Chapter 10: Swing Interfaces and Working with a User Interface

Swing Features

The components we’ve already talked about were Swing versions of classes that were part of the **Abstract Windowing Toolkit (awt)** which is the original Java package for GUI programming. There are many other components including keyboard mnemonics, Tooltips and standalone dialog boxes.

Standard Dialogs

The **JOptionPane** class offers several methods that can be used to create standard **DIALOG BOX**es – small windows that ask a question, present the user with a message or notify the user of an error.

There are four classes of the standard dialog box, each with its own display method in the JOptionPane class. They are:

ConfirmDialog – Asks a question with buttons for yes, no and cancel responses

InputDialog – Prompts for text input

MessageDialog – Displays a message

OptionDialog – Comprises all three of the other dialog boxes

Confirm Dialog Boxes

The **showConfirmDialog(Component, Object)** method is the easiest way to create a Yes/No/Cancel dialog box. The component argument specifies the container that's the parent of the dialog box which determines where the dialog window should be displayed. If this variable is null rather than a container, or if the container is not a JFrame object, the dialog box will be in the center of the screen.

The object argument can be a string, a component or an Icon object. If a string, the text will be displayed in the dialog box. If a component or an icon, the object will be displayed in place of the text message.

The method returns one of five possible integer values, each a class constant of JOptionPane: YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION or CLOSED_OPTION. The following is an example that uses a confirm dialog box with a text message and stores the response in a response variable.

int response = **JOptionPane.showConfirmDialog(null, "Should I delete all of your files?")**;

Input Dialog Boxes

This type of dialog asks a question and uses a text field to store the response. We call the showInputDialog(Component, Object) method to create it using the parent component argument and the string, component or icon to display in the box. This method returns a string that represents the user's response. The following statement creates an input box of this type:

String response = **JOptionPane.showInputDialog(null, "Enter your name: ")**;

A longer method call with more options is the showInputDialog(Component, Object, String, int).

Usage of the longer method is shown below:

String response = **JOptionPane.showInputDialog(null,**

"What is your zip code? "

"Enter your zip code: "

JOptionPane.QUESTION_MESSAGE);

Message Dialog Boxes

A message dialog box is a simple window that displays information. This dialog box is created with the **showMessageDialog(Component, Object)** method. The arguments, like the other dialog boxes we've looked at, are the parent component and the string, component or icon to display.

Unlike the other boxes, this dialog does not return a response value. This is shown in the following code snippet.

JOptionPane.showMessageDialog(null, "The program has been uninstalled.");

Option Dialog Boxes

The Option dialog box is the most complex since it contains the features of all of the other dialog boxes. It is created with the method specified as shown here:

showOptiondialog(Component, Object, String, int, int, Icon, Object[], Object)

Exercise 1: FeedInfo.java, p. 206-7

The FeedInfo.java application uses dialog boxes to get information from the user. The information gathered is then placed into text fields in the application’s main window.

Using Luna, please code the following naming the module FeedInfo.java. Once you get a clean compile, test your application ensuring that all paths are tested.

Please note that after the user fills in the fields of the boxes, the applications main window will be displayed.

FeedInfo.java, p. 206-7

```
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
public class FeedInfo extends JFrame {
   private JLabel nameLabel = new JLabel("Name: ",
      SwingConstants.RIGHT);
   private JTextField name;
   private JLabel urlLabel = new JLabel("URL: ",
      SwingConstants.RIGHT);
   private JTextField url;
   private JLabel typeLabel = new JLabel("Type: ",
      SwingConstants.RIGHT);
   private JTextField type;
```

```
public FeedInfo() {
      super("Feed Information");
      setSize(400, 145);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLookAndFeel();
      // Site name
      String response1 = JOptionPane.showInputDialog(null,
         "Enter the site name:");
      name = new JTextField(response1, 20);
      // Site address
      String response2 = JOptionPane.showInputDialog(null,
         "Enter the site address:");
      url = new JTextField(response2, 20);
```

FeedInfo.java Continued

```
// Site type
        String[] choices = { "Personal", "Commercial",
"Unknown" };
        int response3 = JOptionPane.showOptionDialog(null,
            "What type of site is it?",
            "Site Type",
            0,
            JOptionPane.QUESTION_MESSAGE,
            null,
            choices,
            choices[0]);
        type = new JTextField(choices[response3], 20);
}
 setLayout(new GridLayout(3, 2));
        add(nameLabel);
        add(name);
        add(urlLabel);
        add(url);
        add(typeLabel);
        add(type);
        setLookAndFeel();
        setVisible(true);
    }
```

FeedInfo.java Continued

```
private void setLookAndFeel() {
    try {
        UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
        );
        SwingUtilities.updateComponentTreeUI(this);
    } catch (Exception e) {
        System.err.println("Couldn't use the system "
            + "look and feel: " + e);
    }
  }
 public static void main(String[] arguments) {
    FeedInfo frame = new FeedInfo();
  }
```

Sliders

SLIDERS enable the user to set a number by sliding a control bar within a minimum and maximum range such as a volume control. It can be used for numeric input instead of a text field.

The constructor methods for JSlider are as follows:

JSlider(int) – Creates a slider with a specified orientation, minim value of 0 and maximum value of 100 starting and a starting value of 50

JSlider(int, int) – Creates a slider with the specified minimum value and maximum value

JSlider(int, int, int) – Creates a slider with the specified minimum, maximum and starting value

JSlider(int, int, int, int) – Creates a slider with the specified orientation, minimum, maximum and starting value

Sliders Continued

When using a slider, we have an optional label that can be used to indicate the minimum, maximum and two different set of tick marks ranging between their values. The default for these arguments are minimum of 0, maximum of 100, starting have 50 and horizontal orientation.

setMajorTickSpacing(int) – This value sets the major tick marks by the specified distance that is specified in values between the minimum and maximum values

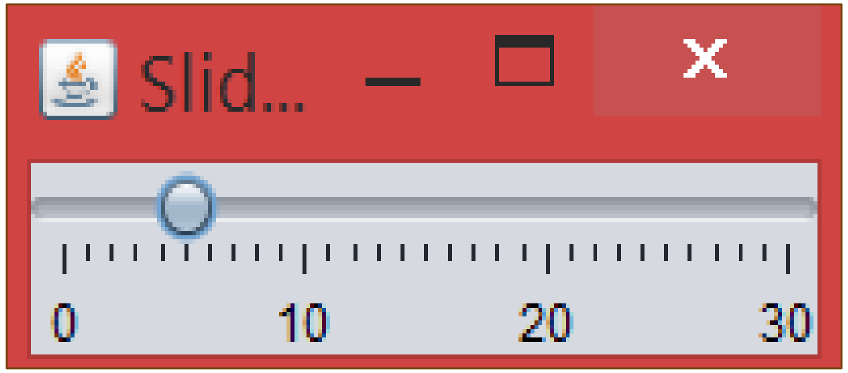
setMinorTickSpacing(int) – This value sets the minor tick marks by the specified distance and are half the height of the major tick marks

setPaintTicks(boolean) – This true/false value determines whether the tick mark should be displayed (true) or not (false)

setPaintLabels(boolean) – This true/false value determines whether the slider's numeric label should be displayed (true) or not (false)

Slider.java, p. 210

Please use Luna to code the following module naming it Slider.java. Once a clean compile is achieved, please run the application to achieve the results as shown.



Silder.java

```
import java.awt.event.*;
import javax.swing.*;
public class Slider extends JFrame {
  public Slider() {
    super("Slider");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLookAndFeel();
    JSlider pickNum = new JSlider(JSlider.HORIZONTAL, 0, 30, 5);
    pickNum.setMajorTickSpacing(10);
    pickNum.setMinorTickSpacing(1);
    pickNum.setPaintTicks(true);
    pickNum.setPaintLabels(true);
add(pickNum);
    pack();
    setVisible(true);
  }
  private void setLookAndFeel() {
    try {
      UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
      );
      SwingUtilities.updateComponentTreeUI(this);
    } catch (Exception e) {
      System.err.println("Couldn't use the system "
        + "look and feel: " + e);
```

Slider.java Continued

```
}
    }


    public static void main(String[] args) {
        Slider frame = new Slider();
    }
}
```

Scroll Panes

Like sliders, **SCROLL BARS** allow the user to navigate the display of screen by moving the display in either a vertical or horizontal fashion such as we see with web browsing. When using Swing to manage our scroll bars, a couple of rules must be kept in mind.

To enable scrolling, the component must be added to the **JScrollPane** container

The JScrollPane container must be added to a container in place of the scrollable component

We create a scroll pane by using the **JScrollPane(Object)** constructor where Object represents the component that can be scrolled. In the following example, this is demonstrated to create a text area in a scroll pane called scroller and then adds it to a container called mainPane.

JTextArea textbox = new JTextArea(7, 30);

JScrollPane scroller – new JScrollPane(textbox);

mainPane.add(scroller);

Scroll Panes

To control the size we want the component to use we can call the **setPreferredSize(Dimension)** method before adding it the container. The **Dimension** object represents the width and height in pixels. The following example expands the one used previously for scroller to add a dimension. As shown, we should set the Dimension before adding it to a container – this is very important when working with a scroller.

Dimension pref = new Dimension(350, 100);

Scroller.setPreferredSize(pref);

Scroll Panes

As we expect, scroll panes do not scroll unless it is needed – the bars will automatically appear when needed. To override this standard behavior, we can set the **POLICY** (rules) for jScrollBar components using one of the following class constants when we create the scroll bar using the jScrollPane(Object, int, int) constructor:

HORIZONTAL_SCROLLBAR_ALWAYS

HORIZONTAL_SCROLLBAR_AS _NEEDED

HORIZONTAL_SCROLLBAR_NEVER

VERTICAL_SCROLLBAR_ALWAYS

VERTICAL_SCROLLBAR_AS _NEEDED

VERTICAL_SCROLLBAR_NEVER

```
JScrollPane scroller = new JScrollPane(textbox,
            VERTICAL_SCROLLBAR_ALWAYS,
            HORIZONTAL_SCROLLBAR_NEVER);
```

Toolbars

We create a toolbar with the JToolBar class which groups several components in a row or column and most often buttons.   **TOOLBARS** are rows or columns of components that group the most commonly used program options. They often contain buttons and lists that can be used as an alternative to using menus and shortcut keys. By default they are horizontal but the orientation can be set with the **HORIZONTAL** or **VERTICAL** class variables of the **SwingContstants** interface. The toolbar's constructor methods are shown in the list below. After we create a tool bar, we add components to it with the **add(Object)** method where Object represents the components to be placed on the toolbar.

Toolbars

If we allow the user to move the toolbar to a different location on the screen, we call this a **DOCKABLE TOOLBAR** since we can “dock” them at a location chosen by the user. Swing toolbars can also be docked in another window.

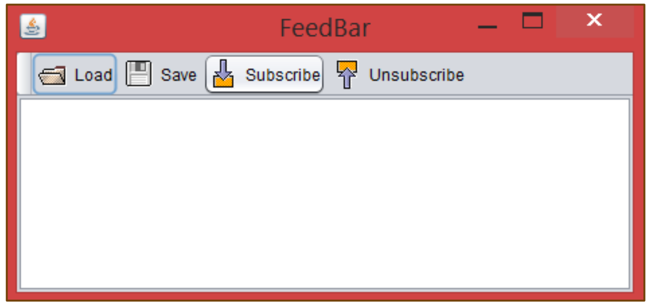
To achieve the best results, a dockable JToolBar component should be arranged in a container using the **BorderLayout** class which is a user interface class that is referred to as a **LAYOUT MANAGER**. The border layout divides a container into five areas: north, south, east, west and center where each of the directional components takes up the space it needs in the container and the remainder goes to the center. The tool bar should be placed in one of the directional areas.

FeederBar.java, p. 212-3

The application FeedBar.java uses four images to represent the graphics on the buttons to create a toolbar in the north quadrant of the container.

Using Luna, code the following module naming it FeedBar.java. Once a clean compile has been achieved, please test the code by running the application ensuring that all test paths are executed.

Note: C:/ **…** / should be replaced with the path where your .gif files reside. If you don't do this, only the text values will appear on your buttons.



FeederBar.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FeedBar extends JFrame {
   public FeedBar() {
      super("FeedBar");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLookAndFeel();
      // create icons
      ImageIcon loadIcon = new ImageIcon("C:/ … /load.gif");
      ImageIcon saveIcon = new ImageIcon("C:/ … /save.gif");
      ImageIcon subscribeIcon = new ImageIcon("C:/ …
/subscribe.gif");
      ImageIcon unsubscribeIcon = new ImageIcon("C:/…
/unsubscribe.gif");
// create buttons
      JButton load = new JButton("Load", loadIcon);
      JButton save = new JButton("Save", saveIcon);
      JButton subscribe = new JButton("Subscribe",
subscribeIcon);
      JButton unsubscribe = new JButton("Unsubscribe",
unsubscribeIcon);
      // add buttons to toolbar
      JToolBar bar = new JToolBar();
      bar.add(load);
      bar.add(save);
      bar.add(subscribe);
      bar.add(unsubscribe);
```

FeederBar.java

```
// prepare user interface
        JTextArea edit = new JTextArea(8, 40);
        JScrollPane scroll = new JScrollPane(edit);
        BorderLayout bord = new BorderLayout();
        setLayout(bord);
        add("North", bar);
        add("Center", scroll);
        pack();
        setVisible(true);
    }
private void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
            );
```

```
private void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
            );
            SwingUtilities.updateComponentTreeUI(this);
        } catch (Exception e) {
            System.err.println("Couldn't use the system "
                + "look and feel: " + e);
        }
}


    public static void main(String[] arguments) {
        FeedBar frame = new FeedBar();
    }
}
```

Progress Bars

A **PROGRESS BAR** is used to track the progression of some task that can be represented in a numerical fashion. They are created with a minimum and maximum value to represent the beginning and ending of the task. We see these when we download a file or install some software. They are important to the user so that they know that the machine is actually doing something and not frozen.

To create a progress bar, we use the following constructors:

JProgressBar() – Creates a new progress bar

JProgressBar(int, int) – Creates a new progress bar with the specified minimum and maximum value

JProgressBar(int, int, int) – Creates a new progress bar with orientation, minimum and maximum

Progress Bars

We update the progress bar with the **setValue(int)** method indicating how far along in the process the task is at that moment. The following is the lines of code that take in the number of files to be processed and is passed to the setValue() method to immediately update the progress bar to represent the percentage of completion for the task:

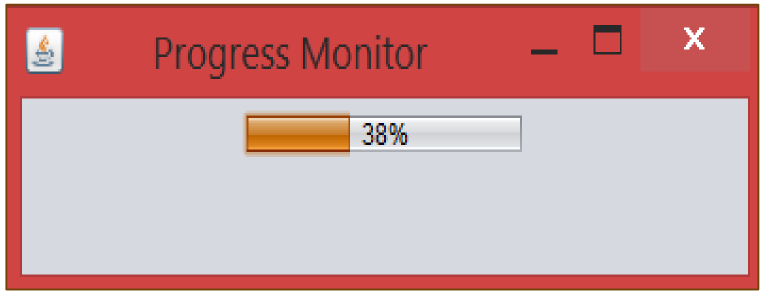
```
int filesDone = getNumberOfFiles();
install.setValue(filesDone);
```

As with our other objects, progress bars often have a text label in addition to some graphic. The label displays the percentage of completion. This is accomplished via the **setStringPainted(boolean)** method with a value of true as its argument (false will turn off the label.

ProgressMonitor.java, p. 215-6

The ProgressMonitor.java application uses a progress bar to track the value of the num variable.

Using Luna, code the following module naming it ProgressMontior.java. Once we have a clean compile, test the application ensuring that all paths in the code is tested. The results should be as shown.



ProgressMonitor.java

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;


public class ProgressMonitor extends JFrame {


    JProgressBar current;

    JTextArea out;

    JButton find;

    Thread runner;

    int num = 0;
```

```
 public ProgressMonitor() {

        super("Progress Monitor");

setDefaultCloseOperation(JFrame.EXIT_ON_CLO
SE);

        setLookAndFeel();

        setSize(205, 68);

        setLayout(new FlowLayout());

        current = new JProgressBar(0, 2000);

        current.setValue(0);

        current.setStringPainted(true);

        add(current);

    }
```

ProgressMonitor.java

```
public void iterate() {
        while (num < 2000) {
            current.setValue(num);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) { }
            num += 95;
        }
    }
    private void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
            );
            SwingUtilities.updateComponentTreeUI(this);
 } catch (Exception e) {
            System.err.println("Couldn't use the system "
                + "look and feel: " + e);
        }
    }
public static void main(String[] arguments) {
        ProgressMonitor frame = new ProgressMonitor();
        frame.setVisible(true);
        frame.iterate();
    }
}
```

Menus

A **MENU BAR** is a series of pull-down menus used to perform some task such as save or open. They are often used as an option to buttons or other interface components since they are hidden most of the time therefore requiring less screen real estate.

In Java there are three components to support menus and work together to achieve the desired results:

JMenuBar – A container that holds one or more JMenu components and displays their names

JMenu – A drop-down menu that contains one or more JMenuItem components, other interface components and separators (lines that are displayed between menu items)

JMenuItem – An Item on a menu

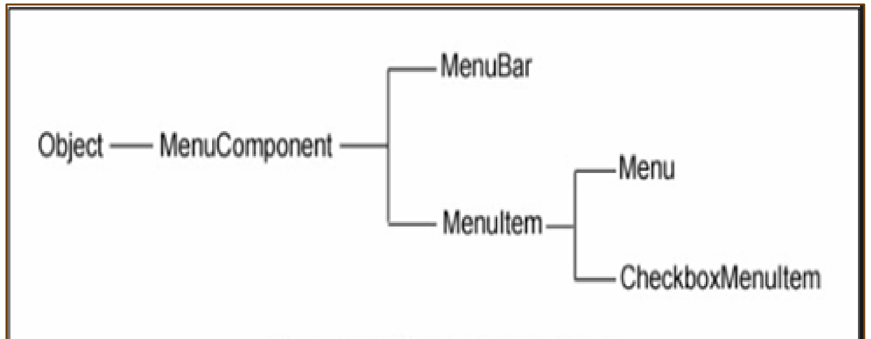


Figure: Menu Component Hierarchy

Menus

A JMenu container holds all of the menu items for a drop down menu. We create it by calling **JMenu(String)** constructor with the name of the menu as the an argument – this name will also appear on the menu bar. After we create the JMenu container we call **add(JMenuItem) t**o add a menu item to the menu – new items are placed at the end. We can call the **add(Component)** method with a user interface as an argument to place items that are not necessarily a menu item such as a checkbox. To add a line separator between items we can call the **addSeparator()** method – this is often done to group like items. We add text to the menu to serve as a label using the **add(String)** method with text as its argument.

Menus

The following is an example of the add() methods we’ve described.

```
JMenu m1 = new JMenu(“File”);
m1.add(j1);
 m1.add(j2);
m1.add(j3);
m1.addSeperator();
m1.add(j4);
m1.add(j5);
m1.addSeperator();
m1.add(j6);
m1.addSeperator();
```

Menus

A JMenuBar container holds one or more JMenu containers and displays each of their names. Typically we see a menu bar directly below an applications title bar. We create the menu bar using the constructor **JMenuBar()** with no arguments. We can add menus to the end of the bar by call the **add(JMenu)** method. After all items are created, added them to the menus and added the menus to a bar, we add them to a frame using the frame's **setJMenuBar(JMenuBar)** method as shown below:

JMenuBar bar = new JMenuBar();

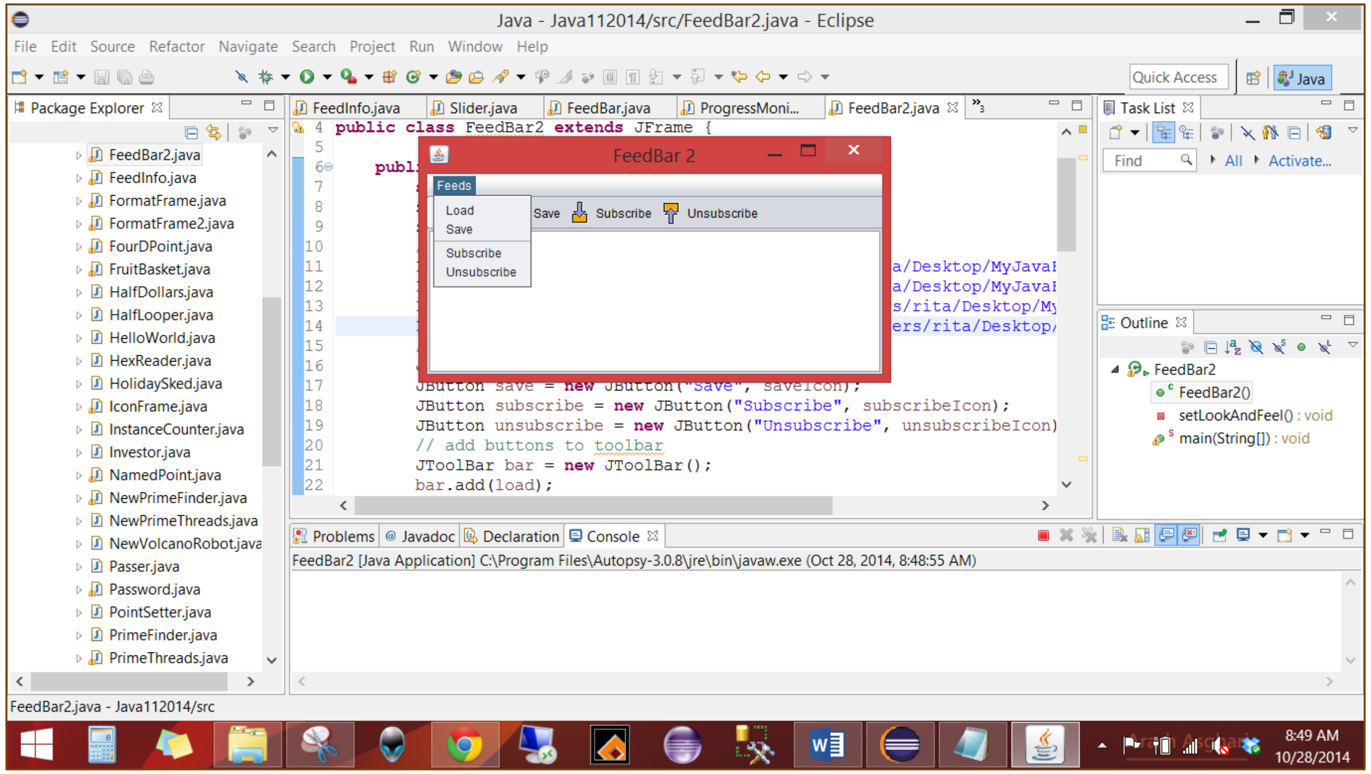
bar.add(m7);

gui.setJMenuBar(bar);

Exercise 1: FeedBar2.java, p. 218-20

In this exercise we build on FeedBar.java by adding a menu bar that holds one menu and four individual items. When we click on these menu items, nothing happens yet. We will explore this when we talk about user input.

Please code the following module naming it FeedBar2.java. Once a clean compile has been achieved, test the module ensuring that all test paths have been covered.



```
public class FeedBar2 extends JFrame {

    publ

        JButton save = new JButton("Save", saveIcon);
        JButton subscribe = new JButton("Subscribe", subscribeIcon);
        JButton unsubscribe = new JButton("Unsubscribe", unsubscribeIcon)
        // add buttons to toolbar
        JToolBar bar = new JToolBar();
        bar.add(load);
```

```
FeedBar2 [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Oct 28, 2014, 8:48:55 AM)
```

FeedBar2.java

```
import java.awt.*;

import javax.swing.*;

 public class FeedBar2 extends JFrame {

    public FeedBar2() {

        super("FeedBar 2");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setLookAndFeel();

// create icons

        ImageIcon loadIcon = new
ImageIcon("C:/Users/rita/Desktop/MyJavaFiles/loa
d.gif");

        ImageIcon saveIcon = new
ImageIcon("C:/Users/rita/Desktop/MyJavaFiles/sav
e.gif");

ImageIcon subscribeIcon = new
ImageIcon("C:/Users/rita/Desktop/MyJavaFiles/subscribe.gif");

        ImageIcon unsubscribeIcon = new
ImageIcon("C:/Users/rita/Desktop/MyJavaFiles/unsubscribe.gi
f");

        // create buttons

        JButton load = new JButton("Load", loadIcon);

        JButton save = new JButton("Save", saveIcon);

        JButton subscribe = new JButton("Subscribe",
subscribeIcon);

        JButton unsubscribe = new JButton("Unsubscribe",
unsubscribeIcon);
```

FeedBar2.java

```
// add buttons to toolbar
    JToolBar bar = new JToolBar();
    bar.add(load);
    bar.add(save);
    bar.add(subscribe);
    bar.add(unsubscribe);
// create menu
    JMenuItem j1 = new JMenuItem("Load");
    JMenuItem j2 = new JMenuItem("Save");
    JMenuItem j3 = new JMenuItem("Subscribe");
    JMenuItem j4 = new JMenuItem("Unsubscribe");
    JMenuBar menubar = new JMenuBar();
JMenu menu = new JMenu("Feeds");
    menu.add(j1);
    menu.add(j2);
    menu.addSeparator();
    menu.add(j3);
    menu.add(j4);
menubar.add(menu);
    // prepare user interface
    JTextArea edit = new JTextArea(8, 40);
    JScrollPane scroll = new JScrollPane(edit);
    BorderLayout bord = new BorderLayout();
```

FeedBar2.java

```
setLayout(bord);
        add("North", bar);
        add("Center", scroll);
        setJMenuBar(menubar);
        pack();
        setVisible(true);
    }
private void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
"com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
            );
            SwingUtilities.updateComponentTreeUI(this);
} catch (Exception e) {
            System.err.println("Couldn't use the system "
                + "look and feel: " + e);
        }
    }

    public static void main(String[] arguments) {
        FeedBar2 frame = new FeedBar2();
    }
}
```

Tabbed Panes

A **TABBED PANE** is a group of stacked panels in which only one panel can be in focus at a time. Tabbed panes are often seen in current browsers where there is a tab per website that we have open. When we want to view a pane, we simply click the tab. This is implemented by the **JTabbedPane** class. Tabs can be arranged either at the top or bottom in a horizontal fashion or on either the left or right in a vertical fashion. We have three constructors available to us:

JTabbedPane() – Creates a horizontal tabbed position along the top that does not scroll.

JTabbedPane(int) – Creates a tabbed pane that does not scroll and has the specified placement

JTabbedPane(int, int) – Creates a tabbed pane with the specified placement (first argument) and the scrolling policy (second argument)

Tabbed Panes

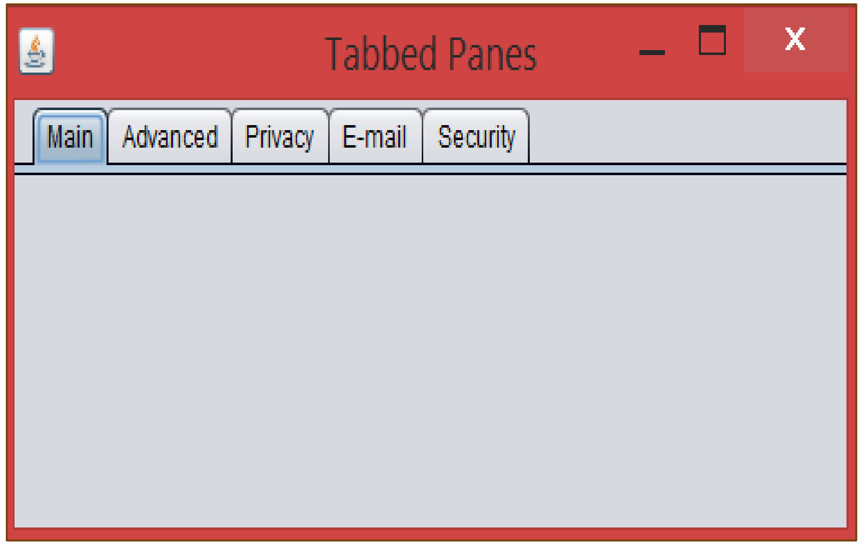
The scrolling policy determines how the tabs will be displayed when there are more tabs than the interface can hold. To scroll a tabbed pane we use the **JTabbedPane.SCROLL_TAB_LAYOUT** or the **JTabbedPane.WRAP_TAB_LAYOUT** to wrap the tabs.

As always, after we crate the tabbed pane, we add components using the **addTab(String, Component)** method where String is the tab's label. The second argument of Component is the component that will make up one of the tables on the pane. It is common to use a JPanel object as the second argument.

TabPanels.java, p. 222-3

In the TabPannels.java application we create a pane with five tabs, each with its own panel.

Please code the following module using Luna naming it TabPanels.java. The application should be tested thoroughly to ensure that all sections work.



TabPanels.java

```
import java.awt.*;
import javax.swing.*;
public class TabPanels extends JFrame {
   public TabPanels() {
      super("Tabbed Panes");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLookAndFeel();
      setSize(480, 218);
      JPanel mainSettings = new JPanel();
      JPanel advancedSettings = new JPanel();
      JPanel privacySettings = new JPanel();
      JPanel emailSettings = new JPanel();
      JPanel securitySettings = new JPanel();
JTabbedPane tabs = new JTabbedPane();
      tabs.addTab("Main", mainSettings);
      tabs.addTab("Advanced", advancedSettings);
      tabs.addTab("Privacy", privacySettings);
      tabs.addTab("E-mail", emailSettings);
      tabs.addTab("Security", securitySettings);
      add(tabs);
      setVisible(true);
   }
private void setLookAndFeel() {
      try {
         UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
         );
         SwingUtilities.updateComponentTreeUI(this);
```

TabPanels.java

```
} catch (Exception e) {
            System.err.println("Couldn't use the
system "
                + "look and feel: " + e);
        }
    }


    public static void main(String[] arguments) {
        TabPanels frame = new TabPanels();
    }
}
```

Working with a User Interface

The **LAYOUT MANAGER** class in Java help us to arrange components on an interface. We are going to explore some of the available layout managers in Java as well as learn how to arrange the components. There will be many exercises to give you experience in this regard.

Basic Interface Layout

If you've experimented with the interfaces we've crated up till now, you've undoubted realized that resizing your window distorts the presentation because the components move to palace on the container that we didn't intend. This is what we call being **FLUID** – ability to move and resize as its interface changes.

Why does Java allow this to happen? It is necessary for Java to support different platforms, (remember Platform Independence?), where there may be very small differences in how the platforms display things such as buttons, scrollbars and the like. By using Swing, we can gain more control over the interface with the use of Layout Managers. We should note however that because of Swing's platform independence nature and its flexibility, there is a hit on the performance factor causing an interface to slow down as well as presenting a foreign look and feel that doesn't compliment the operating system they are using (i.e. an interface for a Mac should have the same look and feel of a Mac).

Laying Out an Interface

A layout manager determines how components will be arranged when they are added to a container. The default layout manager for panels is the **FlowLayout** class. This class lets components flow from left to right in the order in which they were added to the container. When there is not enough room on a line, a new row of components are added immediately below the first in a left-to-right orientation. To create a layout manager for a container, we first call its constructor to create an instance of the class. For example, using the FlowLayout:

```
FlowLayout flo = new FlowLayout();
```

Laying Out an Interface

After we create a flow manager, we designate it as the layout manager for a container using the container’s **setLayout()** method. The layout manager must be set before any components are added to the container. If no layout manager is specified, the container’s default layout is used. The default is FlowLayout for panels and **BorderLayout** for frames. While not necessary to call the default layout directly when you want to use it, it is good coding practice to do so anyway to ensure there is no confusion as to what is being used.

Laying Out an Interface

The following snippet represent the starting point for a frame that uses a layout manger to control the arrangement of all the components that will be added to the frame:

```
import java.awt.*;

import javax.swing.*;

public class Starter extends JFrame{

public Starter() {super(“Example Frame”);

                    FlowLayout manager = new FlowLayout();

                    setLayout(manager);

                    // add the components here}

}
```

After the layout manager is set, we can start to add components to the container it is managing. Sometimes, order of addition is important (as in FlowLayout). Well see this when we work with the many general purpose layout managers provided by Java.

Flow Layout

FlowLayout is part of the **java.awt** package and is the simplest layout manager. It simply lays out the components in rows in a left-to-right fashion until it reaches the end of the row then wraps to the next row starting at the left most edge. By default, the components in each row are centered when we call the **FlowLayout()** constructor with no arguments. If we want to align components, we can use the class variables **FlowLayout.LEFT** , **FlowLayout.RIGHT** or **FlowLayout.CENTER** as the constructor’s only argument as shown below:

```
FlowLayout  toTheRight = new FlowLayout(FlowLayout.RIGHT);
```

Flow Layout

FlowLayout(int, int, int) constructor takes three arguments – in order

Alignment – Must be one of five class variables: **CENTER, LEFT, RIGHT, LOEADING OR TRAILING**

Horizontal Gap – In pixels

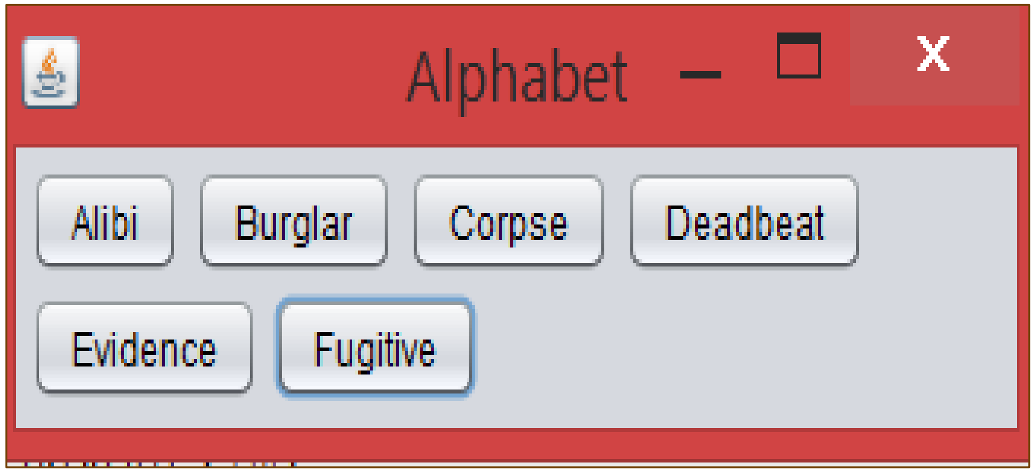
Vertical Gap – In pixels

An example of this format is below:

FlowLayout flo = new **FlowLayout(FlowLayout.CENTER, 30, 10)**;

Exercise 1: Alphabet.java, p. 225-6

This application called Alphabet.java displays six buttons arranged by the flow layout manager. Please code this module in Luna calling it Alphabet.java and test it as necessary ensuring that the results are achieved and that all paths have been tested.



Exercise 1: Alphabet.java

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

public class Alphabet extends JFrame {

  public Alphabet() {

    super("Alphabet");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setLookAndFeel();

    setSize(360, 120);

    FlowLayout lm = new FlowLayout(FlowLayout.LEFT);

    setLayout(lm);

    JButton a = new JButton("Alibi");

    JButton b = new JButton("Burglar");

    JButton c = new JButton("Corpse");

    JButton d = new JButton("Deadbeat");

    JButton e = new JButton("Evidence");

    JButton f = new JButton("Fugitive");
add(a);

    add(b);

    add(c);

    add(d);

    add(e);

    add(f);

    setVisible(true);

  }

private void setLookAndFeel() {

    try {

      UIManager.setLookAndFeel(

        "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"

      );
```

Alphabet.java

```
 Swing } catch (Exception exc) {
            System.err.println("Couldn't use the
system "
                + "look and feel: " + exc);
        }
    }
    public static void main(String[] arguments) {
        Alphabet frame = new
Alphabet();Utilities.updateComponentTreeUI(th
is);
}
}
```

Box Layout

The **BOX LAYOUT** is used to stack components from top to bottom or from left to right. It is managed by the **BoxLayout** class in the **javax.swing** package and improves on flow layout in that it ensures that components always line up a vertical or horizontal fashion. The alignment is specified with a class variable which can be **X_AXIS** for left-to-right horizontal alignment or **Y-AXIS** for top-to-bottom vertical alignment.

Box Layout

This is shown below:

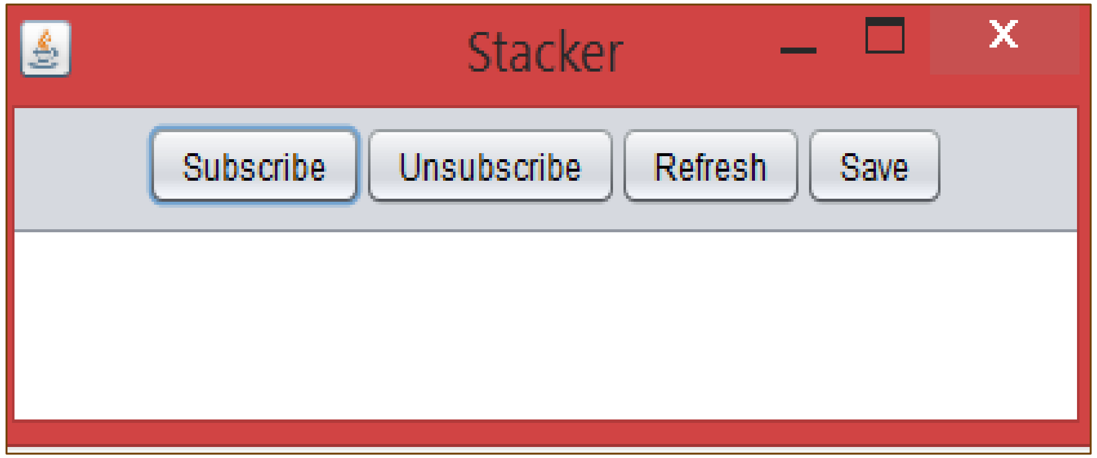
```
JPanel optionPane = new JPanel();

BoxLayout box = new BoxLayout(optionPane,

            BoxLayout.Y_AXIS);

optionPane.setLayout(box);
```

With this specification, components added to the container will line up accordingly on the specified axis and will be displayed at their preferred size. In the horizontal alignment, the manager attempts to give each component the same height were as in the vertical alignment, it attempts to give each the same width.

Exercise 1: Stacker.java, p. 227-8

In the Stacker.java application, a panel is used to contain buttons arranged with the box layout. The container is a JPanel and buttons along the top edge of the interface should be stacked horizontally.

Using Luna, code the following and run the application to completion. It should display the results as shown. Be sure to test all paths of execution (all buttons) to ensure that all code is tested.



Stacker.java

```
import java.awt.*;

import javax.swing.*;


public class Stacker extends JFrame {

    public Stacker() {

        super("Stacker");

        setSize(430, 150);


setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLookAndFeel();

// create top panel

        JPanel commandPane = new JPanel();

        BoxLayout horizontal = new BoxLayout(commandPane,

           BoxLayout.X_AXIS);

        commandPane.setLayout(horizontal);

        JButton subscribe = new JButton("Subscribe");

        JButton unsubscribe = new JButton("Unsubscribe");

        JButton refresh = new JButton("Refresh");

        JButton save = new JButton("Save");

        commandPane.add(subscribe);

        commandPane.add(unsubscribe);

        commandPane.add(refresh);

        commandPane.add(save);
```

Stacker.java

```
// create bottom panel
        JPanel textPane = new JPanel();
        JTextArea text = new JTextArea(4, 70);
        JScrollPane scrollPane = new JScrollPane(text);
        // put them together
        FlowLayout flow = new FlowLayout();
        setLayout(flow);
        add(commandPane);
        add(scrollPane);
        setVisible(true);
    }
private void setLookAndFeel() {
try {
         UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
         );
         SwingUtilities.updateComponentTreeUI(this);
      } catch (Exception exc) {
         System.err.println("Couldn't use the system "
            + "look and feel: " + exc);
      }
   }
    public static void main(String[] arguments) {
      Stacker st = new Stacker();
   }
}
```

Grid Layout

GRID LAYOUT arranges components into a grid of **ROWS AND COLUMNS** – shown below. Components are added first to the top row of the grid beginning with the left most **CELL** intersection of a row and column) and continuing to the right.

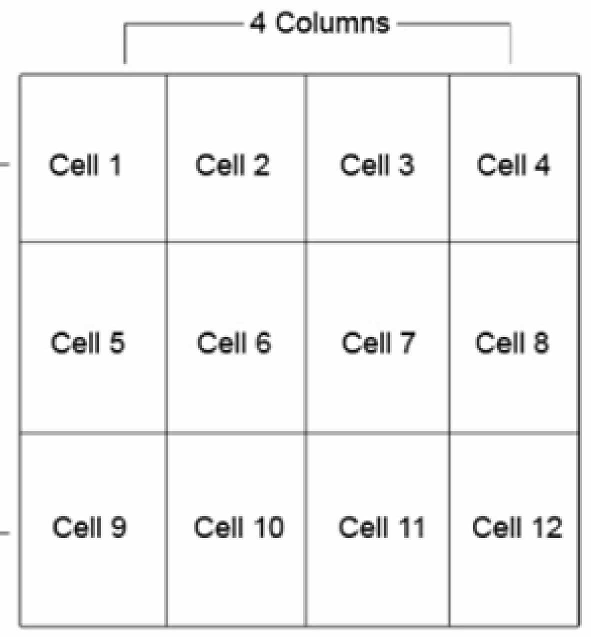


Figure: GridLayout Cell Arrangement

Grid Layout

When all of the cells are filled, the next component is added the left most cell of the next row. Grid Layout is created with the **GridLayout** class which belongs to the **java.awt** package. There are two arguments that can be set for the GridLayout constructor – the number of rows and the number of columns in the grid as shown in the following statement:

GridLayout gr = new GridLayout(10, 3);

We can specify the vertical and horizontal gap away from its default of 0 as we did with FlowLayout with two extra arguments or by calling setHgap() or setVgap(). The following statement creates a grid layout with 10 rows and 3 columns, a horizontal gap of 5 pixels and a vertical gap of 8 pixels:

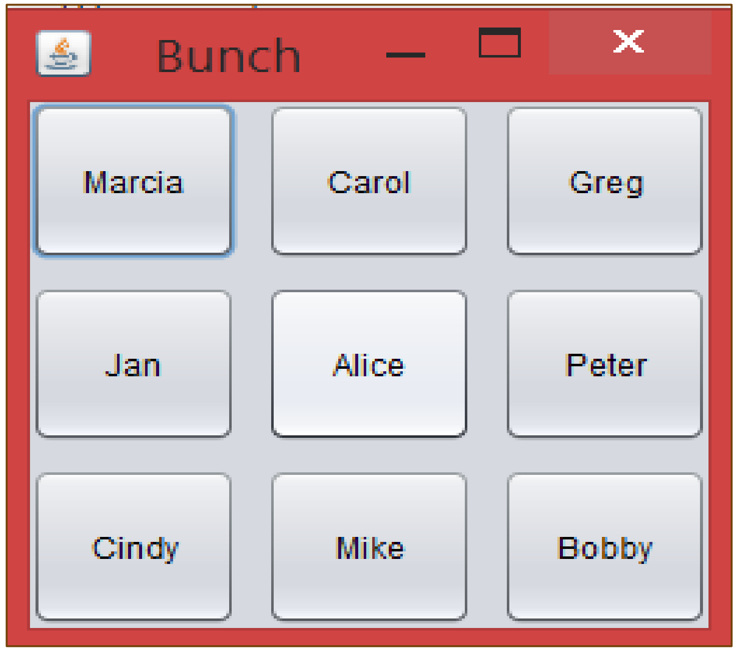
GridLayout gr2 = new GridLayout(10, 3, 5, 8);

An important difference between Grid and some of the other layout mangers is that each button will expand to fill the space available to them in each cell.

Exercise 1: Bunch.java

In this exercise, we create a grid with 3 rows and 3 columns and a 10-pixel gap between the components in both the horizontal and the vertical directions.

Using Luna, create the following module ensuring that all components are tested. It should be named Bunch.java.



Bunch.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Bunch extends JFrame {
public Bunch() {
        super("Bunch");
        setSize(260, 260);

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLookAndFeel();
        JPanel pane = new JPanel();
        GridLayout family = new GridLayout(3, 3, 10, 10);
        pane.setLayout(family);
JButton marcia = new JButton("Marcia");
        JButton carol = new JButton("Carol");
        JButton greg = new JButton("Greg");
        JButton jan = new JButton("Jan");
        JButton alice = new JButton("Alice");
        JButton peter = new JButton("Peter");
        JButton cindy = new JButton("Cindy");
        JButton mike = new JButton("Mike");
        JButton bobby = new JButton("Bobby");
```

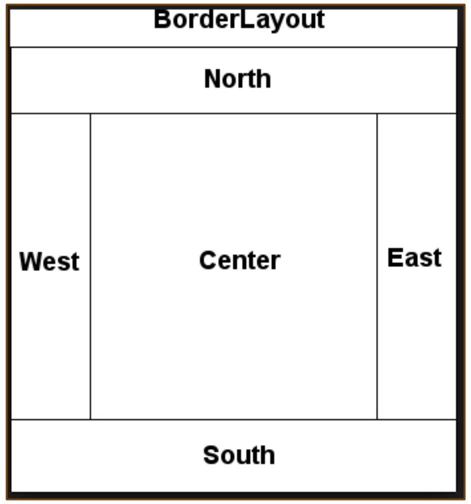
Bunch.java

```
pane.add(marcia);
        pane.add(carol);
        pane.add(greg);
        pane.add(jan);
        pane.add(alice);
        pane.add(peter);
        pane.add(cindy);
        pane.add(mike);
        pane.add(bobby);
        add(pane);
        setVisible(true);
    }
```

```
private void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
            );
            SwingUtilities.updateComponentTreeUI(this);
        } catch (Exception exc) {
            System.err.println("Couldn't use the system "
                + "look and feel: " + exc);
        }
    }
    public static void main(String[] arguments) {
        Bunch frame = new Bunch();
    }
}
```

Border Layout

BORDER LAYOUT is a bit more complex than the other layouts we've used.   This layout is created by the **BorderLayout** class in the **java.awt** package. BorderLayout divides the container into five sections: north, south, east, west and center.



When using this layout, the components represented by the directional labels fill their sections first and center gets what is left over. Usually we'll get 4 smaller components and one large one in the center since the preferred size of the components is not followed by the manager.

Border Layout

We create border layout with either the **BorderLayout()** or **BorderLayout(int, int)** constructors. The first creates a boarder layout with no gap while the second uses arguments to specify the horizontal and vertical gaps – must be in that order – we could also use setVgap() and setHgap().

After we create a boarder layout and establish it as the container's layout manger, we add components using a call to add() that is a bit different than the ones we've used previously: **add(Component, String).** The first argument is the component that should be added to the container. The second is a BorderLayout class variable that indicates the region where the component should be added. These class variables are: **NORTH, SOUTH, EAST, WEST** and **CENTER**. This is shown in the following snippet:

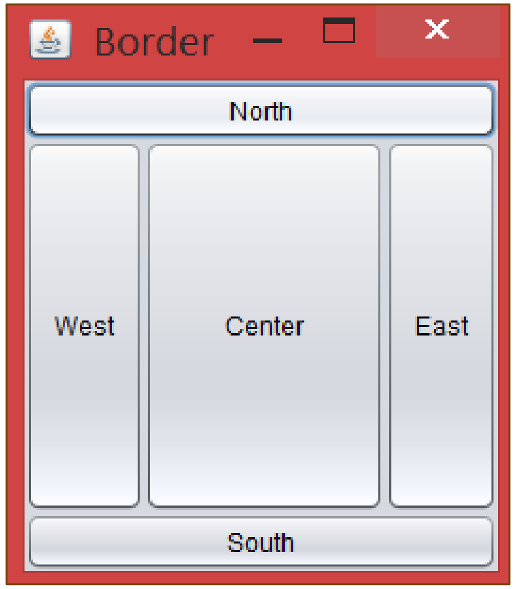
JButton quitButton = new JButton("quit");

add(quitButton, BorderLayout.NORTH);

Border.java, p. 232

In this exercise, we create a border layout application called Border.java to achieve a layout as shown above using buttons to indicate the regions of the layout: North, South, East, West and Center.

Using Luna, code the following module naming it Border.java. Please test each component. The results are below.



Border.java

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

public class Border extends JFrame {

   public Border() {
      super("Border");
      setSize(240, 280);
setDefaultCloseOperation(JFrame.EXIT_ON_CLO
SE);
      setLookAndFeel();
      setLayout(new BorderLayout());
JButton nButton = new JButton("North");
      JButton sButton = new JButton("South");
      JButton eButton = new JButton("East");
      JButton wButton = new JButton("West");
      JButton cButton = new JButton("Center");
      add(nButton, BorderLayout.NORTH);
      add(sButton, BorderLayout.SOUTH);
      add(eButton, BorderLayout.EAST);
      add(wButton, BorderLayout.WEST);
      add(cButton, BorderLayout.CENTER);
      setVisible(true);
   }
```

Mixing Layout Managers

We often combine layout managers to achieve the desired results. We do this by putting several containers inside a larger container and giving each its own layout. The container to be used inside of a larger container is the panel which as we know is created from the **JPanel** class in the **java.awt** package. These panels are simple containers used to group components. There are two things that must be kept in mind:

Panels must be filled with the components before it can be placed in the larger container

Panel has its own layout manager

We create panels with a call to the JPanel class: **JPanel pane = new JPanel();**

Mixing Layout Managers

We set the layout method for the panel by calling setLayout() on that panel as shown in the following snippet where we create a layout manager and apply it to a JPanel object called pane:

```
FlowLayout flo = new FlowLayout();
```

```
Pane.setLayout(flo);
```

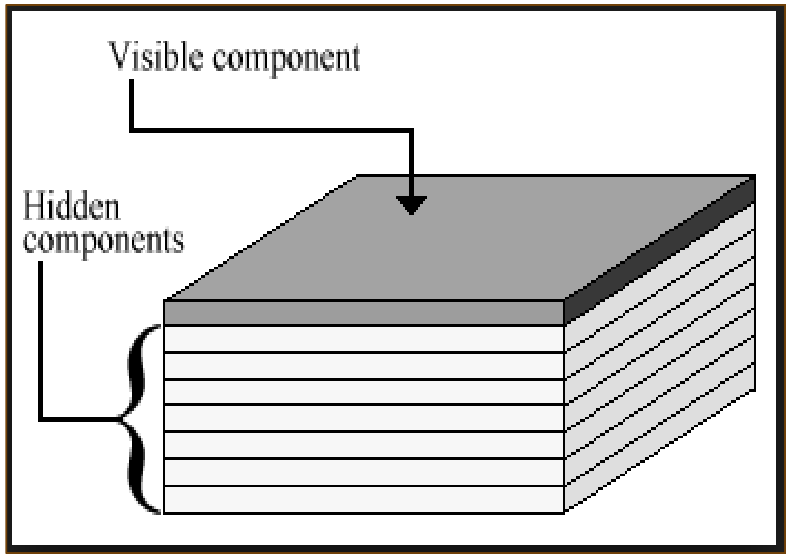
We add components by calling the panels add() method which is the same for panels as it is for containers. This is shown in the following snippet to create a text field and add it to a JPanel object called pane:

```
JTextField nameField = new JTextField(80);
```

```
pane.add(nameField);
```

Card Layout

With **CARD LAYOUT**, some of the components are hidden from view. A card layout is a group of containers or components displayed one at a time in much the same way as a card dealer reveals one card at a time from a deck. Each container in the group is called a card – hence the name Card Layout. We see these in installation wizards where each step in the process has its own 'card'. Often times, a next button advances to the next card.



Card Layout

Of course we have to use setLayout() to make card layout the manager for our container. After we set the container to use Card layout, we must use the add(Component, String) method to add the components. Component specifies the container or component that serves as the card. If it is a container, all components must have been added to it before the card is added. String names the card. This can be set to any value we would like such as Card1, Next or Finish. The following statement adds a panel object named options to a container and names the card “Options Card”:

```
add(options, “Options Card”);
```

Card Layout

When a container using card layout is displayed for the first time, the visible card is the first added to the container. Subsequent cards are displayed by calling the show() method of the layout manager which has two arguments: the container and the name of the card. For example:

cc.show(this, “Fact Card”);

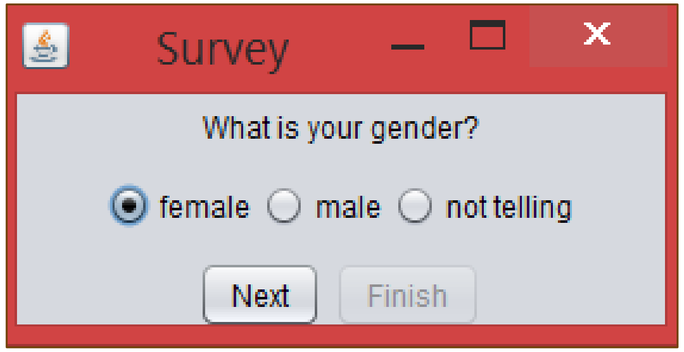
The this keyword would be used in a frame governed by card layout and refers to the object inside which the cc.show() statement appears. When a card is shown, the previously displayed card is hidden automatically – only one card can be shown at a time.

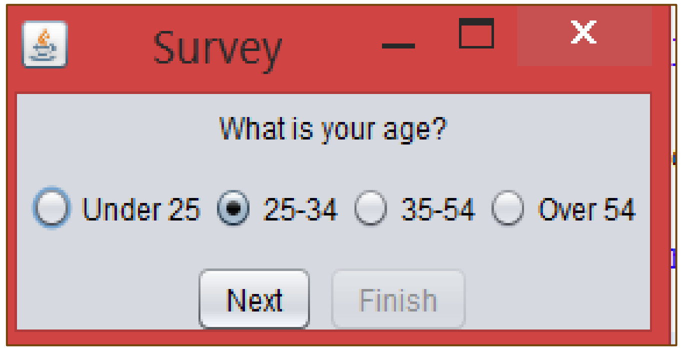
In a program that uses a card layout manager, a card change generally is triggered by a user’s actions as in an installation program where the user would choose Next to see the next step in the process.

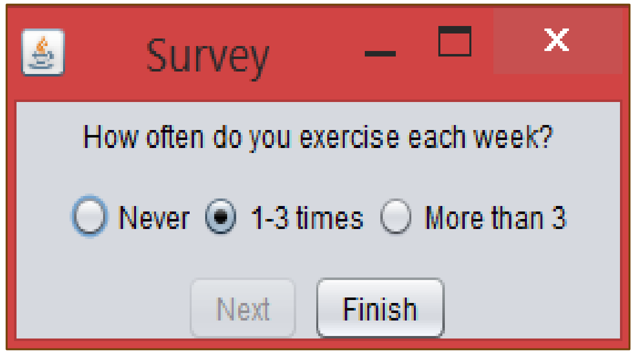
Exercise 1. SurveyWizard.java and SurveyFrame.java, p. 235-7

In this project the card layout is used along with several different layout managers within the same GUI.

Please code the following 2 modules and Run SurveyFrame.java application to kick off the application. Be sure to test all paths of execution.






Survey
How often do you exercise each week?
Never [✓ 1-3 times] More than 3
Next
Finish

SurveyWizard.java

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;


public class SurveyWizard extends JPanel implements
ActionListener {

   int currentCard = 0;

   CardLayout cards = new CardLayout();

   SurveyPanel[] ask = new SurveyPanel[3];


   public SurveyWizard() {

      super();

      setSize(240, 140);

      setLayout(cards);

// set up survey

      String question1 = "What is your gender?";

      String[] responses1 = { "female", "male", "not telling" };

      ask[0] = new SurveyPanel(question1, responses1, 2);

      String question2 = "What is your age?";

      String[] responses2 = { "Under 25", "25-34", "35-54",

         "Over 54" };

      ask[1] = new SurveyPanel(question2, responses2, 1);

      String question3 = "How often do you exercise each week?";

      String[] responses3 = { "Never", "1-3 times", "More than 3" };

      ask[2] = new SurveyPanel(question3, responses3, 1);

      ask[2].setFinalQuestion(true);

      addListeners();

   }
```

SurveyWizard.java

```
private void addListeners() {
    for (int i = 0; i < ask.length; i++) {
        ask[i].nextButton.addActionListener(this);
        ask[i].finalButton.addActionListener(this);
        add(ask[i], "Card " + i);
    }
  }
  public void actionPerformed(ActionEvent evt) {
    currentCard++;
    if (currentCard >= ask.length) {
        System.exit(0);
    }
    cards.show(this, "Card " + currentCard);
  }
}
```

```
class SurveyPanel extends JPanel {
  JLabel question;
  JRadioButton[] response;
  JButton nextButton = new JButton("Next");
  JButton finalButton = new JButton("Finish");
  SurveyPanel(String ques, String[] resp, int def) {
    super();
    setSize(160, 110);
    question = new JLabel(ques);
    response = new JRadioButton[resp.length];
    JPanel sub1 = new JPanel();
    ButtonGroup group = new ButtonGroup();
    JLabel quesLabel = new JLabel(ques);
    sub1.add(quesLabel);
    JPanel sub2 = new JPanel();
```

SurveyWizard.java

```
for (int i = 0; i < resp.length; i++) {
      if (def == i) {
        response[i] = new JRadioButton(resp[i], true);
      } else {
        response[i] = new JRadioButton(resp[i], false);
      }
      group.add(response[i]);
      sub2.add(response[i]);
    }
    JPanel sub3 = new JPanel();
    nextButton.setEnabled(true);
    sub3.add(nextButton);
    finalButton.setEnabled(false);
    sub3.add(finalButton);
```

```
GridLayout grid = new GridLayout(3, 1);
    setLayout(grid);
    add(sub1);
    add(sub2);
add(sub3);
  }

  void setFinalQuestion(boolean finalQuestion) {
    if (finalQuestion) {
      nextButton.setEnabled(false);
      finalButton.setEnabled(true);
    }
  }
}
```

SurveyFrame.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.*;
import javax.swing.*;
public class SurveyFrame extends JFrame {
  public SurveyFrame() {
    super("Survey");
    setSize(290, 140);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLookAndFeel();
    SurveyWizard wiz = new SurveyWizard();
    add(wiz);
    setVisible(true);
  }
private void setLookAndFeel() {
    try {
      UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
      );
      SwingUtilities.updateComponentTreeUI(this);
    } catch (Exception exc) {
      System.err.println("Couldn't use the system "
        + "look and feel: " + exc);
    }
  }
  public static void main(String[] arguments) {
    SurveyFrame surv = new SurveyFrame();
  }
}
```

Grid Bag Layout

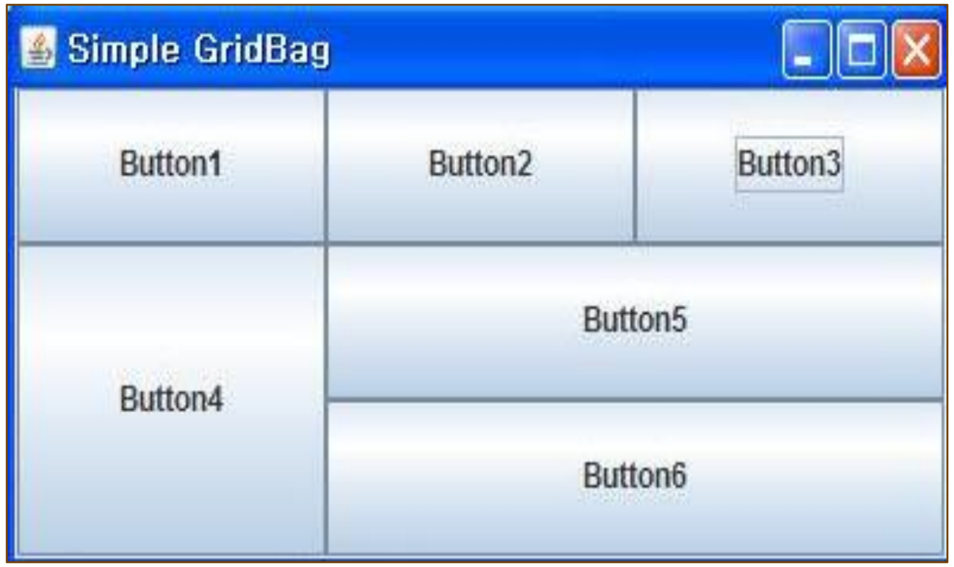
The final layout manager is **GRID BAG** – an extension to grid. We still get rows and columns however we get the following benefits:

- A component can take up more than one cell in the grid

- Proportions between the rows and columns do not have to be equal

- Component does not have to fill the entire cell(s) it occupies

- Component can be aligned along the edge of any cell



Grid Bag Layout

Grid Bag requires the **GridBagLayout** and **GridBagConstraints** classes which below to the **java.awt** package. GridBagLayout is the layout manager and GridBagConstraints defines the placement of the components on the screen.

The constructor for grid bag takes no arguments and can be applied to a container like any other manager. The following statements could be used in a frame's constructor method to use grid bag in the container:

GridBagLayout bag = GridBagLayout();

setLayout(bag);

In grid bag, each component uses a GridBagConstraints object to define the cell(s) it occupies in the grid, its size and other aspects of its presentation.

Grid Bag Layout

There are 11 instance variables associated with GridBagConstraints to determine the component placement:

gridx – The x position of the cell that holds the component – upper-left portion of the component

gridy – The y position of the cell or upper-left portion

gridwidth – The number of cells the component occupies in the horizontal direction

gridheight – The number of cells the component occupies in a vertical direction

weightx – A value that indicates the component's size relative to other components on the same row of the grid

weighty – A value that indicates the component's size relative to other components on the same column on the grid

anchor – A value that determines where the component is displayed within its cell

fill – A value that determines whether the component expands horizontally or vertically to fill its cell

insets – An Insets object that sets the white space around the component inside the cell

ipadx – The amount by which to expand the component's width beyond its minimum size

ipady – The amount by which to expand the component's height

All but the insets can hold an integer value.

Grid Bag Layout

The easiest way to use the class is to create a constraint with no arguments and set its variables individually as shown in the following snippet:

```
GridBagLayout gridbag = new GridBagLayout();
GridBagContstraints constraint = new GridBagConstraings();
setLayout(gridbag);
constraint.gridx = 0;
constraint.gridy = 0;
constraint.gridwidth = 2;
constraint.gridheight = 1;
constraint.weightx = 100;
constraint.weighty = 100;
constraint.fill = GridBagConstraints.NONE;
constraint.anchor = GridBagConstraints.CENTER;
```

Grid Bag Layout

This code creates a grid bag to put components at position (0, 0) that is two cells wide and one cell tall. The component is added to the grid bag layout in two steps:

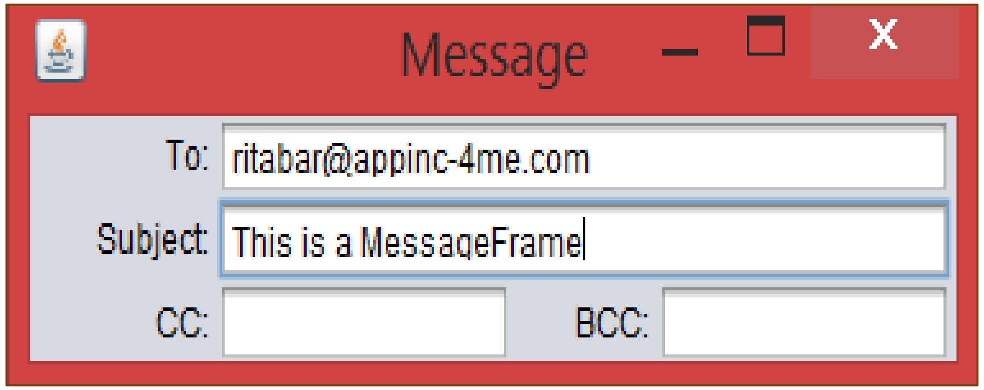
The layout manager’s setConstraints(Component, GridBagConstraints) method is called with the component and constraint arguments

The component is added to the container that uses the manager

Exercise 1: MessagePanel.java and MessageFrame.java

In this exercise we create a simple email interface program using a grid bag layout. Since there is not a main() in MessagePanel.java, Run MessageFrame.java to kick off the application.

Using Luna create and test the modules shown below to achieve the following results.



To: ritabar@appinc-4me.com

Subject: This is a MessageFrame

CC:

BCC:

MessagePanel.java

```
import java.awt.*;
import javax.swing.*;
public class MessagePanel extends JPanel {
  GridBagLayout gridbag = new GridBagLayout();
  public MessagePanel() {
    super();
    GridBagConstraints constraints;
    setLayout(gridbag);
    JLabel toLabel = new JLabel("To: ");
    JTextField to = new JTextField();
JLabel subjectLabel = new JLabel("Subject: ");
    JTextField subject = new JTextField();
    JLabel ccLabel = new JLabel("CC: ");
    JTextField cc = new JTextField();
    JLabel bccLabel = new JLabel("BCC: ");
    JTextField bcc = new JTextField();
addComponent(toLabel, 0, 0, 1, 1, 10, 100,
      GridBagConstraints.NONE, GridBagConstraints.EAST);
    addComponent(to, 1, 0, 9, 1, 90, 100,
      GridBagConstraints.HORIZONTAL,
GridBagConstraints.WEST);
    addComponent(subjectLabel, 0, 1, 1, 1, 10, 100,
      GridBagConstraints.NONE, GridBagConstraints.EAST);
    addComponent(subject, 1, 1, 9, 1, 90, 100,
```

MessagePanel.java

```
GridBagConstraints.HORIZONTAL, GridBagConstraints.WEST);
        addComponent(ccLabel, 0, 2, 1, 1, 10, 100,
            GridBagConstraints.NONE, GridBagConstraints.EAST);
        addComponent(cc, 1, 2, 4, 1, 40, 100,
            GridBagConstraints.HORIZONTAL,
GridBagConstraints.WEST);
        addComponent(bccLabel, 5, 2, 1, 1, 10, 100,
            GridBagConstraints.NONE, GridBagConstraints.EAST);
        addComponent(bcc, 6, 2, 4, 1, 40, 100,
            GridBagConstraints.HORIZONTAL,
GridBagConstraints.WEST);
private void addComponent(Component component, int gridx, int
gridy,
        int gridwidth, int gridheight, int weightx, int weighty, int fill,
        int anchor) {
        GridBagConstraints constraints = new GridBagConstraints();
        constraints.gridx = gridx;
        constraints.gridy = gridy;
        constraints.gridwidth = gridwidth;
        constraints.gridheight = gridheight;
        constraints.weightx = weightx;
        constraints.weighty = weighty;
        constraints.fill = fill;
        constraints.anchor = anchor;
        gridbag.setConstraints(component, constraints);
        add(component);
    }
}
```

MessageFrame.java

```
import javax.swing.*;
public class MessageFrame extends JFrame {
  public MessageFrame() {
    super("Message");
    setSize(380, 120);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLookAndFeel();
    MessagePanel mPanel = new MessagePanel();
    add(mPanel);
    setVisible(true);
private void setLookAndFeel() {
    try {
      UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
      );
      SwingUtilities.updateComponentTreeUI(this);
    } catch (Exception exc) {
      System.err.println("Couldn't use the system "
        + "look and feel: " + exc);
    }
  }
  public static void main(String[] arguments) {
    MessageFrame frame = new MessageFrame();
  }
}
```

Cell Padding and Inserts

In our preceding exercise, we did not use the three GridBagConstraints variables **insets, ipadx** and **ipady**. The ipadx and ipady constraints control padding which is the extra space around the individual components. By default, there is not extra space. The variable ipadx adds space to either side of the component and the ipady adds it above and below. These are also used to determine the amount of space between components in the panel. Insets are used to determine the amount of space around the panel itself. Insets include values for top, bottom, left and right.

Summary

When considering the Swing interface, we learned how to paint a user interface on a Java application window using the components of the Swing package. These included buttons, bars, lists and fields as well as advanced features such as sliders, dialog boxes, progress bars and menu bars. We implemented interface components by creating instances of their class and adding them to containers such as frame.

When we think about the layout managers, we have the ability to build very sophisticated interfaces for our users that work on a variety of platforms. We covered the five layout managers and panels. We can nest containers within containers give each a different layout.

Swing offers many more user interface components that we can cover in our short time. Please visit Oracle's Javadoc site at http://docs.oracle.com/javase/7/docs/api/ then click the javax.swing package link to explore the further attributes of Swing.