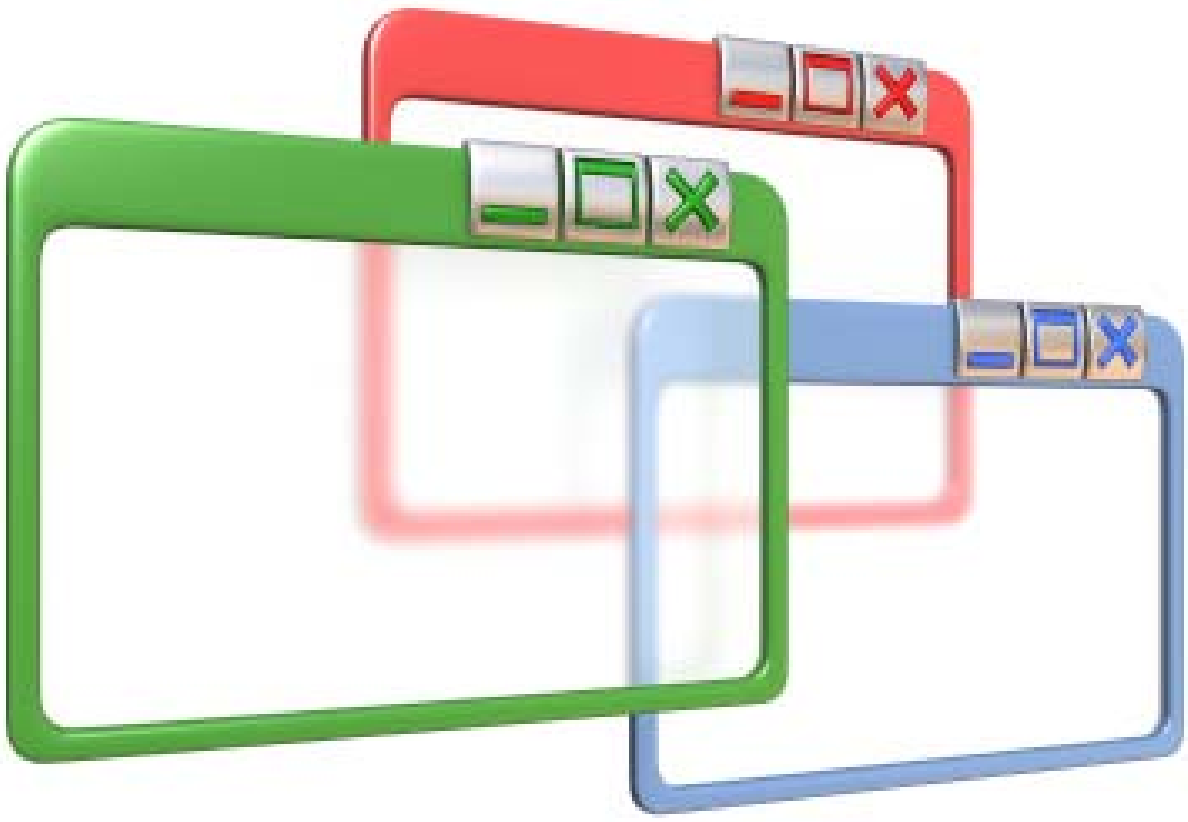
Chapter 7

User Interface Extensions

What is in This Chapter ?

This chapter discusses additional features that can be used to improve and extend your Graphical User Interfaces. It discusses the notion of Layout Managers in java which allow **automatic resizing** of components on the window. The chapter also shows how to add **menus** to your user interfaces as well as develop your own **dialog boxes**.



7.1 Automatic Resizing Using Layout Managers

As you may know ... JAVA was developed for the internet and JAVA applications were initially meant to run as applets within an internet browser. Since browsers are often resized, the application's components need to be rearranged so that they ALL fit on the browser window at all times. In fact, JAVA provides a mechanism called a ***Layout Manager*** that allows the automatic arrangement (i.e., layout) of the components of an application as the window is resized.

Why should we use a Layout Manager ?

- we would not have to compute locations and sizes for our components
- our components will resize automatically when the window is resized
- our interface will appear "nicely" on all platforms

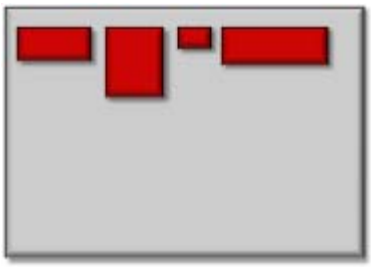
In JAVA, each layout manager defines methods necessary for a class to be able to arrange **Components** within a **Container.** There are 6 commonly used layout manager classes that implement the **LayoutManager** interface:

FlowLayout, **BoxLayout**, **BorderLayout**, **CardLayout**, **GridLayout**, and **GridBagLayout**

Layouts are set for a panel using the **setLayout()** method. If set to **null**, then no layout manager is used. This is what we have been doing up until this point.

Let us now look at each of these layout managers in turn.

*Example (**FlowLayout**):*



The simplest layout manager is the **FlowLayout**. It is commonly used to arrange just a few components on a panel. With this manager, components on the window (e.g., buttons, text fields, etc..) are arranged horizontally from left to right ... like lines of words in a paragraph written in English. If no space remains on the current line, components flow (or wrap around) to the next "line". The height of each line is the maximum height of any component on that line. By default, components are centered horizontally on each line, but this can be changed.

There are three constructors that can be used to create a **FlowLayout** manager:

```
public FlowLayout();
public FlowLayout(int align);
public FlowLayout(int align, int hGap, int vGap);
```

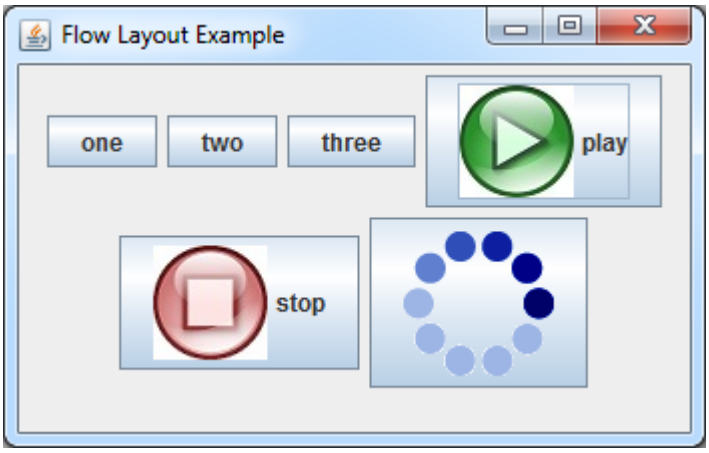
Here, **align** specifies how the components are to be justified horizontally. It may be any one of three constants:

FlowLayout.LEFT, **FlowLayout.RIGHT** or **FlowLayout.CENTER**

Also, **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. Here is a simple example that adds 6 buttons (3 with icons) to a panel which uses a **FlowLayout**.

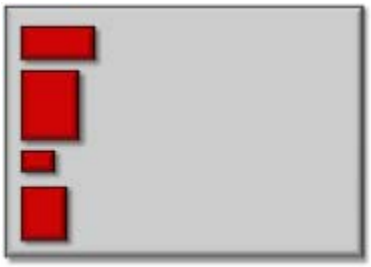
```
import java.awt.*;
import javax.swing.*;

public class FlowLayoutExample extends JFrame {
    public FlowLayoutExample(String title) {
        super(title);
        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
        getContentPane().add(new JButton("one"));
        getContentPane().add(new JButton("two"));
        getContentPane().add(new JButton("three"));
        getContentPane().add(new JButton("play", new ImageIcon("GreenButton.jpg")));
        getContentPane().add(new JButton("stop", new ImageIcon("RedButton.jpg")));
        getContentPane().add(new JButton(new ImageIcon("Progress.gif")));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 200);
    }
    public static void main(String[] args) {
        new FlowLayoutExample("Flow Layout Example").setVisible(true);
    }
}
```

Notice that we can place an image onto a component as an **ImageIcon** object, passing in the name of the **gif** or **jpg** file ... provided that it is in the same directory that this code is running in. Even animated gif files can be used. Here is the result obtained when the application window is resized in different ways ... take notice of how the components wrap around to the next "line":



Keep in mind that the above example just places JButtons as the components, however any components can be used here.

*Example (**BoxLayout**):*



The **BoxLayout** is also very simple to use. It is similar to the **FlowLayout** in that it arranges components one after another. However, it does not have a wrap around effect. Instead, any components that do not fit on the line are simply not shown. Also, a **BoxLayout** allows you to arrange the components horizontally or vertically.

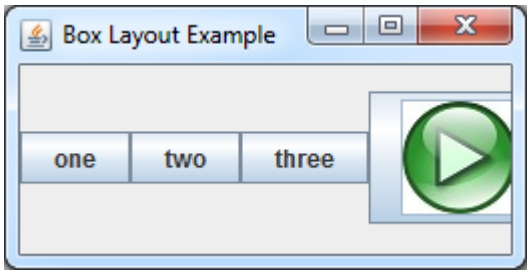
There is one constructor that can be used to create a **BoxLayout** manager:

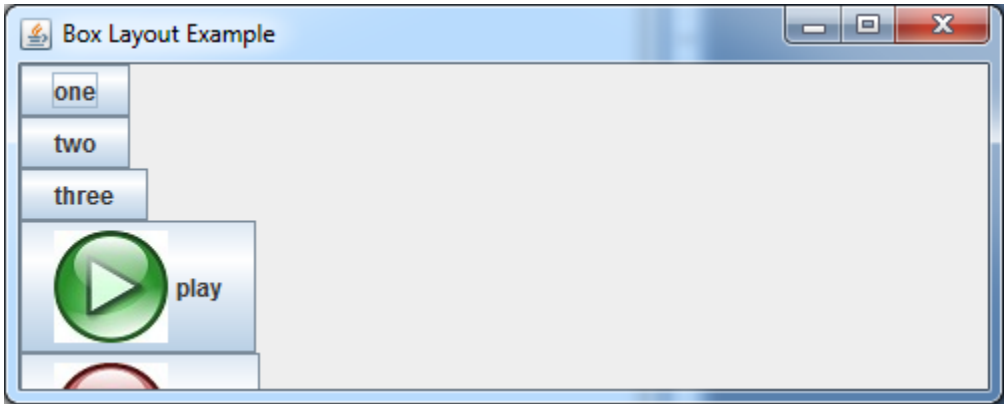
```
public BoxLayout(Container panel, int axis);
```

Here, **axis** specifies how the components are to be justified ... either horizontally by using **BoxLayout.X_AXIS** or vertically by using **BoxLayout.Y_AXIS**. The **panel** parameter is the panel that contains the components (i.e., the panel on which we are applying this layout manager). Here is a similar example to that from the **FlowLayout** example:

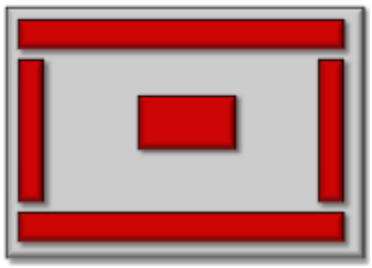
```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample extends JFrame {
    public BoxLayoutExample(String title) {
        super(title);
        getContentPane().setLayout(new BoxLayout(this.getContentPane(),
                                                 BoxLayout.X_AXIS));
        getContentPane().add(new JButton("one"));
        getContentPane().add(new JButton("two"));
        getContentPane().add(new JButton("three"));
        getContentPane().add(new JButton("play", new ImageIcon("GreenButton.jpg")));
        getContentPane().add(new JButton("stop", new ImageIcon("RedButton.jpg")));
        getContentPane().add(new JButton(new ImageIcon("Progress.gif")));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 200);
    }
    public static void main(String[] args) {
        new BoxLayoutExample("Box Layout Example").setVisible(true);
    }
}
```

Here is the result obtained when the application window is resized in different ways ... take notice of how the components DO NOT wrap around to the next "line". The first two snapshots below represent an X_AXIS layout while the bottom and right one represent a Y_AXIS layout:





*Example (**BorderLayout**):*



The **BorderLayout** is a useful layout manager. Instead of re-arranging components, it allows you to place components at one of 5 anchored positions on the window (i.e., north, south, east, west or center). As the window resizes, components stay "anchored" to the side of the window or to its center. The components will grow accordingly. You may place at most one component in each of the 5 anchored positions ... but this one component may be a container such as a **JPanel** that contains other components inside of it. Typically, you do NOT place a component in each of the 5 areas, but choose just a few of the areas.

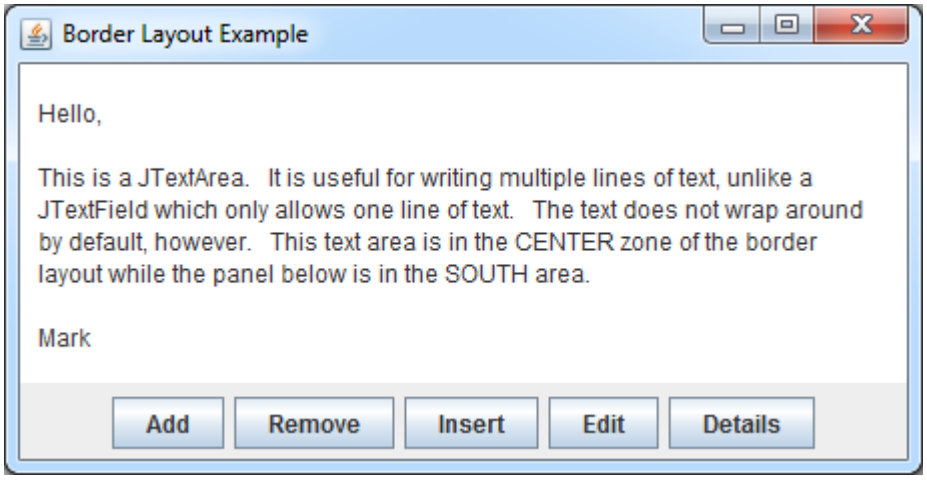
There are two constructors that can be used to create a **BorderLayout** manager:

```
public BorderLayout()
public BorderLayout(int hgap, int vgap);
```

As with the **FlowLayout**, the **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. When adding components, the **add()** method requires a 2nd parameter indicating the area to add to which must be one of **BorderLayout.NORTH**, **BorderLayout.SOUTH**, **BorderLayout.EAST**, **BorderLayout.WEST** or **BorderLayout.CENTER**. Here is a simple example that adds a **JTextField** to the CENTER and a **JPanel** with buttons to the SOUTH.

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

public class BorderLayoutExample extends JFrame {
    public BorderLayoutExample(String title) {
        super(title);

        getContentPane().setLayout(new BorderLayout(2,2));

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(new JButton("Add"));
        buttonPanel.add(new JButton("Remove"));
        buttonPanel.add(new JButton("Insert"));
        buttonPanel.add(new JButton("Edit"));
        buttonPanel.add(new JButton("Details"));
        getContentPane().add(BorderLayout.SOUTH, buttonPanel);
        getContentPane().add(BorderLayout.CENTER, new JTextArea());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 300);
    }
    public static void main(String[] args) {
            new BorderLayoutExample("Border Layout Example").setVisible(true);
    }
}
```

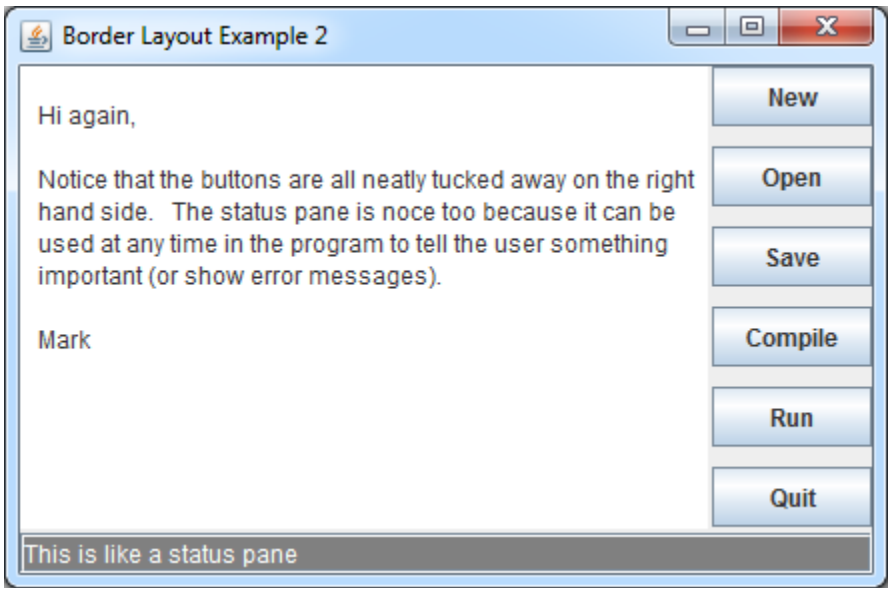
Here is the result:

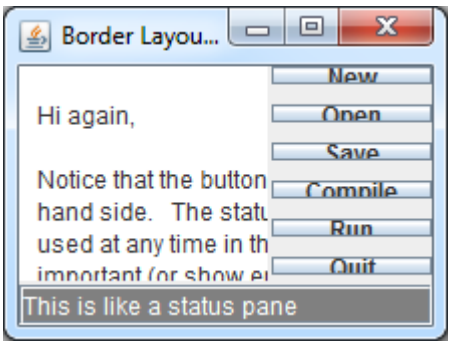


We can also make some buttons on the right hand side. Here is an example with a status pane at the bottom as well as a **JPanel** of buttons on the right:

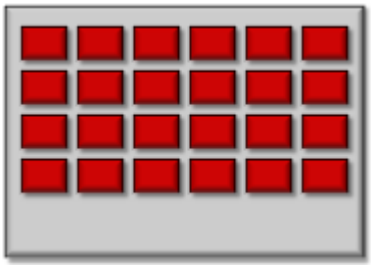
```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutExample2 extends JFrame {
    public BorderLayoutExample2(String title) {
        super(title);

        getContentPane().setLayout(new BorderLayout(2,2));

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(6,1,10,10));
        buttonPanel.add(new JButton("New"));
        buttonPanel.add(new JButton("Open"));
        buttonPanel.add(new JButton("Save"));
        buttonPanel.add(new JButton("Compile"));
        buttonPanel.add(new JButton("Run"));
        buttonPanel.add(new JButton("Quit"));
        getContentPane().add(BorderLayout.EAST, buttonPanel);

        JTextField statusPane = new JTextField("This is like a status pane");
        statusPane.setBackground(Color.gray);
        statusPane.setForeground(Color.white);
        getContentPane().add(BorderLayout.SOUTH, statusPane);
        getContentPane().add(BorderLayout.CENTER, new JTextArea());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 300);
    }
    public static void main(String[] args) {
            new BorderLayoutExample2("Border Layout Example 2").setVisible(true);
    }
}
```

Here is the output as the window is resized ... the resizing behavior may not be nice:





*Example (**GridLayout**):*



A GridLayout is excellent for arranging a 2-dimensional grid of components (such as buttons on a keypad). It automatically aligns the components neatly into rows and columns. Typically, the components are all of the same size, however you can add different sized components as well. Components are added in sequence one after another until the grid has been filled.

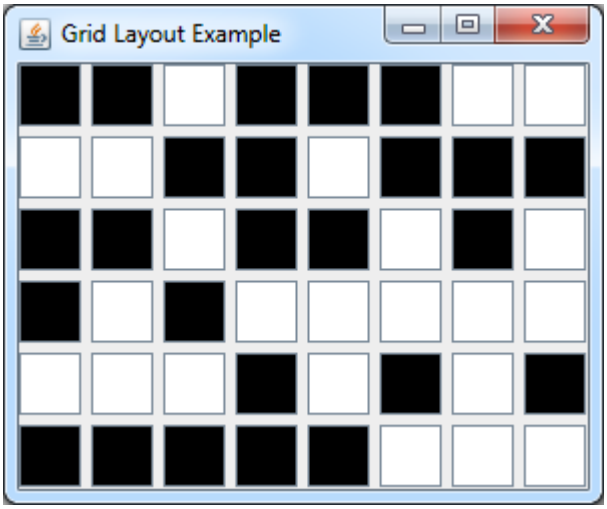
There are two constructors that can be used to create a **GridLayout** manager:

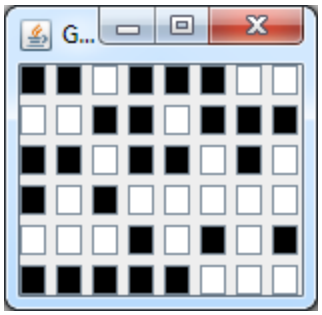
```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns, int hGap, int vGap)
```

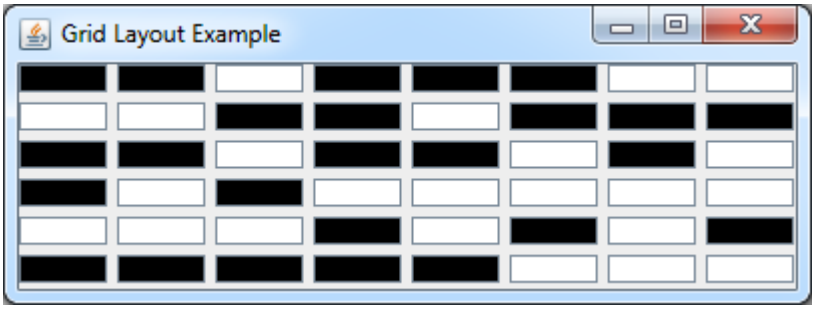
The **rows** and **columns** parameters tell JAVA how many rows and columns to use for the grid. Again, the **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. When adding components, the **add()** method does not allow you to specify which row and column the component will reside in. Instead, JAVA forces you to add the components one by one and it automatically places them in the grid starting at the top left and moving horizontally until a row is completed ... then moving down to the next row. Here is a simple example that adds some buttons with random background colors of white or black:

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutExample extends JFrame {

    public GridLayoutExample(String title) {
        super(title);

        getContentPane().setLayout(new GridLayout(6,8,5,5));

        for (int row=1; row<=6; row++)
            for (int col=1; col<=8; col++) {
                JButton b = new JButton();
                if (Math.random() < 0.5)
                    b.setBackground(Color.black);
                else
                    b.setBackground(Color.white);
                getContentPane().add(b);
            }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 250);
    }
    public static void main(String[] args) {
        new GridLayoutExample("Grid Layout Example").setVisible(true);
    }
}
```

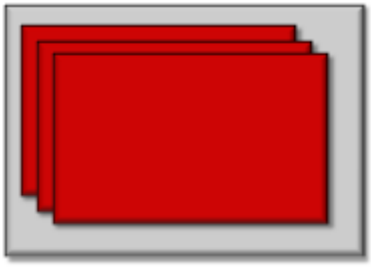
Here is the result showing a few different resizings:







*Example (**CardLayout**):*



The **CardLayout** manager is a little different from the others. It is used to simulate a kind of slideshow because it allows you to have many components in the container but to show only one at a time. A typical application would be to place ImageIcons on a set of labels and have the user click buttons to cycle through the images like a slideshow. In addition to image swapping, you can actually have entire panels swap in and out to completely alter the user interface at the "drop of a hat" as the expression goes.

There are two constructors that you can use:

```
public CardLayout()
public CardLayout(int hgap, int vgap)
```

Once the layout manager has been created, it allows you to jump around to various cards (i.e., slides) like a slide projector. Here are the various methods available (the **owner** container is the panel that this layout manager has been applied to):

```
public void first(Container owner)
public void next(Container owner)
public void previous(Container owner)
public void last(Container owner)
public void show(Container owner, String name)
```

Notice that the last method allows you to jump to a specific card/slide, which is uniquely identified by a name. Therefore, when we add cards/slides to the layout, we will supply a unique name for each one.

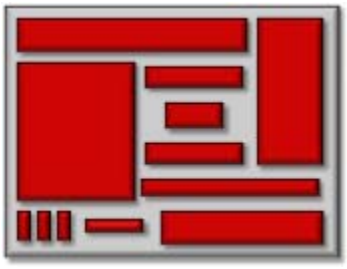
Here is an example that cycles through 5 images as the user clicks on forward and reverse buttons:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.BasicArrowButton;

public class CardLayoutExample extends JFrame {
    JPanel      slides;
    CardLayout  layoutManager;

    public CardLayoutExample(String title) {
        super(title);

        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

        // Create a JPanel with a CardLayout manager for the slides
        slides = new JPanel();
        slides.setBackground(Color.WHITE);
        slides.setBorder(BorderFactory.createLineBorder(Color.BLACK));
        slides.setLayout(layoutManager = new CardLayout(0,0));
        slides.add("1", new JLabel(new ImageIcon("smallDog.jpg")));
        slides.add("2", new JLabel(new ImageIcon("dog1.jpeg")));
        slides.add("3", new JLabel(new ImageIcon("polarBear.jpg")));
        slides.add("4", new JLabel(new ImageIcon("hamsterweights.jpg")));
        slides.add("5", new JLabel(new ImageIcon("catSwim.jpg")));
        getContentPane().add(slides);

        // Now add some slide show buttons for forward and reverse
        JButton rev = new BasicArrowButton(JButton.WEST);
        getContentPane().add(rev);
        JButton fwd = new BasicArrowButton(JButton.EAST);
        getContentPane().add(fwd);

        // Set up the listeners using anonymous classes
        rev.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.previous(slides);
            }
        });

        fwd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.next(slides);
            }
        });
```

```
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500,500);
    }

    public static void main(String[] args) {
        new CardLayoutExample("Simple Slide Show").setVisible(true);
    }
}
```

Here is the result showing a few slides in sequence:

*Example (**GridBagLayout**):*



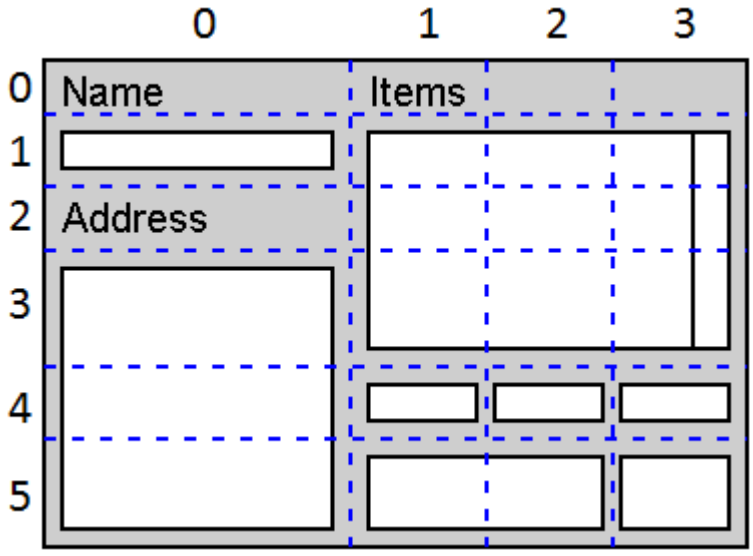
The **GridBagLayout** manager is absolutely the most flexible of all the layout managers. It allows you to be very specific in the placement of all components and to indicate exactly how each component is to resize as the window shrinks or grows. However, due to the flexibility of this layout manager, it is MUCH more complicated to use than any of the other layout managers.

The **GridBagLayout** also arranges components in a grid, but the grid rows and columns are not explicitly defined. Also, the rows and columns may have variable heights and widths. Each component can occupy (i.e., span) multiple rows and columns.

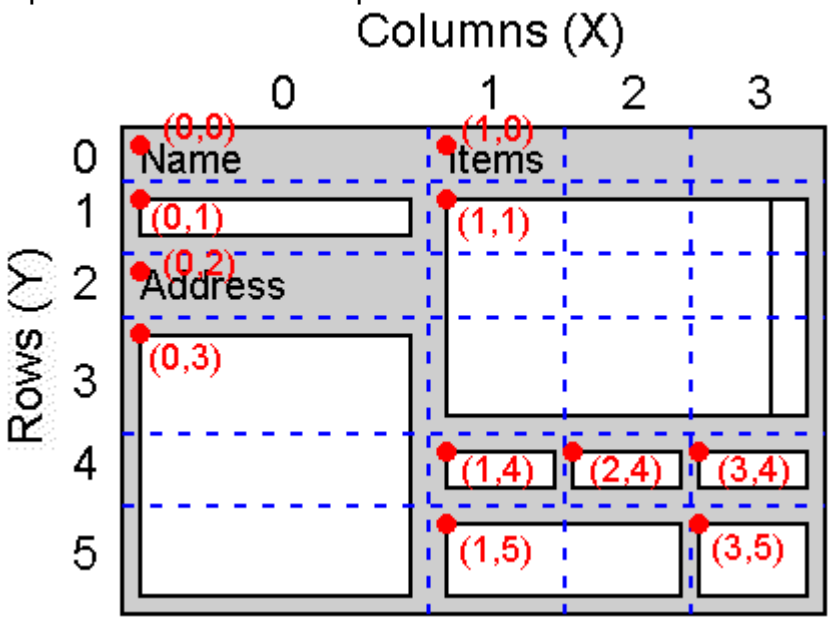
Since there are so many parameters for each component, JAVA supplies a **GridBagConstraints** object that is like a blank form that we must fill-out.

The **GridBagConstraints** objects are used to package together a set of constraints (i.e. parameters) for a particular component. Once the constraints are chosen, they must be set using the **setConstraints()** method for the component. Each constraint has a default which is automatically used if the constraint is not specified.

We will now take a look at the many constraints which we can use for each component. The first step when using a **GridBagLayout** is to determine the lines that separate the rows and the columns. We can do this by drawing horizontal and vertical lines whenever there is a change in components in that row or column. Then, we number the rows and columns starting at 0. Here is an example of a window showing the breakdown of the components onto such a grid:



Once we do this, we are ready to specify the parameters for each component. In the above example, there are 11 components (including the text components which are JLabels). For each of these components, we need to determine which grid cell (i.e., row and column numbers) that the top left corner of the component lies in as follows:



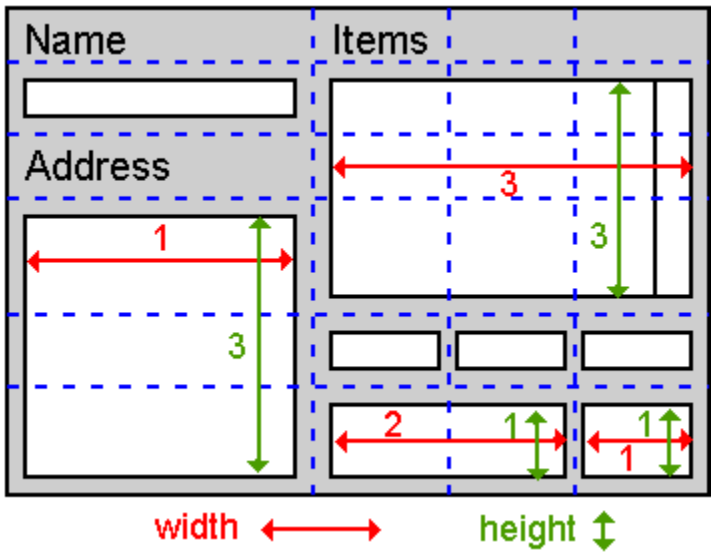
This value represents the first parameter that we must set for each component. They are called the **gridx** and **gridy** parameters.

Consider the items box (above) with the topleft corner at (1,1). Here is how we would begin to set the constraints for that single component. Notice that we set the layout manager for the

window's panel and then we start filling in the **constraints** object parameters **gridx** and **gridy**. Afterwards, we apply this to the component by using **setConstraints()**.

```
public class GridBagLayoutExample extends JFrame {
    public CardLayoutExample(String title) {
        super(title);

        GridBagLayout layout = new GridBagLayout();
        getContentPane().setLayout(layout);

        GridBagConstraints constraints = new GridBagConstraints();

        JList aList = new JList();
        constraints.gridx = 1;
        constraints.gridy = 1;
        ...

        layout.setConstraints(aList, constraints);
        getContentPane().add(aList);

        ...
    }
    ...
}
```

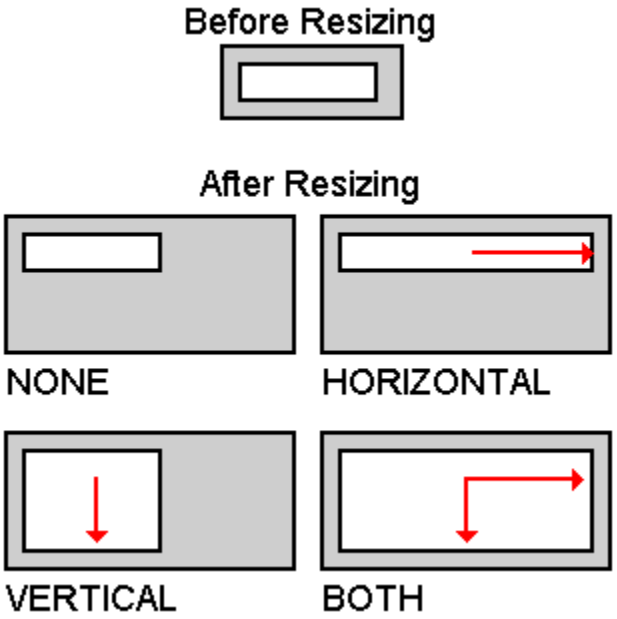
The next step is to determine how many rows and columns each component spans (i.e., takes up). Here are the values for some of the components:



Notice that some of the components take up only 1 row and 1 column (i.e., one grid cell). However, our list takes up space across 3 rows and 3 columns. We set this constraint as the **gridWidth** and **gridHeight** parameters as follows:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
...
```

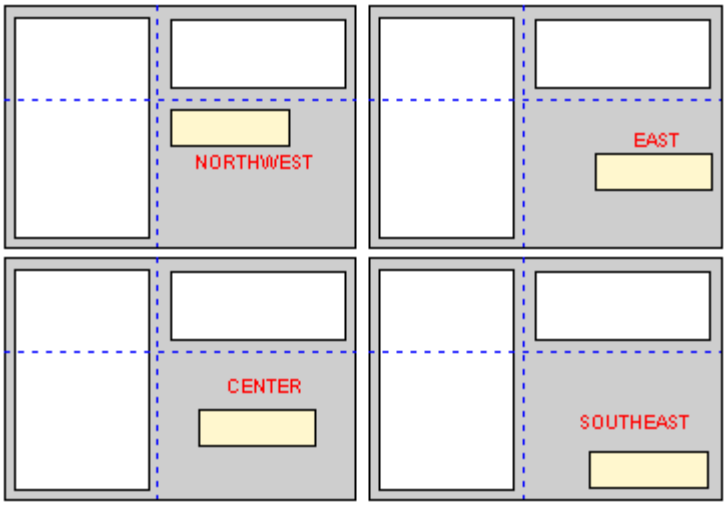
Now ... when the window is enlarged, we need to indicate whether or not the component is to stretch horizontally or vertically to fill up the extra space that becomes available. We can either specify not to fill in the extra space (i.e., NONE), to take up only the horizontal (i.e., HORIZONTAL) or vertical (i.e., VERTICAL) space ... or to take up all space vertically and horizontally (i.e., BOTH). Here is what will happen:



We need to set the **fill** parameter to one of these constants. In our list, we perhaps want the list to grow both vertically and horizontally when t he window grows, so we can set it to **GridBagConstraints.BOTH** as follows:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
...
```

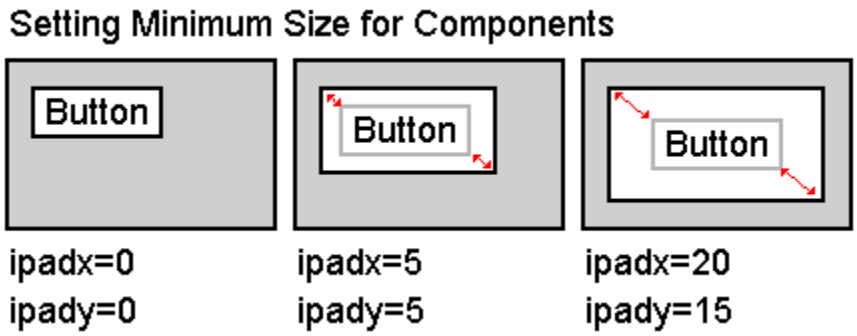
For components that do not have their **fill** set to **BOTH**, we need to indicate how the component will move around (or not move) within its cell when the window is resized. We do this by anchoring (i.e., tying) the component to the **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST** or **CENTER** of the cell using the **anchor** parameter→



For our list, since we have the fill set to BOTH, it does not matter what anchor we set. However, for something like the top/left label, we would want to anchor it using:

```
constraints.anchor = GridBagConstraints.NORTHWEST;
```

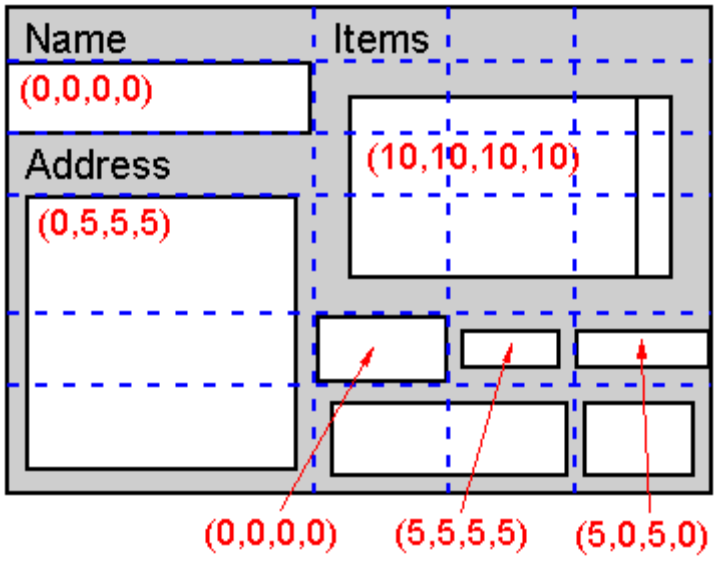
Now, for each component, we can fatten it by supplying a margin around it. There are parameters called **ipadx** and **ipady** that represent the internal padding (i.e., margin in pixels) around the inside of the component:



Here is how we can set this for our list:

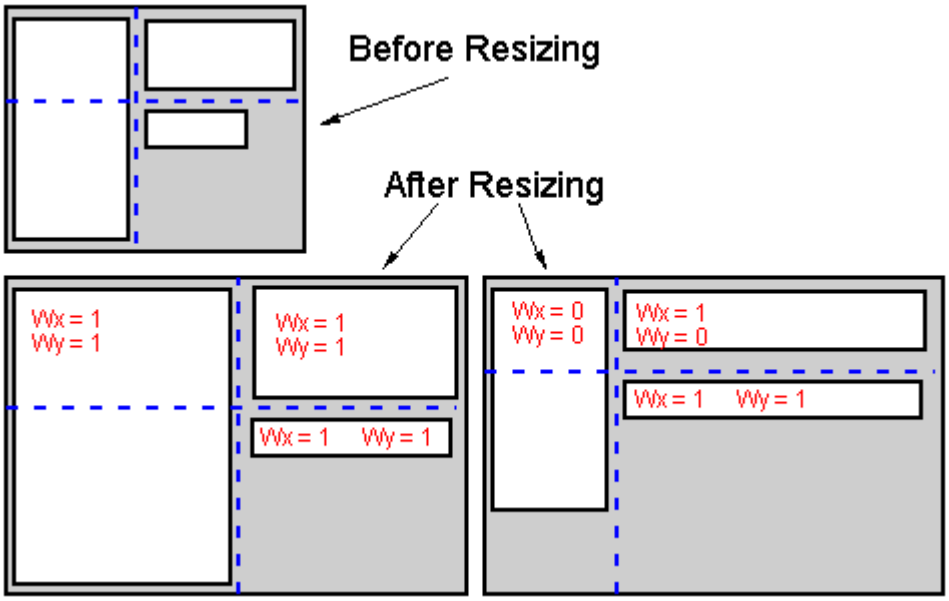
```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
...
```

Next, we can specify the margins around the outside of the components. That is, we can specify the margins between components in the same row and column. We do this using the **insets** parameter which is an Insets object which specifies the top, left, bottom and right margins (in pixels) that will offset this component within its cell. Here is how to set it for our list→




```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
constraints.insets = new Insets(10, 10, 10, 10);
...
```

Lastly, we can allow components to resize at different rates with respect to one another. To do this, we use the **weightx** and **weighty** parameters which specify the rate at which the component grows with respect to other components in the same row and column.

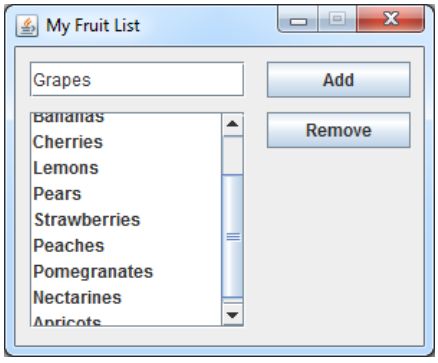


The weight can be sometimes difficult to set as it depends on the fill values of the other components in the same row and column. To begin, it is often best to start all components off with a **weightx** and **weighty** of 1, then adjust them as necessary. It is recommended not to have any row or column where all the weights are set to 0, as this is unpredictable. For our list, perhaps we could set the weights to 10:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
constraints.insets = new Insets(10, 10, 10, 10);
constraints.weightx = 10;
constraints.weighty = 10;
...
```

The **GridBagLayout** manager can be very complicated. It is best to follow the steps shown above by setting the value for each component individually.

Recall our FruitList example shown here →
How can we us a **GridBagLayout** so that the window is resizable ?

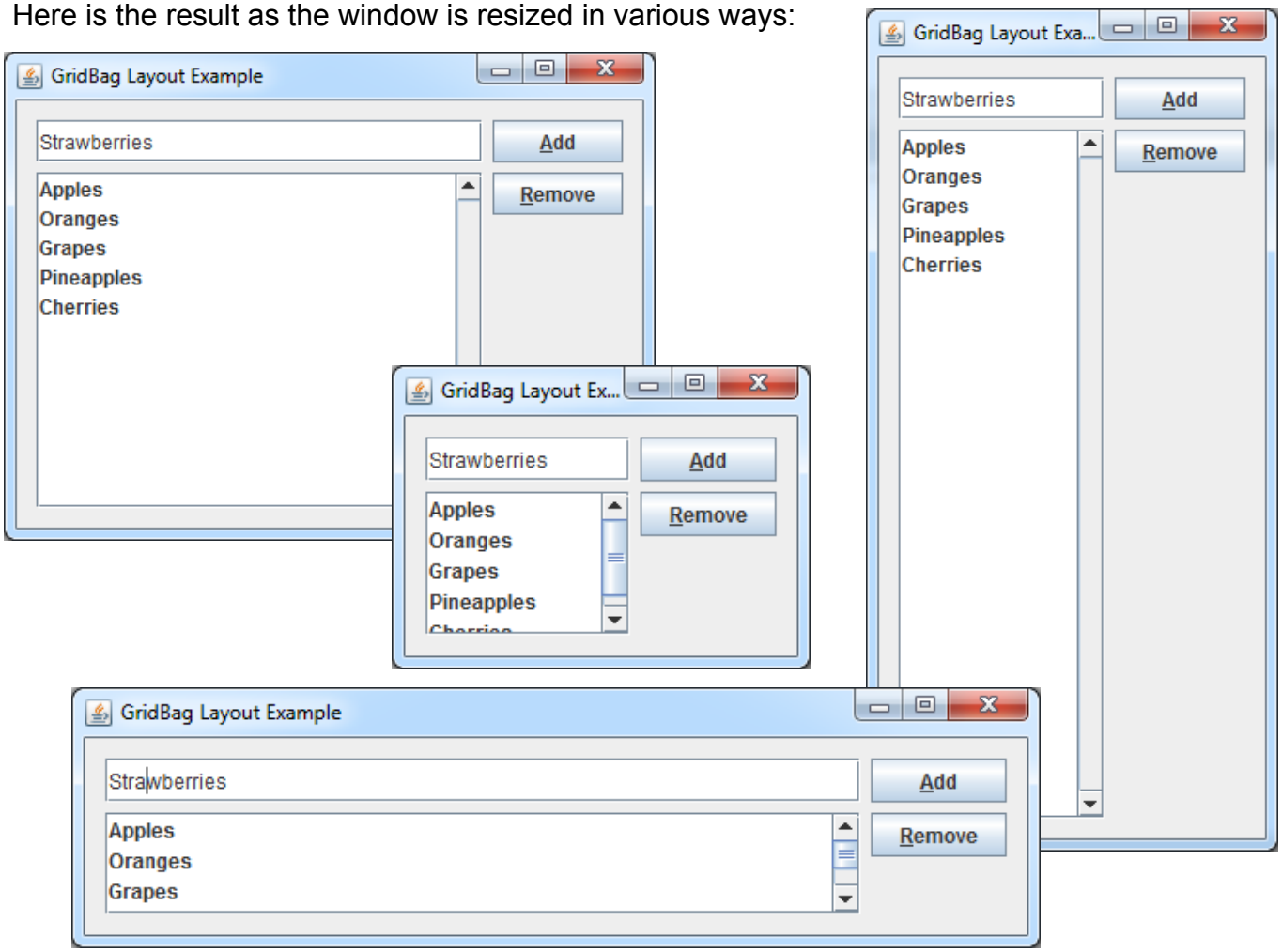


Here is the code:

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

public class GridBagLayoutExample extends JFrame {
     public GridBagLayoutExample(String name) {
          super(name);

          GridBagLayout layout = new GridBagLayout();
          GridBagConstraints constraints = new GridBagConstraints();
          getContentPane().setLayout(layout);


          JTextField newItemField = new JTextField();
          constraints.gridx = 0;
          constraints.gridy = 0;
          constraints.gridwidth = 1;
          constraints.gridheight = 1;
          constraints.fill = GridBagConstraints.BOTH;
          constraints.insets = new Insets(12, 12, 3, 3);
          constraints.weightx = 10;
          constraints.weighty = 0;
          layout.setConstraints(newItemField, constraints);
          getContentPane().add(newItemField);


          JButton addButton = new JButton("Add");
          addButton.setMnemonic('A');
          constraints.gridx = 1;
          constraints.gridy = 0;
          constraints.gridwidth = 1;
          constraints.gridheight = 1;
          constraints.fill = GridBagConstraints.HORIZONTAL;
          constraints.insets = new Insets(12, 3, 3, 12);
          constraints.anchor = GridBagConstraints.NORTHWEST;
          constraints.weightx = 0;
          constraints.weighty = 0;
          layout.setConstraints(addButton, constraints);
          getContentPane().add(addButton);


          String[] stuff = {"Apples","Oranges","Grapes","Pineapples","Cherries"};
          JList itemsList = new JList(stuff);
          JScrollPane scrollPane = new JScrollPane(itemsList,
              ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
              ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
          constraints.gridx = 0;
          constraints.gridy = 1;
          constraints.gridwidth = 1;
          constraints.gridheight = 1;
          constraints.fill = GridBagConstraints.BOTH;
          constraints.insets = new Insets(3, 12, 12, 3);
          constraints.anchor = GridBagConstraints.CENTER;
          constraints.weightx = 10;
          constraints.weighty = 1;
          layout.setConstraints(scrollPane, constraints);
          getContentPane().add(scrollPane);
```

```
        JButton removeButton = new JButton("Remove");
        removeButton.setMnemonic('R');
        constraints.gridx = 1;
        constraints.gridy = 1;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.insets = new Insets(3, 3, 0, 12);
        constraints.anchor = GridBagConstraints.NORTH;
        constraints.weightx = 0;
        constraints.weighty = 0;
        layout.setConstraints(removeButton, constraints);
        getContentPane().add(removeButton);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,300);
    }

    public static void main(String[] args) {
        new GridBagLayoutExample("GridBag Layout Example").setVisible(true);
    }
}
```

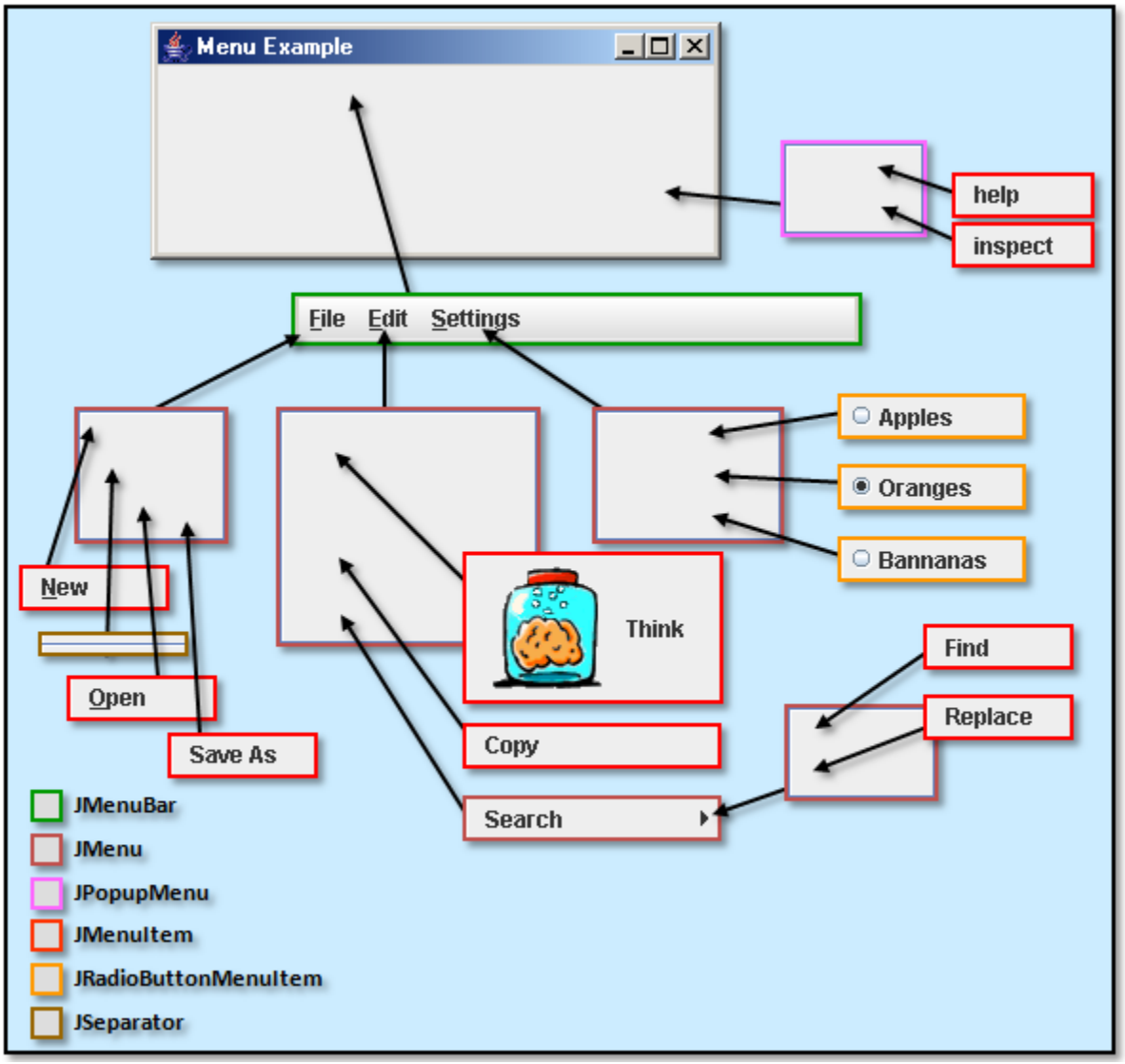
Here is the result as the window is resized in various ways:



7.2 Adding Menus

A ***menu*** is a list of commands presented to the user at his/her request. Menus can be attached to a menu bar at the top of an application or they may be pop-up menus that appear anywhere on the screen.

In JAVA, menus are as easy to use as buttons. There are several component classes that may be used including **JMenuBar**, **JMenu**, **JPopupMenu**, **JMenuItem**, **JSeparator** and **JRadioButtonMenuItem**. The diagram below shows how these components are connected together:



Notice that the **JMenuBar** is attached to the main **JFrame** as well as the **JPopupMenu**. The **JMenus** are then added to the **JMenuBar**, or to another menu to form a ***cascaded menu*** (e.g., the **Search** menu here). The **JMenuItems** are simply added to the **JMenus**.

Example:

Consider writing a program to produce the menu hierarchy in the above diagram. We will make a simple **JFrame** with nothing inside it except for the menu bar attached to the top.

A **JMenuBar** is added to a **JFrame** by doing the following in the **JFrame** constructor:

```
JMenuBar myMenuBar = new JMenuBar();
this.setJMenuBar(myMenuBar);
```

Once a menu bar has been created, then **JMenus** can be added to it in a simple manner:

```
JMenu  fileMenu = new JMenu("File");
myMenuBar.add(fileMenu);
```

Optionally, we can set the keyboard accelerators (i.e., quick keys) for the menu as well:

```
fileMenu.setMnemonic('F');
```

Once a menu has been created we can add **JMenuItems** and/or **JSeparators** to it. The menus will appear in the order that we add them:

```
JMenuItem  newItem = new JMenuItem("New");
JSeparator sepItem = new JSeparator();
fileMenu.add(newItem);
fileMenu.add(sepItem);
```

We can also set the keyboard accelerators for the **JMenuItems** if desired:

```
// This could have been done in the
// constructor:  new JMenuItem("New", 'N');
newItem.setMnemonic('N');
```

To get it working, we then add an **ActionListener** to <u>each</u> **JMenuItem**:

```
// they may all go to the same event handler or to separate ones
newItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
       // Handle the selection of this item from the menu
    }
});
```

We can also add **JRadioButtonMenuItems** to our **JMenus**:

```
JRadioButtonMenuItem rbItem1 = new JRadioButtonMenuItem("Apples");
JRadioButtonMenuItem rbItem2 = new JRadioButtonMenuItem("Oranges");
JRadioButtonMenuItem rbItem3 = new JRadioButtonMenuItem("Bananas");
settingsMenu.add(rbItem1);
settingsMenu.add(rbItem2);
settingsMenu.add(rbItem3);
```

```
rbItem1.addActionListener(this);
rbItem2.addActionListener(this);
rbItem3.addActionListener(this);

// Add them to a button group so that only one is on at a time
ButtonGroup fruits = new ButtonGroup();
fruits.add(rbItem1);
fruits.add(rbItem2);
fruits.add(rbItem3);
```

We can add cascading menus simply by adding a **JMenu** to another **JMenu**:

```
JMenu  searchMenu = new JMenu("Search");
JMenuItem findItem = new JMenuItem("Find");
JMenuItem replaceItem = new JMenuItem("Replace");
searchMenu.add(findItem);
searchMenu.add(replaceItem);
```

We then add the cascaded menu to some other **JMenu**:

```
fileMenu.add(searchMenu);
```

Finally, we add a **JPopupMenu** to the **JFrame**:

```
JPopupMenu popupMenu = new JPopupMenu();
JMenuItem helpItem = new JMenuItem("help");
JMenuItem inspectItem = new JMenuItem("inspect");
popupMenu.add(helpItem);
popupMenu.add(inspectItem);
```

To bring up a popup menu, we have to do a bit more work. On a PC, the right mouse button is usually used to bring up a popup menu. This is different on a Mac. JAVA has a method that can determine whether or not the "popup trigger" action (e.g., right mouse click) has just occurred. We can make use of this in our **mouseReleased()** event handler. When we determine that we really do want to bring up the menu, the **show()** method is used … which lets us specify the component (e.g., panel) on which to pop up the menu along with the x, y position within that component that we want the menu to appear at:

```
myFrame.addMouseListener(new MouseAdapter() {
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
            popupMenu.show(e.getComponent(), e.getX(), e.getY());
        }
    });
```

Keep in mind that there are other settings for our **JMenus** and **MenuItems**. To set the **Color** we can do this:

```
anItem.setBackground(Color.red);
anItem.setForeground(Color.yellow);
```

Or to Enable/Disable various items we can do this:

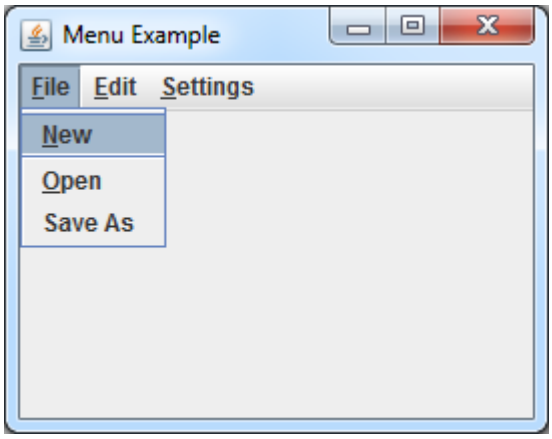
```
anItem.setEnabled(true);
anItem.setEnabled(false);
```

Here is the completed code:

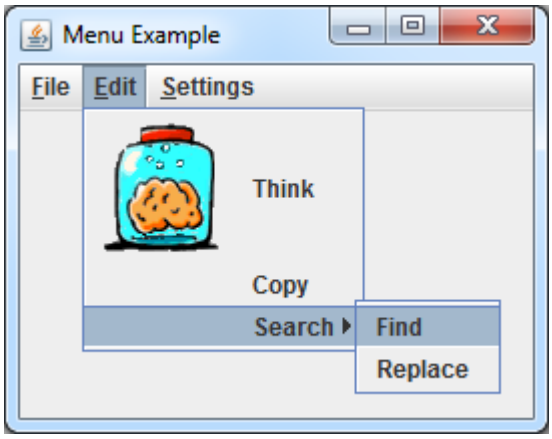
```
import java.awt.event.*;
import javax.swing.*;

public class MenuExample extends JFrame implements ActionListener {

   // Store menu items and popup menu for access from event handlers
   JMenuItem   thinkItem, copyItem, newItem, openItem, saveAsItem,
               findItem, replaceItem, appleItem, orangeItem,
               bananaItem, helpItem, inspectItem;

   JPopupMenu popupMenu;

   public MenuExample(String title) {
       super(title);

       // Create the menu bar
       JMenuBar menuBar = new JMenuBar();
       this.setJMenuBar(menuBar);   // call on JFrame, not on getContentPane()

       // Create and Add the File menu to the Menu Bar
       JMenu  fileMenu = new JMenu("File");
       fileMenu.setMnemonic('F');
       fileMenu.add(newItem = new JMenuItem("New", 'N'));
       fileMenu.add(new JSeparator());
       fileMenu.add(openItem = new JMenuItem("Open", 'O'));
       fileMenu.add(saveAsItem = new JMenuItem("Save As"));
       menuBar.add(fileMenu); // Don't forget to do this
       newItem.addActionListener(this);
       openItem.addActionListener(this);
       saveAsItem.addActionListener(this);

       // Create and Add the Edit menu to the Menu Bar
       JMenu  editMenu = new JMenu("Edit");
       editMenu.setMnemonic('E');
       editMenu.add(thinkItem = new JMenuItem("Think", new ImageIcon("brain.gif")));
       editMenu.add(copyItem = new JMenuItem("Copy"));
       menuBar.add(editMenu);
       thinkItem.addActionListener(this);
       copyItem.addActionListener(this);

       // Create and Add the Settings menu to the Menu Bar
       JMenu  settingsMenu = new JMenu("Settings");
       settingsMenu.setMnemonic('S');
       settingsMenu.add(appleItem = new JRadioButtonMenuItem("Apples"));
       settingsMenu.add(orangeItem = new JRadioButtonMenuItem("Oranges"));
       settingsMenu.add(bananaItem = new JRadioButtonMenuItem("Bananas"));
       menuBar.add(settingsMenu);

       // Ensure that only one radio button is on at a time
       ButtonGroup  fruits = new ButtonGroup();
       fruits.add(appleItem);
       fruits.add(orangeItem);
       fruits.add(bananaItem);
```

```
        // Create the cascading Search menu on the Settings menu
        JMenu  searchMenu = new JMenu("Search");
        searchMenu.add(findItem = new JMenuItem("Find"));
        searchMenu.add(replaceItem = new JMenuItem("Replace"));
        editMenu.add(searchMenu);
        findItem.addActionListener(this);
        replaceItem.addActionListener(this);

        // Create and Add items to the popup menu.   Notice
        // that we do not add the popup menu to anything.
        popupMenu = new JPopupMenu();
        popupMenu.add(helpItem = new JMenuItem("help"));
        popupMenu.add(inspectItem = new JMenuItem("inspect"));
        helpItem.addActionListener(this);
        inspectItem.addActionListener(this);

        // Register the event handler for the popup menu
        addMouseListener(new MouseAdapter() {
             public void mouseReleased(MouseEvent e){
                if (e.isPopupTrigger())
                   popupMenu.show(e.getComponent(), e.getX(), e.getY());
             }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
    }

    // Handle all menu selections accordingly
    public void actionPerformed(ActionEvent e){
        if (e.getSource() == newItem)
             System.out.println("reacting to NEW selection from menu");
        else if (e.getSource() == openItem)
             System.out.println("reacting to OPEN selection from menu");
        else if (e.getSource() == saveAsItem)
             System.out.println("reacting to SAVE AS selection from menu");
        else if (e.getSource() == copyItem)
             System.out.println("reacting to COPY selection from menu");
        else if (e.getSource() == thinkItem)
             System.out.println("reacting to THINK selection from menu");
        else if (e.getSource() == findItem)
             System.out.println("reacting to FIND selection from menu");
        else if (e.getSource() == replaceItem)
             System.out.println("reacting to REPLACE selection from menu");
        else if (e.getSource() == helpItem)
             System.out.println("reacting to HELP selection from popup menu");
        else if (e.getSource() == inspectItem)
             System.out.println("reacting to INSPECT selection from popup menu");
    }

    public static void main(String[] args) {
        new MenuExample("Menu Example").setVisible(true);
    }
}
```

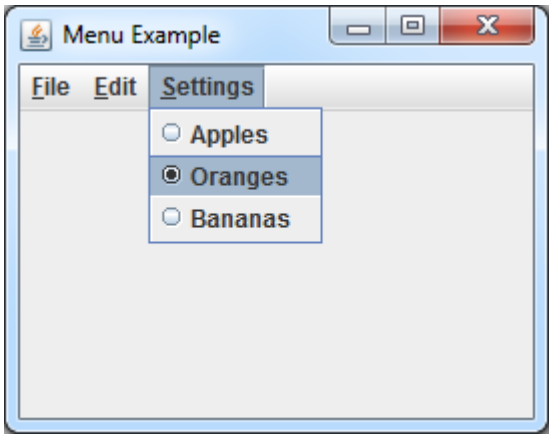
Here are the resulting screen snapshots showing the various menus:



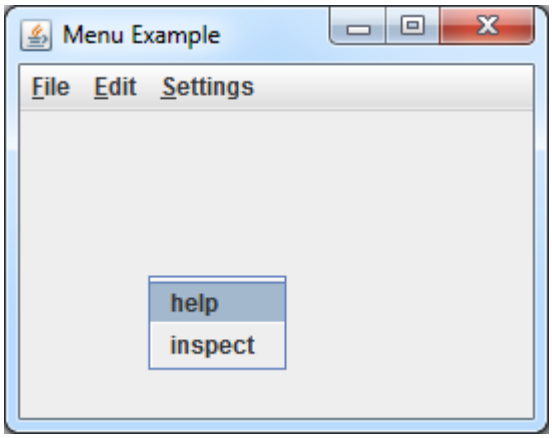
A standard menu



A cascaded menu



A menu with radio buttons



A popup menu

7.3 Standard Dialog Boxes

If a main application window has too many components on it, it will look cluttered and it will not be simple and easy to use. It is a good idea not to display components on your window if they are not needed at that time. For example, a main application may not want to display name, address and phone number fields until the user has selected some action that requires that information to be entered. Usually, this information is placed in a different window that "pops up" when needed.

> *A **Dialog Box** is a secondary window (i.e., not the main application window) that is used to interact with the user ... usually to display or obtain additional information.*

So ... a dialog box is another window that can be brought up at any time in your application to interact with the user.

There are various types of commonly used dialog boxes in JAVA:

1. Message Dialog - displays a message indicating information, errors, warnings etc...
2. Confirmation Dialog - asks a question such as yes/no
3. Input Dialog - asks for some kind of input
4. Option Dialog - asks the user to select some option

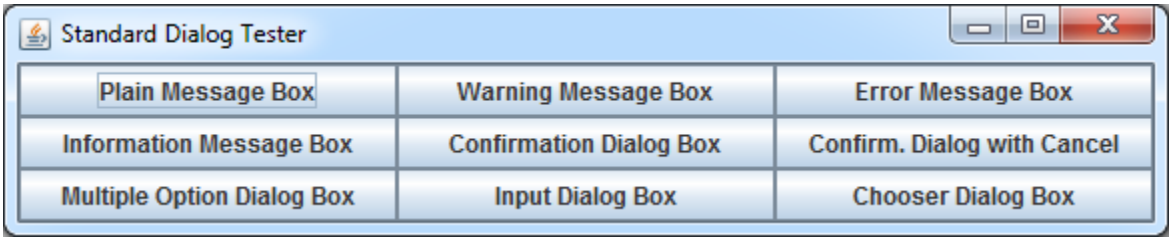
JAVA has a class called **JOptionPane** that can bring up one of these standard dialog boxes. There are many parameters and JAVA allows you to be very flexible in the way that you use them. For instance, there are standard icons that are displayed on these dialog boxes, but you can also make your own.

When using one of these standard dialog boxes, you may specify:

- the frame (owner)
- the title on the dialog box
- the message or question to be asked
- the icon displayed
- the buttons to be shown on the dialog box (i.e. OK, CANCEL, YES, NO)
- a set of options to be asked

Example:

Instead of describing ALL the options and all combinations here, I have decided to just give you a few templates that you can use. Here is some code that tests various standard dialog boxes. It brings up an interface with 9 buttons that allow you to "try out" the boxes. The interface looks as follows:



Here is the code for our test application. Notice the output that appears in the console when running the code. You should be able to figure out how to get information easily from your dialog boxes from this example.

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

public class StandardDialogTestProgram extends JFrame {
    public StandardDialogTestProgram(String title) {
        super(title);
```

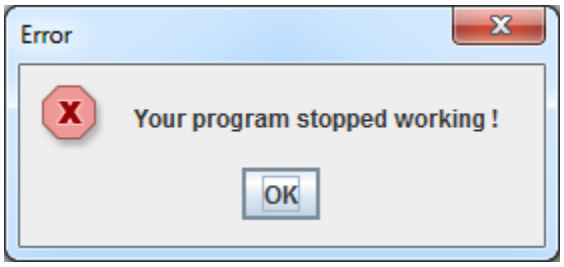
```
        // Make a grid layout for the 9 buttons
        getContentPane().setLayout(new GridLayout(3, 3));
        JButton aButton;

        // Create the button and event handler for the Plain Message Box
        getContentPane().add(aButton = new JButton("Plain Message Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "This is a plain message !!!",
                                              "Read This", JOptionPane.PLAIN_MESSAGE);
            }});

        // Create the button and event handler for the Warning Message Box
        getContentPane().add(aButton = new JButton("Warning Message Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "Don't eat yellow snow.",
                                            "Warning", JOptionPane.WARNING_MESSAGE);
            }});

        // Create the button and event handler for the Error Message Box
        getContentPane().add(aButton = new JButton("Error Message Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "Your program stopped working !",
                                            "Error", JOptionPane.ERROR_MESSAGE);
            }});

        // Create the button and event handler for the Information Message Box
        getContentPane().add(aButton = new JButton("Information Message Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null,
                    "You better pass the final exam or else ...",
                    "Information", JOptionPane.INFORMATION_MESSAGE);
            }});

        // Create the button and event handler for the Confirmation Dialog Box
        getContentPane().add(aButton = new JButton("Confirmation Dialog Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int result = JOptionPane.showConfirmDialog(null,
                                        "Do you want me to erase your hard drive ?",
                                        "Answer this Question",
                                        JOptionPane.YES_NO_OPTION);
                if (result == 0)
                    System.out.println("OK, I'm erasing it now ...");
                 else
                    System.out.println("Fine then, you clean it up!");
            }});

        // Create the button & event handler for Confirmation Dialog Box With Cancel
        getContentPane().add(aButton = new JButton("Confirm. Dialog with Cancel"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int result = JOptionPane.showConfirmDialog(null,
                           "Do you want to overwrite the file ?",
```

```
                        "Answer this Question", JOptionPane.YES_NO_CANCEL_OPTION);
                switch(result) {
                    case 0: System.out.println("OK, here goes..."); break;
                    case 1: System.out.println("Then choose a new name..."); break;
                    case 2: System.out.println("I will ask you later..."); break;
                }
            }});

        // Create the button & event handler for Multiple Option Dialog Box
        getContentPane().add(aButton = new JButton("Multiple Option Dialog Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Object[] options = {"Outstanding","Excellent","Good","Fair","Poor"};
                int result = JOptionPane.showOptionDialog(null,
                                "How would you rate your vehicle's performance ?",
                                "Pick an Option", JOptionPane.DEFAULT_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, options, options[0]);
                if (result > 0) {
                    System.out.print("You have rated your vehicle's performance as "
                                            + options[result]);
                    if (result < 3)
                        System.out.println("We are glad you are pleased.");
                    else
                        System.out.println("Please explain why.");
                }}});

        // Create the button & event handler for Chooser Dialog Box
        getContentPane().add(aButton = new JButton("Chooser Dialog Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Object[] options = {"Apple", "Orange", "Strawberry", "Banana"};
                Object selectedValue = JOptionPane.showInputDialog(null,
                                                "Choose your favorite fruit",
                                                "Fruit Information",
                                                JOptionPane.INFORMATION_MESSAGE,
                                                null, options, options[1]);
                System.out.println(selectedValue + "s sure do taste yummy.");
            }});
        // Create the button & event handler for Input Dialog Box
        getContentPane().add(aButton = new JButton("Input Dialog Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String in = JOptionPane.showInputDialog("Please input your name");
                System.out.println("Your name is " + in);
            }});

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack(); //chooses reasonable window size based on component prefered sizes
    }

    public static void main(String[] args) {
        new StandardDialogTestProgram("Standard Dialog Tester").setVisible(true);
    }
}
```

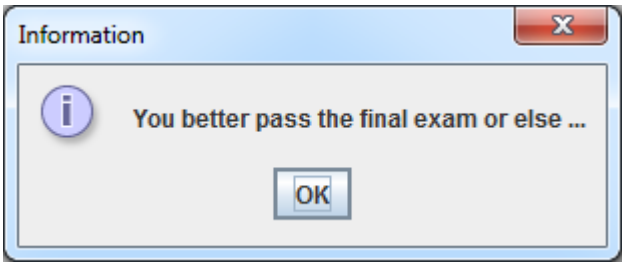
Here are the dialog boxes that will appear.

Plain Message Box

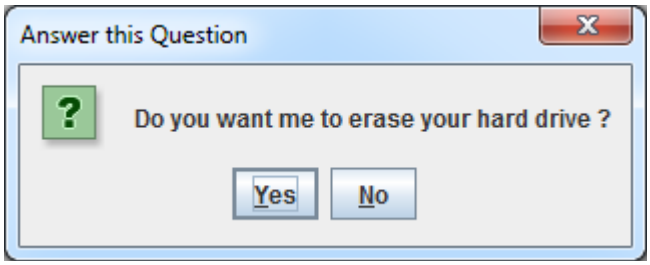
Warning Message Box



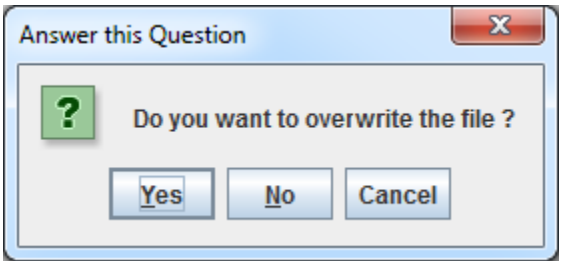
Error Message Box



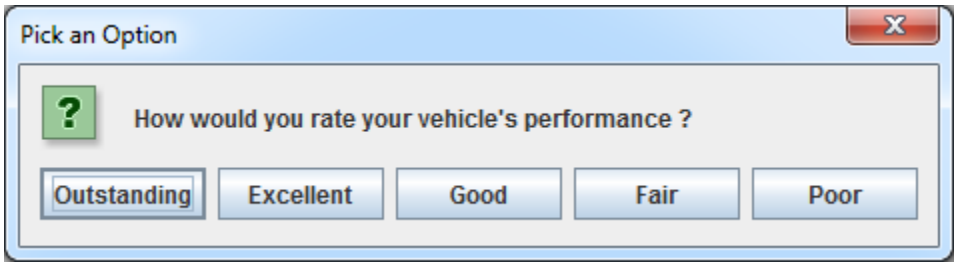
Information Message Box



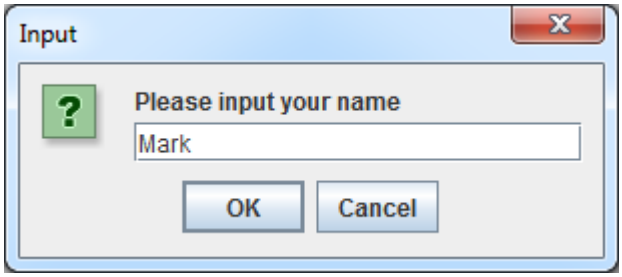
Confirmation Dialog Box



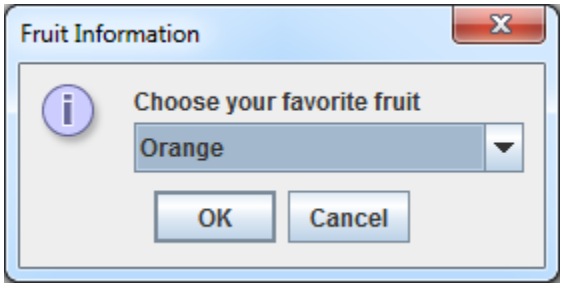
Confirmation Dialog Box with Cancel



Multiple Option Dialog Box

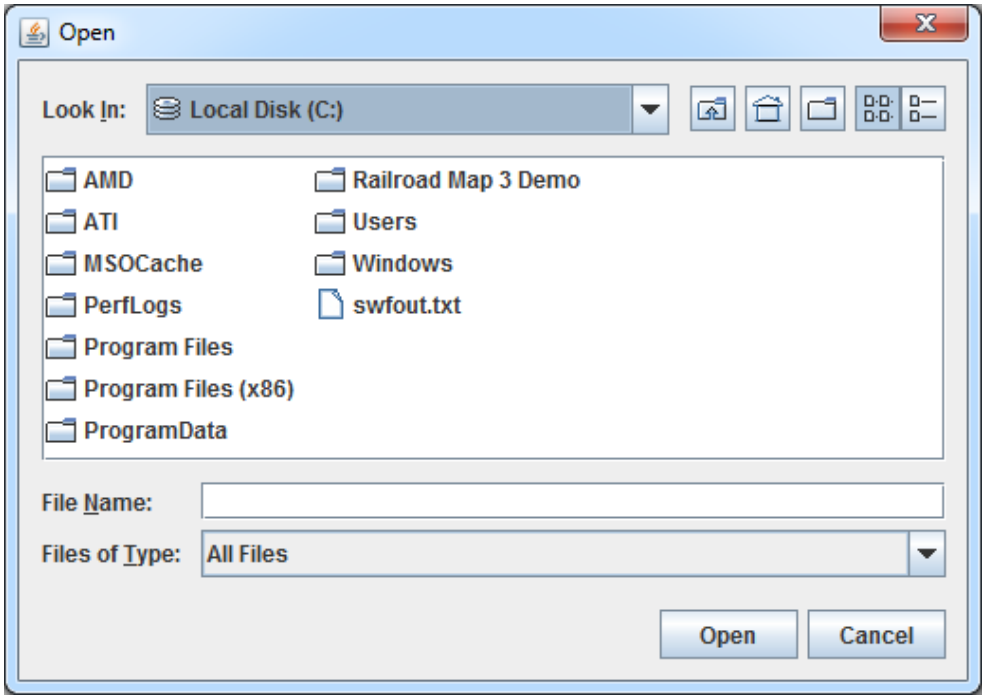


Input Dialog Box



Option Dialog Box

There is another useful ***standard*** dialog box in JAVA that is used for selecting files. It is called a **JFileChooser**. Here is what it looks like:



Here is some code that opens up a **JFileChooser** box and displays the filename (no path) that the user selects.

```
import javax.swing.*;

public class FileChooserTestProgram {
      public static void main(String[] args) {
            JFileChooser chooser = new JFileChooser();
            int returnVal = chooser.showOpenDialog(null);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
                  System.out.println("You chose to open this file: " +
                                          chooser.getSelectedFile().getName());
            }
      }
}
```

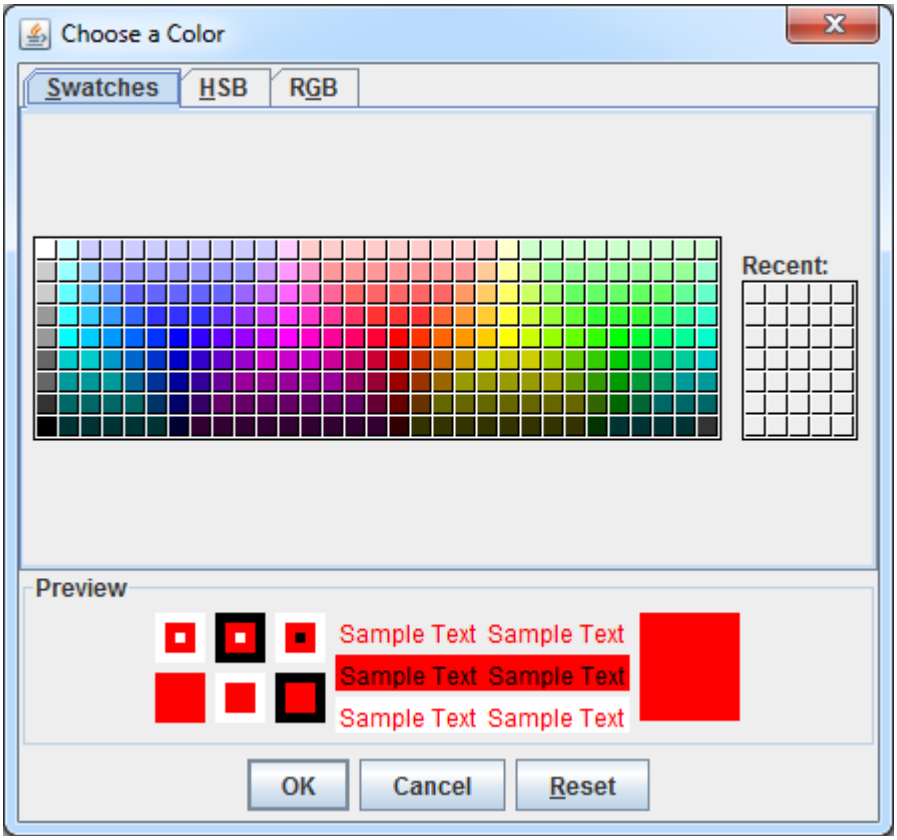
In the above code, the **null** parameter in the **showOpenDialog()** represents the parent window that will bring up this dialog box. Since there are no other windows in this example, it has been set to **null**. There are more options available that allow you to set the filters and starting directories. Take a look at the Swing API. Here, for example, is how to set some filters:

```
        FileNameExtensionFilter filter = new FileNameExtensionFilter(
                                              "JPG & GIF Images", "jpg", "gif");
        chooser.setFileFilter(filter);
```

Try adding it to the above code ... you will need to add this to the top of your program:

```
      import javax.swing.filechooser.FileNameExtensionFilter;
```

There is also a **JColorChooser** class in JAVA that can be used to bring up a dialog box that allows you to select a color. Here is what it looks like:



You create and add a **JColorChooser** just as you would any other component:

```
import javax.swing.JColorChooser;
import java.awt.Color;

public class ColorChooserTestProgram {
	public static void main(String[] args) {
		Color newColor = JColorChooser.showDialog(
					null,			// The parent window
					"Choose a Color", // Title on Dialog Box
					Color.RED);		// Initial color selected

		System.out.println("You selected this color: " + newColor);
	}
}
```

Notice that the dialog box returns the color selected when the window is closed.

7.4 Making Your Own Dialog Boxes

There are two modes that a dialog box may be brought up in:

- ***modal:*** the application window will not respond until this dialog box is closed. This mode forces the user to "deal with" the dialog box information before continuing.

- ***non-modal:*** the dialog box can remain open while the user works in other windows.

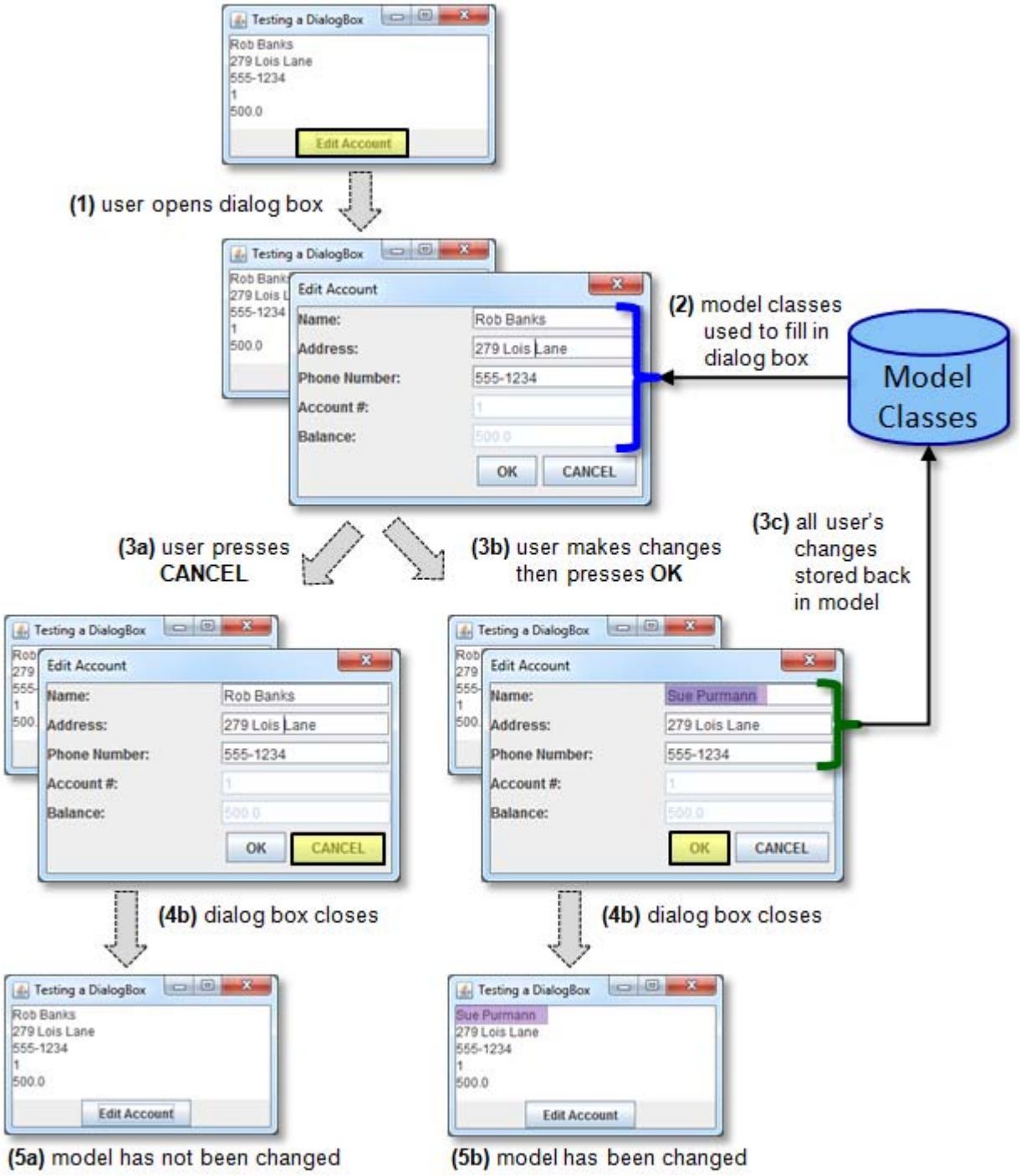
Dialog boxes have an ***owner*** which is the window that caused it to appear. This allows the dialog box to be closed automatically when the user quits the application from the main window (i.e., all windows belonging to the same application are closed when the application shuts down). Also, when the owner window is minimized, the dialog boxes are also minimized.

There are two important terms pertaining to dialog boxes:

- ***Dialog client*:** the application that caused the dialog box to appear.
- ***Dialog model*:** the object(s) that the dialog box should affect.

Normally, an application communicates with its dialog box through a ***model*** of some kind. That is, the owner opens up a dialog box, passing model-specific information to it. The user may then change this information from the dialog box, which in turn modifies the ***model***. When the dialog box is closed, then the main application continues with the modified model objects.

The next page shows a diagram of how everything should work. Notice that the model is used as the "middle-man" between the two windows. That is, when the dialog box is first opened, the model contents are used to populate the components (i.e., fill in the text fields, button selections etc...). The user then makes appropriate changes to the components. When the dialog box is closed with the OK button, the model is updated with these new changes. When the dialog box is closed with the CANCEL button, the model remains unchanged. When either button is clicked, the dialog box closes. The closing of the dialog box using the standard "close" (i.e., X at the top corner) should be treated as a cancel operation.



The dialog box itself is easy to make. It is simply another window. To create your own, simply make it a subclass of **JDialog**:

```
public class MyDialogBox extends JDialog {
      ...
}
```

Then, you can add components and event handlers to this dialog box as if it were a **JPanel**. Typically, you will ensure that there is some combination of ok/apply and cancel/close buttons which are usually located at the bottom right or bottom center of a dialog box.

There are many constructors in the **JDialog** class. We will use the following format for our constructors:

```
public MyDialog(Frame  owner, String  title, boolean  modal, ...) {
      super(owner, title, modal);
      ...

      // Ensure that the dialog box appears close to the main window
      setLocationRelativeTo(owner);
}
```

The **owner** parameter is the client application. In our examples, the **owner** will also need to be a class that implements the **DialogClientInterface** (described soon below). The **title** parameter is what will appear on the dialog box title bar. The **modal** parameter is a **boolean** indicating whether or not the dialog box should be modal. Notice the call to the superclass constructor (this is a standard **JDialog** constructor being called).

We may also want to supply additional model-related parameters to pass information into the dialog box. Often the model itself is passed in as a parameter so that we can (1) fill in the dialog box information based on the current model data, and (2) then we can modify the model as necessary after the user makes changes to the data and presses OK.

In some cases, we may not want the user of the dialog box to decide whether or not it should be modal, nor may we want them to specify the title. We can simply hard-code these into the dialog box if we wish:

```
public MyDialog(Frame owner, ...) {
      super(owner, "Mt Cool Dialog Box", true);
      ...

      // Ensure that the dialog box appears close to the main window
      setLocationRelativeTo(owner);
}
```

In addition to this, we will use the **dispose()** method to dispose of (i.e., close and delete) the dialog box from within our code once we press OK, CANCEL or close the window from the top X.

So dialog boxes are easy to create ... but how do we coordinate the interaction with the main application window and its dialog box ?

The dialog box is defined in a separate class than its owner application. As a result, the ***client*** (i.e., the application that brought up the dialog box) has no idea what is going on within the dialog box class (nor should it need to know). The *client*, however, usually needs to know whether or not the interaction with the dialog box was ***accepted*** or whether or not it was ***cancelled***. That is, it may need to know whether or not changes were made to the model.

Since the client application does not have access to the internal dialog box classes, the dialog box will need to inform the *client* application as to whether or not the user pressed OK or CANCEL. That means, the dialog box must call one or more methods in the *client* class.

In order to do this cleanly, the dialog box should not need to know who exactly the *client* is. That is, a dialog box in general should allow any application to bring it up and then interact with it. Therefore, we need a general way of communicating with an arbitrary *client* class.

JAVA interfaces are perfect for this situation. We can define the following general interface:

```
public interface DialogClientInterface {
     public void dialogFinished();
     public void dialogCancelled();
}
```

Then, we can have the *client* application implement this interface:

```
public class MyApplication implements DialogClientInterface {
    ...
    public void dialogFinished() {
        ...
    }
    public void dialogCancelled() {
        ...
    }
    ...
}
```

Now, since the *client* application implements the interface, the dialog box has a guarantee that it will have **dialogFinished()** and **dialogCancelled()** methods available to call.

Therefore, in the dialog class, when the OK button is pressed, we can call the *client*'s **dialogFinished()** method before closing the dialog box. That will tell the *client* that the model has been changed. Similarly, when CANCEL is pressed (or when the window is closed from the top right X), we can tell the client that the dialog was cancelled. Here is the structure of the dialog box code:

```
public class SomeDialog extends JDialog {
    // A constructor that takes the model and client as parameters
    public SomeDialog(Frame owner, ...){
        ...
    }

    private void okButtonPressed() {
        ...
        ((DialogClientInterface)getOwner()).dialogFinished();
    }
    private void cancelButtonPressed() {
        ...
        ((DialogClientInterface)getOwner()).dialogCancelled();
    }
}
```

Notice how the **owner** (which is a **JFrame**) is type-casted to a **DialogClientInterface** so that the **dialogFinished()** and **dialogCancelled()** methods can be called.

The last piece of information that you will need is that of making the dialog box appear. To have a dialog box appear, you simply create an instance of it and make it visible. Here is some code that would appear in the main application client:

```
public void bringUpDialogBox() {
    MyDialog  dialog =  new MyDialog (this, "My Dialog", true, model);

    System.out.println("This appears before Dialog box is opened.");

    dialog.setVisible(true);    // Open the dialog box

    System.out.println("This appears after Dialog box is closed...");
    System.out.println("unless the dialog box was non-modal.");
}
```

Example:

Consider having many "buddies" (i.e., friends) that you send e-mails to regularly. You would like to make a nice little electronic address book that you can store the buddy's names along with his/her e-mail addresses. Perhaps you even want to categorize the buddies as being "hot" (i.e., you talk to them often), or not-so-hot.

What exactly is an e-mail buddy ? Well we can easily develop a simple model of an **EmailBuddy** as follows:

```
public class EmailBuddy {
    private String   name;
    private String   address;
    private boolean  onHotList;

    // Here are some constructors
    public EmailBuddy() {
        name = "";
        address = "";
        onHotList = false;
    }
    public EmailBuddy(String aName, String anAddress) {
        name = aName;
        address = anAddress;
        onHotList = false;
    }

    // Here are the get methods
    public String getName() { return name; }
    public String getAddress() { return address; }
    public boolean onHotList() { return onHotList; }
```

```
    // Here are the set methods
    public void setName(String newName) { name = newName; }
    public void setAddress(String newAddress) { address = newAddress; }
    public void onHotList(boolean onList) { onHotList = onList; }

    // The appearance of the buddy
    public String toString() {
        return(name);
    }
}
```

As you may have noticed, there is nothing difficult here ... just your standard "run-of-the-mill" model class. However, this class alone does not represent the whole model for our GUI since we will have many of these **EmailBuddy** objects. So we will need a class to represent the list. We can do this in the same way that we created our grocery item list with an array of **EmailBuddy** objects

```
public class EmailBuddyList {

      public final int  MAXIMUM_SIZE = 100;

      private EmailBuddy[]    buddies;
      private int             size;

      public EmailBuddyList() {
            buddies = new EmailBuddy[MAXIMUM_SIZE];
            size = 0;
      }

      // Return the number of buddies in the whole list
      public int getSize() { return size; }

      // Return all the buddies
      public EmailBuddy[] getEmailBuddies() { return buddies; }

      // Get a particular buddy from the list, given the index
      public EmailBuddy getBuddy(int i) { return buddies[i]; }

      // Add an email buddy to the list unless it has reached its capacity
      public void add(EmailBuddy buddy) {
            // Make sure that we do not go past the limit
            if (size < MAXIMUM_SIZE)
                  buddies[size++] = buddy;
      }

      // Remove the buddy with the given index from the list
      public void remove(int index) {
            // Make sure that the given index is valid
            if ((index >= 0) && (index < size)) {
                  // Move every item after the deleted one up in the list
                  for (int i=index; i<size-1; i++)
                        buddies[i] = buddies[i+1];
                  size--;     // Reduce the list size by 1
            }
      }
```

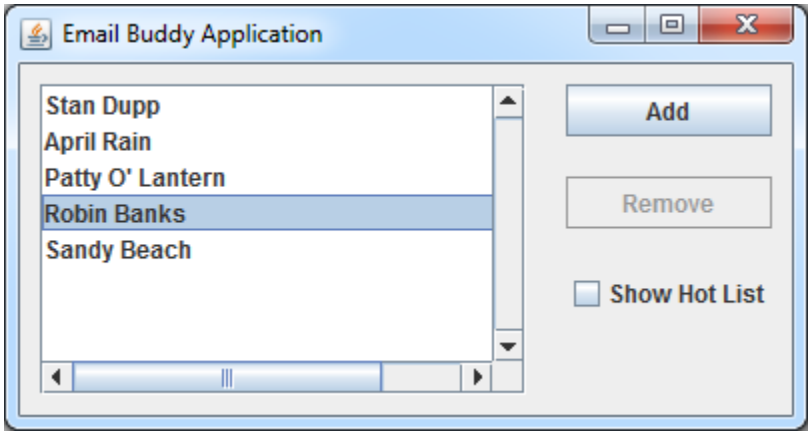
```
	// Return the number of buddies on the hot list
	public int getHotListSize() {
		int count = 0;
		for (int i=0; i<size; i++)
			if (buddies[i].onHotList())
				count++;
		return count;
	}

	// Get a particular "hot" buddy from the list, given the hot list index
	public EmailBuddy getHotListBuddy(int i) {
		int count = 0;
		for (int j=0; j<size; j++) {
			if (buddies[j].onHotList()) {
				if (count == i)
					return buddies[j];
				count++;
			}
		}
		return null;
	}
}
```

Notice that there is a **getSize()** method that is a simple "get" method and there is also a **getHotListSize()** method that returns the number of buddies on the hot list. Notice as well that there are methods to get a buddy at a given index in the array. The method **getHotListBuddy()** will find the **i**th buddy that is on the hot list. You will see soon why these methods will be useful.

The task now is to design a nice interface for the main application. To start, we must decide what the interface should do. Here is a possible interface:

- A **list** of all buddies is shown (names only)
- We should be able to
  - **Add** and **Remove** buddies from the list
  - **Edit** buddies when their name or email changes
  - Show only those buddies that are **"hot"** or perhaps show all of them

Assume that we have decided upon the following view for the interface:



Notice that the interface does not show the e-mail addresses in the list. It may look cluttered, but we could certainly have done this. Perhaps we could have made a second list box or something that would show the e-mail addresses. Here is a good exercise: make a **JTextField** just beneath the list that will show the e-mail address of the currently selected **EmailBuddy** in the list. This is not hard to do. Nevertheless, it is not necessary for the purposes of explaining this dialog box example.

How can we build the view for this interface ? We will start with a **JPanel**. We will use **GridBagLayout** to allow nice resizing.

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

public class EmailBuddyPanel extends JPanel {
    private EmailBuddyList    model;      // This is the list of buddies

    private JButton   addButton;
    private JButton   removeButton;
    private JList     buddyList;
    private JCheckBox hotListButton;

    // These are the get methods that are used to access the components
    public JButton getAddButton() { return addButton; }
    public JButton getRemoveButton() { return removeButton; }
    public JCheckBox getHotListButton() { return hotListButton; }
    public JList getBuddyList() { return buddyList; }

    // This is the default constructor
    public EmailBuddyPanel(EmailBuddyList  m){
        super();

        model = m; // Store the model so that he update() method can access it

        // Use a GridBagLayout (lotsa fun)
        GridBagLayout layout = new GridBagLayout();
        GridBagConstraints layoutConstraints = new GridBagConstraints();
        setLayout(layout);

        // Add the buddy list
        buddyList = new JList();
        JScrollPane scrollPane = new JScrollPane(buddyList,
                                 ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
                                 ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        layoutConstraints.gridx = 0;  layoutConstraints.gridy = 0;
        layoutConstraints.gridwidth = 1;  layoutConstraints.gridheight = 3;
        layoutConstraints.fill = GridBagConstraints.BOTH;
        layoutConstraints.insets = new Insets(10, 10, 10, 10);
        layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
        layoutConstraints.weightx = 1.0;  layoutConstraints.weighty = 1.0;
        layout.setConstraints(scrollPane, layoutConstraints);
        add(scrollPane);

        // Add the Add button
        addButton = new JButton("Add");
        layoutConstraints.gridx = 1;  layoutConstraints.gridy = 0;
        layoutConstraints.gridwidth = 1;  layoutConstraints.gridheight = 1;
```

```
        layoutConstraints.weightx = 0.0;  layoutConstraints.weighty = 0.0;
        layout.setConstraints(addButton, layoutConstraints);
        add(addButton);

        // Add the Remove button
        removeButton = new JButton("Remove");
        layoutConstraints.gridx = 1;  layoutConstraints.gridy = 1;
        layout.setConstraints(removeButton, layoutConstraints);
        add(removeButton);

        // Add the ShowHotList button
        hotListButton = new JCheckBox("Show Hot List");
        layoutConstraints.gridx = 1;  layoutConstraints.gridy = 2;
        layout.setConstraints(hotListButton, layoutConstraints);
        add(hotListButton);

        // Now update the components by filling them in
        update();
    }
}
```

Of course, we will need to write the **update()** method as well here. Recall that the update method should read from the model and then refresh the "look" of the components. The only components that need their appearance updated is the list and the remove button. The remove button is easily updated as we simply disable it when there is nothing selected in the list:

```
removeButton.setEnabled(buddyList.getSelectedIndex() >= 0);
```

The list is more complicated. First of all, we need to populate the list with the most recent data. Recall that we did something similar in the grocery list example. We need to create an appropriate sized array and then fill it up with email buddies and then set the list data:

```
EmailBuddy[]  exactList;

exactList = new EmailBuddy[model.getSize()];
for (int i=0; i<model.getSize(); i++)
      exactList[i] = model.getBuddy(i);
buddyList.setListData(exactList);
```

However, things are a little more difficult now. If we have the hot list button selected, then we do not want all the buddies ... instead we want only those on the hot list:

```
exactList = new EmailBuddy[model.getHotListSize()];
for (int i=0; i<model.getHotListSize(); i++)
      exactList[i] = model.getHotListBuddy(i);
buddyList.setListData(exactList);
```

We can use an **IF** statement to select the appropriate code:

```
EmailBuddy[]  exactList;
if (hotListButton.isSelected()) {
      exactList = new EmailBuddy[model.getHotListSize()];
```

```
            for (int i=0; i<model.getHotListSize(); i++)
                  exactList[i] = model.getHotListBuddy(i);
      }
      else {
            exactList = new EmailBuddy[model.getSize()];
            for (int i=0; i<model.getSize(); i++)
                  exactList[i] = model.getBuddy(i);
      }
      buddyList.setListData(exactList);
```

We will also need to ensure that we select the selected item each time we make an update. That is, if we were to select an item from the list and then update ... we want to make sure that the item remains selected. At this point, when we refresh the list contents, the selected item does not remain selected. So, we will need to remember which item was selected and then reselect it again after the list is re-populated. Here is the final **update()** method that must be added to the view code:

```
      // Update the components so that they reflect the contents of the model
      public void update() {
            //Remember what was selected
            int selectedItem = buddyList.getSelectedIndex();

            // Now re-populate the list by creating and returning a new
            // array with the exact size of the number of items in it.
            EmailBuddy[]  exactList;
            if (hotListButton.isSelected()) {
                  exactList = new EmailBuddy[model.getHotListSize()];
                  for (int i=0; i<model.getHotListSize(); i++)
                        exactList[i] = model.getHotListBuddy(i);
            }
            else {
                  exactList = new EmailBuddy[model.getSize()];
                  for (int i=0; i<model.getSize(); i++)
                        exactList[i] = model.getBuddy(i);
            }           buddyList.setListData(exactList);

            // Reselect the selected item
            buddyList.setSelectedIndex(selectedItem);

            // enable/disable the remove button accordingly
            removeButton.setEnabled(buddyList.getSelectedIndex() >= 0);
      }
```

At this point, the view is complete and we just have to create the controller. The controller will keep track of the view as well as the model. We will be handling events for the pressing of the Add button, Remover button, Hot List button as well as List Selection. Here is the basic framework for the controller:

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

public class EmailBuddyApp extends JFrame implements DialogClientInterface {

      private EmailBuddyList        model;      // The model
      private EmailBuddyPanel       view;       // The view
```

```
	// Here is the default constructor
	public EmailBuddyApp(String title){
		super(title);

		// Initially, no buddies
		model = new EmailBuddyList();

		// Make a new viewing panel and add it to the pane
		view = new EmailBuddyPanel(model);
		getContentPane().add(view);

		// Make a listener for the add button
		view.getAddButton().addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent event) {
				addBuddy();
			}});

		// Make a listener for the remove button
		view.getRemoveButton().addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent event){
				removeBuddy();
			}});

		// Make a listener for the hot list checkbox
		view.getHotListButton().addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent event){
				view.update();
			}});

		// Make a double-click listener
		view.getBuddyList().addMouseListener(new MouseAdapter() {
			public void mouseClicked(MouseEvent event){
				if (event.getClickCount() == 2)
					editBuddy();
				view.update();
			}});

		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setSize(600,300);
		setLocation(800,400);
	}

	...

	// This is called when the user clicks the add button
	private void addBuddy() { ... }

	// This is called when the user clicks the remove button
	private void removeBuddy() { ... }

	// This is called when the user double clicks a buddy
	private void editBuddy() { ... }


	// Code that starts the application
	public static void main(String[] args) {
		new EmailBuddyApp("Email Buddy Application").setVisible(true);
	}
}
```

Notice that the code is straight forward. The hot list button event handler only requires a refreshing of the list, so the view's **update()** method is called.

Now let us examine the helper methods in turn. The **addBuddy()** method is called when the user clicks the **Add** button. This method should bring up the dialog box for adding a new buddy. We have not created this dialog box, but we will do so soon. The adding of the new email buddy should only occur if the user presses the OK button. If the CANCEL button is pressed, or the dialog box is closed down, then no email buddy should be added.

To make the code simpler, it is a good idea to create the new email buddy when the **Add** button is pressed so that we can pass this buddy into the dialog box so that its contents can be set. We can go ahead and add it to the model. Of course, if the user cancels the adding of the new buddy, we will have to remove this newly added buddy. In the **addBuddy()** event handler, we will just need to create the buddy, add it to the model and then open up the dialog box (which we have not yet created, but will do so soon):

```
private void addBuddy() {
      EmailBuddy  aBuddy = new EmailBuddy();
      model.add(aBuddy);            // Add the buddy to the model

      // Now bring up the dialog box
      BuddyDetailsDialog dialog = new BuddyDetailsDialog(this,
                                    "New Buddy Details", true, aBuddy);
      dialog.setVisible(true);
}
```

This code will bring up the dialog box. Remember, that this main application implements the **DialogClientInterface**. Therefore, the **BuddyDetailsDialog** will be trying to call the **dialogFinished()** and **dialogCancelled()** methods, so we will need to write these:

```
// Called when dialog box is closed with OK button
public void dialogFinished() {
      view.update();
}
// Called when dialog box is closed with CANCEL button or manually closed
public void dialogCancelled() {
      // Remove the latest buddy that was added
      model.remove(model.getSize()-1);
}
```

The methods are simple. Since we added the buddy when in the **addBuddy()** method, when the user presses OK, there is nothing left to do except update the view. However, when the user presses CANCEL, we need to remove the last added buddy from the model ... which is done in the **dialogCancelled()** method here.

The **removeBuddy()** event handler is also easy to write. We just need to determine which buddy is selected from the list and then as long as there is someone selected ... we just remove the buddy by calling the model's **remove()** method and then update the view:

```
// This is called when the user clicks the remove button
private void removeBuddy() {
      int index = view.getBuddyList().getSelectedIndex();
      if (index >= 0) {
```

```
                model.remove(index);
                view.update();
        }
    }
```

The **editBuddy()** event handler is a little more involved. First, we need to determine which buddy was selected ... we can grab the index in the list. Then, as long as there was a buddy selected, we can bring up the dialog box for that buddy. Unfortunately, however, the index in the list will be different from the index into the model's list if the hot list button is enabled. So we will need to handle these cases separately.

Likely, we will want to use the same dialog box as we did with the add button, but with the selected buddy's information as opposed to adding a new buddy. Here is the basic idea:

```
    private void editBuddy() {
        EmailBuddy selectedBuddy;

        int   selectedIndex = view.getBuddyList().getSelectedIndex();
        if (selectedIndex >= 0) {
            if (view.getHotListButton().isSelected())
                selectedBuddy = model.getHotListBuddy(selectedIndex);
            else
                selectedBuddy = model.getBuddy(selectedIndex);
            if (selectedBuddy == null)
                return;
            BuddyDetailsDialog dialog = new BuddyDetailsDialog(this,
                              "Edit Buddy Details", true, selectedBuddy);
            dialog.setVisible(true);
        }
    }
```

Of course, there will be a problem when the user cancels the editing since the **dialogCancelled()** method removes the last buddy in the list! We do not want to do this removal if we are merely editing. To avoid this, we can add a new instance variable to the class to determine whether or not we are in "add" mode or "edit" mode:

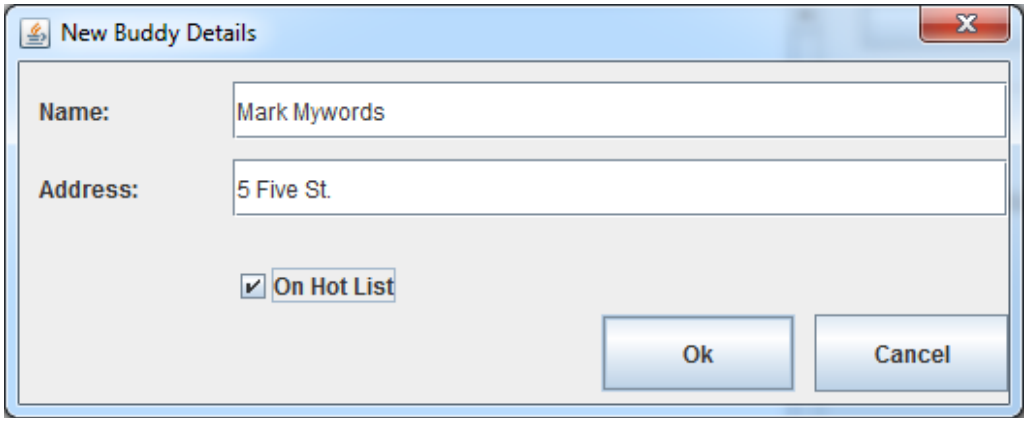
```
    // This is set to true if the dialog box was opened to add a new buddy
    // and is set to false if it was opened to edit a buddy
    private boolean                    inAddMode;
```

Then, we can set this value to true in the **addBuddy()** event handler (i.e., `inAddMode = true;`) and false in the **editBuddy()** event handler (i.e., `inAddMode = false;`). The **dialogCancelled()** method will need to change as well:

```
    public void dialogCancelled() {
        // Remove the latest buddy that was added if in add mode
        if (inAddMode)
            model.remove(model.getSize()-1);
    }
```

That is it! Now, we need to create the dialog box itself. It should allow the user to set the name, address and hot list status for the buddy that it is working on (i.e., either a newly added buddy or one being edited).

Here is what the dialog box will look like:



Here is the code:

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

public class BuddyDetailsDialog extends JDialog {
    // This is a pointer to the email buddy that is being edited
    private EmailBuddy aBuddy;

    // These are the components of the dialog box
    private JLabel        aLabel;
    private JTextField    nameField;
    private JTextField    addressField;
    private JCheckBox     hotListButton;
    private JButton       okButton;
    private JButton       cancelButton;

    public BuddyDetailsDialog(JFrame owner, String title, boolean modal,
                              EmailBuddy bud){
        super(owner,title,modal);
        aBuddy = bud;

        // Put all the components onto the window and given them initial values
        buildDialogWindow(aBuddy);

        // Add listeners for the Ok and Cancel buttons as well as window closing
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event){
                okButtonClicked();
            }});

        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                cancelButtonClicked();
            }});

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                cancelButtonClicked();
            }});

        setSize(526, 214);
```

```
        setLocationRelativeTo(owner);
    }

    private void buildDialogWindow(EmailBuddy aBuddy) {
        setLayout(null);

        // Add the name label
        aLabel = new JLabel("Name:");
        aLabel.setLocation(10,10);
        aLabel.setSize(80, 30);
        add(aLabel);
        // Add the name field
        nameField = new JTextField(aBuddy.getName());
        nameField.setLocation(110, 10);
        nameField.setSize(400, 30);
        add(nameField);
        // Add the address label
        aLabel = new JLabel("Address:");
        aLabel.setHorizontalAlignment(JLabel.LEFT);
        aLabel.setLocation(10,50);
        aLabel.setSize(80, 30);
        add(aLabel);
        // Add the address field
        addressField = new JTextField(aBuddy.getAddress());
        addressField.setLocation(110, 50);
        addressField.setSize(400, 30);
        add(addressField);
        // Add the onHotList button
        hotListButton = new JCheckBox("On Hot List");
        hotListButton.setSelected(aBuddy.onHotList());
        hotListButton.setLocation(110, 100);
        hotListButton.setSize(120, 30);
        add(hotListButton);
        // Add the Ok button
        okButton = new JButton("Ok");
        okButton.setLocation(300, 130);
        okButton.setSize(100, 40);
        add(okButton);
        // Add the Cancel button
        cancelButton = new JButton("Cancel");
        cancelButton.setLocation(410, 130);
        cancelButton.setSize(100, 40);
        add(cancelButton);
    }

    private void okButtonClicked(){
        aBuddy.setName(nameField.getText());
        aBuddy.setAddress(addressField.getText());
        aBuddy.onHotList(hotListButton.isSelected());
        if (getOwner() != null)
            ((DialogClientInterface)getOwner()).dialogFinished();
        dispose();
    }

    private void cancelButtonClicked(){
        if (getOwner() != null)
            ((DialogClientInterface)getOwner()).dialogCancelled();
        dispose();
    }
}
```

This page was intentionally left blank.