0Button = new javax.swing.JButton("row 0 column 0");
javax.swing.JButton r0c1Button = new javax.swing.JButton("row 0 column 1");
javax.swing.JButton r0c2Button = new javax.swing.JButton("row 0 column 2");
javax.swing.JButton r1c01Button = new javax.swing.JButton("row 1 column 0 - 1");
javax.swing.JButton r1c2Button = new javax.swing.JButton("row 1 column 2");
javax.swing.JButton r2c0Button = new javax.swing.JButton("row 2 column 0");
javax.swing.JButton r2c12Button = new javax.swing.JButton("row 2 column 1 - 2");
javax.swing.JButton r3c0Button = new javax.swing.JButton("row 3 column 0");
javax.swing.JButton r3c1Button = new javax.swing.JButton("row 3 column 1");
javax.swing.JButton r3c2Button = new javax.swing.JButton("row 3 column 2");
javax.swing.JButton r4c0Button = new javax.swing.JButton("row 4 column 0");
javax.swing.JButton r4c2Button = new javax.swing.JButton("row 4 column 2");
```

- Set the constraint weightx, weighty, and fill
  - We only need to set these once, since they are not changing

```
// make the weights non zero so that the components spread out
buttonPanelConstraints.weightx = 1;
buttonPanelConstraints.weighty = 1;

// have the components fill all of the cells that they occupy
buttonPanelConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
```

- Set the constraints for the first row of buttons
  - We use the setConstraints method to apply the constraints to a component




```
// button row 1
// put button at (0, 0)
buttonPanelConstraints.gridx = 0;
buttonPanelConstraints.gridy = 0;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r0c0Button, buttonPanelConstraints);

// put button at (0, 1)
buttonPanelConstraints.gridx = 1;
buttonPanelConstraints.gridy = 0;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r0c1Button, buttonPanelConstraints);

// put button at (0, 2)
buttonPanelConstraints.gridx = 2;
buttonPanelConstraints.gridy = 0;
buttonPanelConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
gridBagLayout.setConstraints(r0c2Button, buttonPanelConstraints);
```

- Set the constraints for the second row of buttons
  - There are only two buttons on this row, with the first one being double width

```
// button row 2
// put button at (1, 0)
buttonPanelConstraints.gridx = 0;
buttonPanelConstraints.gridy = 1;
buttonPanelConstraints.gridwidth = 2;
gridBagLayout.setConstraints(r1c01Button, buttonPanelConstraints);

// put button at (1, 1)
buttonPanelConstraints.gridx = 2;
buttonPanelConstraints.gridy = 1;
buttonPanelConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
gridBagLayout.setConstraints(r1c2Button, buttonPanelConstraints);
```

- Set the constraints for the third row of buttons
  - There are only two buttons on this row, with the second one being double width
    - We use the REMAINDER constant to have it finish off the row, which makes it double width

```
// button row 3
// put button at (2, 0)
buttonPanelConstraints.gridx = 0;
buttonPanelConstraints.gridy = 2;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r2c0Button, buttonPanelConstraints);

// put button at (2, 1)
buttonPanelConstraints.gridx = 1;
buttonPanelConstraints.gridy = 2;
buttonPanelConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
gridBagLayout.setConstraints(r2c12Button, buttonPanelConstraints);
```

- Set the constraints for the fourth row of buttons
  - Just three single width boring buttons

```
// button row 4
// put button at (3, 0)
buttonPanelConstraints.gridx = 0;
buttonPanelConstraints.gridy = 3;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r3c0Button, buttonPanelConstraints);

// put button at (3, 1)
buttonPanelConstraints.gridx = 1;
buttonPanelConstraints.gridy = 3;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r3c1Button, buttonPanelConstraints);

// put button at (3, 2)
buttonPanelConstraints.gridx = 2;
buttonPanelConstraints.gridy = 3;
buttonPanelConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
gridBagLayout.setConstraints(r3c2Button, buttonPanelConstraints);
```

- Set the constraints for the fifth row of buttons
  - There are only two buttons, one in the first column, and in the third column, and none in the second column

```
// button row 5
// put button at (4, 0)
buttonPanelConstraints.gridx = 0;
buttonPanelConstraints.gridy = 4;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r4c0Button, buttonPanelConstraints);

// put button at (4, 2)
buttonPanelConstraints.gridx = 2;
buttonPanelConstraints.gridy = 4;
buttonPanelConstraints.gridwidth = 1;
gridBagLayout.setConstraints(r4c2Button, buttonPanelConstraints);
```

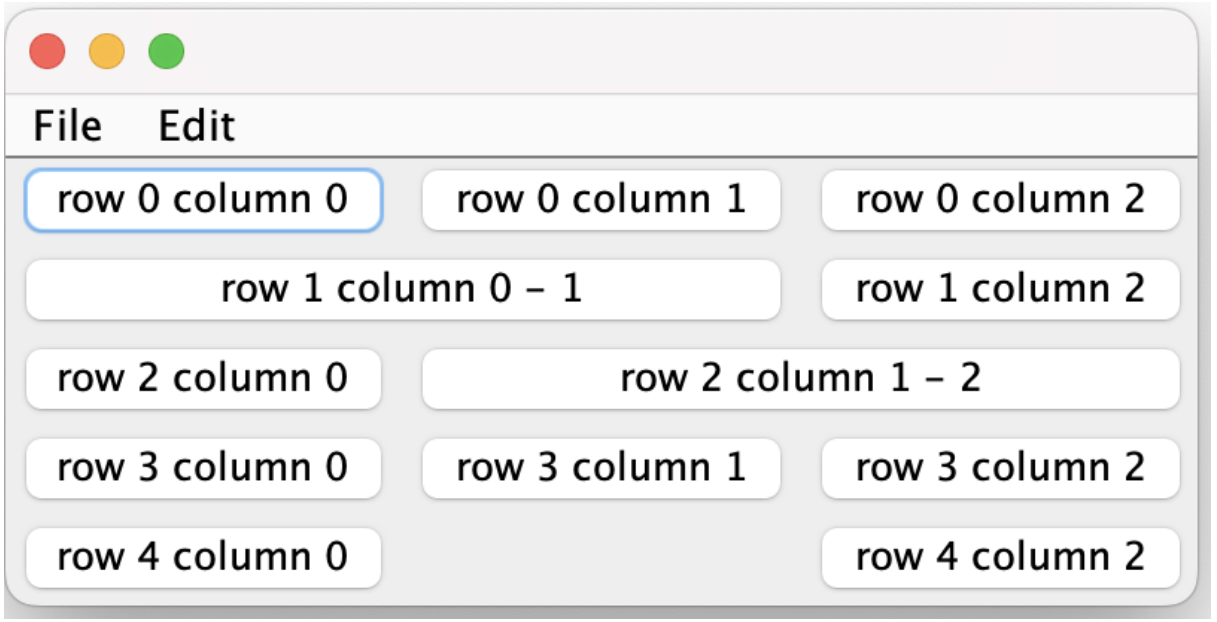
- Add the buttons to the panel

```
// add the buttons to the panel
buttonPanel.add(r0c0Button);
buttonPanel.add(r0c1Button);
buttonPanel.add(r0c2Button);
buttonPanel.add(r1c01Button);
buttonPanel.add(r1c2Button);
buttonPanel.add(r2c0Button);
buttonPanel.add(r2c12Button);
buttonPanel.add(r3c0Button);
buttonPanel.add(r3c1Button);
buttonPanel.add(r3c2Button);
buttonPanel.add(r4c0Button);
buttonPanel.add(r4c2Button);
```

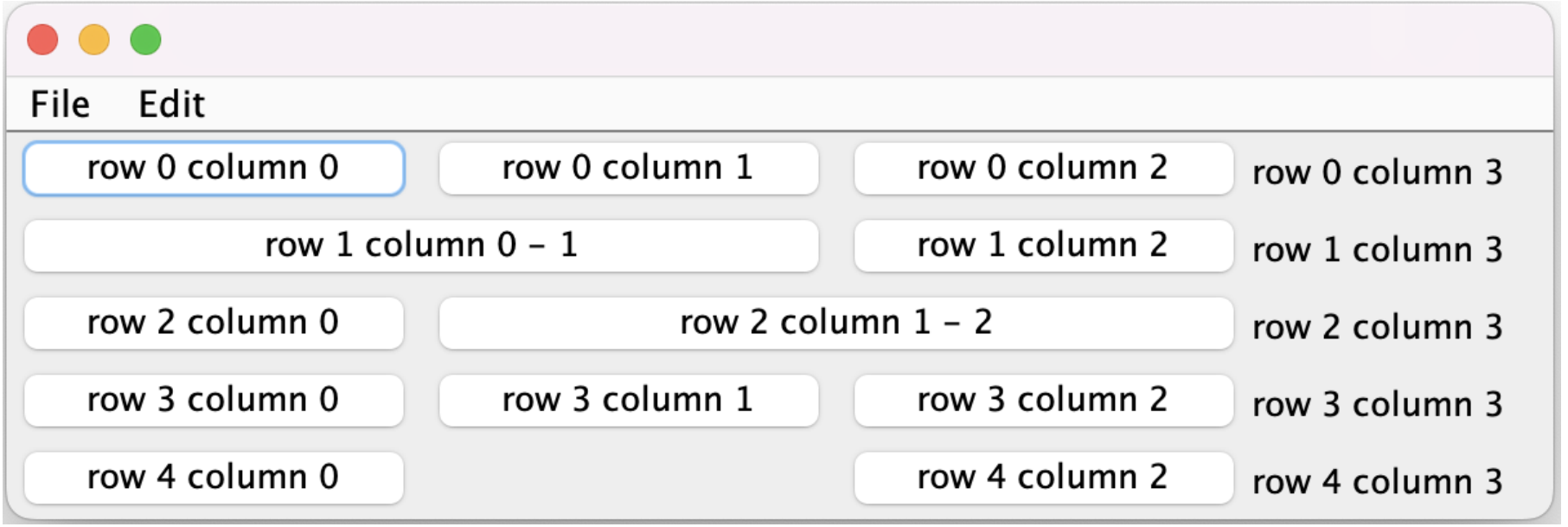
- Create the main panel and add the menubar and button panels to it
  - Note that we are adding the menubar to the main panel as the north component versus of as the menubar of the frame that the main panel will be added to a the content pane
    - This is an example of how there are often more than one way to setup your GUI with Swing (or any GUI/GUI builder)

```
// create the panel
javax.swing.JPanel mainPanel = new javax.swing.JPanel();
mainPanel.setLayout(new java.awt.BorderLayout());
mainPanel.add(menuBar, java.awt.BorderLayout.NORTH);
mainPanel.add(buttonPanel, java.awt.BorderLayout.CENTER);
```

- Here is what example4.java looks like



- example4c.java
  - This is a minor modification to example4b.java, in which we add a row of labels to the right end of each row



- Take a look at the Oracle Java tutorial related to Swing,
  - https://docs.oracle.com/javase/tutorial/uiswing/
- One last example (example5.java)
  - We create a class that extends JPanel, called DrawingArea, that we can use to
    - Draw lines on the screen
    - Read an image from a file and display it on the screen
    - Write the image that is contained in the panel
    - We also want to capture
      - Keyboard events
        - Key press
        - Modifier press (alt, shift, and so on)
      - Mouse events
        - Button press
        - Button release
        - Button click
      - Mouse motion
        - Mouse dragged
        - Mouse moved
      - Component events
        - Window resize
        - Component visibility

- example5.java (cont)
  - The GUI contains
    - Frame
    - Panel with BorderLayout
      - North is a menubar
      - Center is our class that extends JPanel
    - Menubar
      - File menu
        - Save button saves the image as a png
        - Exit button exits the program
      - Edit menu
        - Color button pops up a warning window
        - set the line color red
        - set the line color green
        - set the line color blue
        - set the line color yellow
        - plot sine/cosine/tangent curves

- example5.java (cont)
  - We modify the MenuItemActionListener class to contain a second field, the DrawingArea object
    - So that we can call methods in the DrawingArea class to
      - Change the line color
      - Save the current image to a png
      - Plot the sine/cosine/tangent curves

```
// action listener for the menu items
static class MenuItemActionListener implements java.awt.event.ActionListener
{
    // the menu item associated with the action listener, so that we can
    // share this one class with multiple menu items
    private javax.swing.JMenuItem m;
    private DrawingArea drawingArea;

    MenuItemActionListener(javax.swing.JMenuItem m, DrawingArea drawingArea)
    {
        this.m = m;
        this.drawingArea = drawingArea;
    }
```

- example5.java (cont)
  - We also add if statements to the MenuItemActionListener class to make calls to the DrawingArea to
    - Change the line color
    - Save the image
    - Plot the sine/cosine/tangent curves

- example5.java (cont)

```
public void actionPerformed(java.awt.event.ActionEvent e)
{
    System.out.println("action performed on " + m.getText() + " menu item");

    // if exit is selected from the file menu, exit the program
    if( m.getText().toLowerCase().equals("exit") )
    {
        System.exit(0);
    }

    if( m.getText().toLowerCase().equals("save") )
    {
        drawingArea.writeImage("output.png", "png");
    }

    // if color is selected from the edit menu, put a popup on the screen
    // saying something
    if( m.getText().toLowerCase().equals("color") )
    {
        Object[] options = {"OK"};
        javax.swing.JOptionPane.showOptionDialog(null, "This is unimplemented,\nclick OK to continue",
                "Warning", javax.swing.JOptionPane.DEFAULT_OPTION,
                javax.swing.JOptionPane.WARNING_MESSAGE, null, options, options[0]);
    }

    if( m.getText().toLowerCase().equals("set line color red") )
    {
        drawingArea.setLineColor(new java.awt.Color(255, 0, 0, 255));
    }

    if( m.getText().toLowerCase().equals("set line color green") )
    {
        drawingArea.setLineColor(new java.awt.Color(0, 255, 0, 255));
    }

    if( m.getText().toLowerCase().equals("set line color blue") )
    {
        drawingArea.setLineColor(new java.awt.Color(0, 0, 255, 255));
    }

    if( m.getText().toLowerCase().equals("set line color yellow") )
    {
        drawingArea.setLineColor(new java.awt.Color(255, 255, 0, 255));
    }

    if( m.getText().toLowerCase().equals("plot") )
    {
        drawingArea.plotSineAndCosineAndTangent();
    }
}
```

exit button

save button

color button

red button

green button

blue button

yellow button

plot button

- example5.java (cont)
  - We use the ImageIO package to read and write the images to files
    - It has static methods to read various file formats
    - It has static methods to write various file formats
    - Looks like the following formats are supported
      - jpeg, tiff, bmp, gif, wbmp, png
      - There are methods to find out what file formats are supported
  - The Color class is used to specify colors (look at the Java doc)
    - Uses 32 bit colors
      - 8 bits of red, green, blue, alpha
      - The 32 bits are packed into an int
      - Alpha specifies the transparency
        - 0 is 100% transparent
        - 255 is 0% transparent

- The Color class
  - Has constants for well know colors
    - Black, blue, cyan, dark grey, green, light grey, magenta, orange, pink, red, white, yellow
    - Has constructors for
      - Floats with RGB/RGBA values
      - Ints with RGB/RGBA values
      - Int with packed RGB/RGBA values
    - Has various methods to get the components of the color

- example5.java (cont)
  - DrawingArea class
    - Extends JPanel
    - Implelements MouseListener, MouseMotionListener, KeyListener, ComponentListener
    - We use offscreen memory to draw the image, and then when the image is ready we draw the offscreen image to the screen
      - This is similar to “double buffering”, where there are two image buffers, the current image buffer and the next image buffer
        - The next image buffer is being updated while the current image buffer is displayed, when the next image is ready, the two buffers are swapped

- example5.java (cont)
  - DrawingArea class (cont)
    - The class uses a BufferedImage for the offscreen image
      - A buffer for image data
    - And uses a Graphics2D object to draw on the BufferedImage
      - Class for 2D graphics (extends Graphics)
      - Has methods to draw various things
        - Geometric objects
          - Lines, rectangles, ovals, shapes
        - BufferedImages
        - Strings
        - Apply transformations
    - We aren't going to do anything spectacular, we are going to draw lines when the user drags the mouse

- BufferedImage class
  - Subclass of the Image class
  - There is a constructor that takes the size and type of the image
  - I typically create a BufferedImage indirectly
    - BufferedImage bi = java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment().getDefaultScreenDevice().getDefaultConfiguration().createCompatibleImage(maxWidth, maxHeight, java.awt.image.BufferedImage.TYPE_INT_ARGB);
  - You can set individual pixel colors, but that is pretty slow
  - I typically get the Graphics2D object associated with the BufferedImage using the createGraphics() method

- Graphics2D class
  - Extends Graphics class
  - Used to draw 2d shapes, text, and images
  - Has methods to
    - Draw shapes, BufferedImages, apply transforms, set colors (foreground & background), draw lines, strings, rectangles, polygons, polylines, arcs, ovals
  - You can't directly draw 3d graphics with a Graphics2D object, but you can do the math and project 3d graphics onto the Graphics2D object
    - This requires quite a bit of work
  - I used BufferedImages and Graphics2D extensively in my former job to show visualizations of various analysis and algorithm results
    - An image or animation can often convey a lot on information more easily than many slides and complicated discussions
- Java also has a 3D API
  - I haven't used this in quite some time

- example5.java (cont)
  - DrawingArea class (cont)
    - Allocate the BufferedImage
    - Get the Graphics2D associated with the BufferedImage (to draw on the BufferedImage)
    - Load an image and draw it on the BufferedImage (if an image filename was supplied)
    - Capture mouse, mouse motion, and keyboard events
      - When the mouse button is pressed, capture the current (x, y) location of the mouse and if the mouse is dragged, draw a line from the last mouse position to the current mouse position
      - If the Save menu item is selected, it calls the writeImage() method
      - If one of the “set line color” menu items is selected, the setLineColor() method is called

- example5.java (cont)
  - DrawingArea class (cont)
    - If the “c” key is pressed, any lines that were draw are cleared
      - “c” wants to equate to clear
    - If the “r” key is pressed, any lines that were draw are cleared and the line color is set back to black
      - “r” wants to equate to reset
    - The x/y/X/Y keys change the scale on the sine/cosine/tangent plot

- example5.java (cont)
  - DrawingArea class (cont)
    - If a filename is provide, the image is read and stored in a BufferedImage, and then copied to the main BufferedImage that all of the drawing is done on
      - bi is the main BufferedImage
      - bi2 is the BufferedImage that the image is written to
      - biG is the Graphics2D associated with bi

```
bi2 = null;
if( filename.length() > 0 )
{
    try
    {
        bi2 = javax.imageio.ImageIO.read(new java.io.File(filename));
        biG.drawImage(bi2, 0, 0, null);
    }
    catch(Exception e)
    {
        System.out.println(e.toString());
        System.exit(0);
    }
}
```

- example5.java (cont)
  - DrawingArea class (cont)
    - Since we are handling the drawing for the JPanel, we override the paintComponent() method
      - The Graphics parameter is the Graphics object associated with the screen
      - We call the parent paintComponent() and then copy the BufferedImage bi to the Graphics object parameter

```
public void paintComponent(java.awt.Graphics g)
{
    super.paintComponent(g);
    if( bi == null )
    {
        return;
    }
    g.drawImage(bi, 0, 0, null);
}
```

- example5.java (cont)
  - DrawingArea class (cont)
    - When the mouse is dragged, we use the Graphics2D biG associated with the BufferedImage bi to draw a line from the last mouse location to the current mouse location and then update the last mouse location
      - repaint(0) is the method to repaint the window as soon as possible

```
public void mouseDragged(java.awt.event.MouseEvent e)
{
    int x = e.getX();
    int y = e.getY();

    if( (lastX >= 0) && (lastY >= 0) )
    {
        biG.drawLine(lastX, lastY, x, y);
        lastX = x;
        lastY = y;
        repaint(0);
    }
    else
    {
        lastX = x;
        lastY = y;
    }
}
```

- example5.java (cont)
  - DrawingArea class (cont)
    - The keyPressed() method handles key press events

```
public void keyPressed(java.awt.event.KeyEvent e)
{
    if( e.getKeyCode() == e.VK_C )
    {
        biG.setColor(backgroundColor);
        biG.fillRect(0, 0, maxWidth, maxHeight);
        if( filename.length() > 0 )
        {
            biG.drawImage(bi2, 0, 0, null);
        }
        biG.setColor(lineColor);
        lastX = -1;
        lastY = -1;

        repaint(0);
    }

    if( e.getKeyCode() == e.VK_R )
    {
        biG.setColor(backgroundColor);
        biG.fillRect(0, 0, maxWidth, maxHeight);
        if( filename.length() > 0 )
        {
            biG.drawImage(bi2, 0, 0, null);
        }
        lineColor = defaultLineColor;
        biG.setColor(lineColor);
        lastX = -1;
        lastY = -1;

        repaint(0);
    }
}
```

The “c” button is pushed
- Clear bi
- Reset the last mouse location
- Request the panel repainted

The “r” button is pushed
- Clear bi
- Reset the last mouse location
- Set the line color to black
- Request the panel repainted

- example5.java (cont)
  - The keyPressed() (cont)

```
if( e.getKeyCode() == e.VK_X )
{
    if( e.isShiftDown() )
    {
        scaleX = scaleX+1.0;
    }
    else
    {
        scaleX = scaleX-1.0;
    }
    plotSineAndCosineAndTangent();
}

if( e.getKeyCode() == e.VK_Y )
{
    if( e.isShiftDown() )
    {
        scaleY = scaleY+1.0;
    }
    else
    {
        scaleY = scaleY-1.0;
    }
    plotSineAndCosineAndTangent();
}
```

The “x” button is pushed
- Update the x scale
- Redraw the sine/cosine/tangent plots

The “y” button is pushed
- Update the y scale
- Redraw the sine/cosine/tangent plots