Chapter 5

Graphical User Interfaces

What is in This Chapter ?

This chapter discusses developing JAVA applications that bring up windows that the user can interact with. It discusses **Graphical User Interfaces** and how we can develop our own to represent main windows for our JAVA applications.

5.1 User Interfaces

All applications require some kind of ***user interface*** which allows the user to interact with the underlying program/software. Most user interfaces have the ability to take-in information from the user and also to provide visual or audible information back to the user.

In the *real* world, an interface often has physical interactive components . For example, there are two obvious ways to interact with (or *interface* with) our bank account. We may go up to a teller at the bank and perform some transactions, or we may use an ATM.

In the *virtual* world, we may interact with our bank account electronically by using a web browser, or phone app or dedicated stand-alone software from the bank.

In this case, the interaction is through software menus, buttons, text fields, lists, etc..

In this course, we will consider software-based user interfaces such as those shown above. We will concentrate on stand-alone applications that do not require the use of a browser.

Up until this point, our programs/applications did not really have any interactive user interface. That is, we defined classes, wrote programs and then simply ran them inside of the IDE that we were using. The results of our program were displayed as graphics (in the case of Processing) or as text output (as in the case of the JAVA console window).

When writing programs that bring up a main interactive window (or those that run on a phone or in a browser), it is important to understand that more is "going on behind the scenes". That is, when we use an ATM machine, there is actually quite a bit "going on" in the way that our actions at the machine affect the current state of the bank and of our bank account. For example, the bank account changes, transaction logs are updated and security/error-checking is taking place. The ATM machine was simply making use of these underlying entities (i.e., the **bank** and the **account**) in order to complete the transaction. It is necessary at this point to bring up some definitions:

> *The **model** of an application consists of all classes that represent the "business logic" part of the application ... the underlying system on which a user interface is attached.*

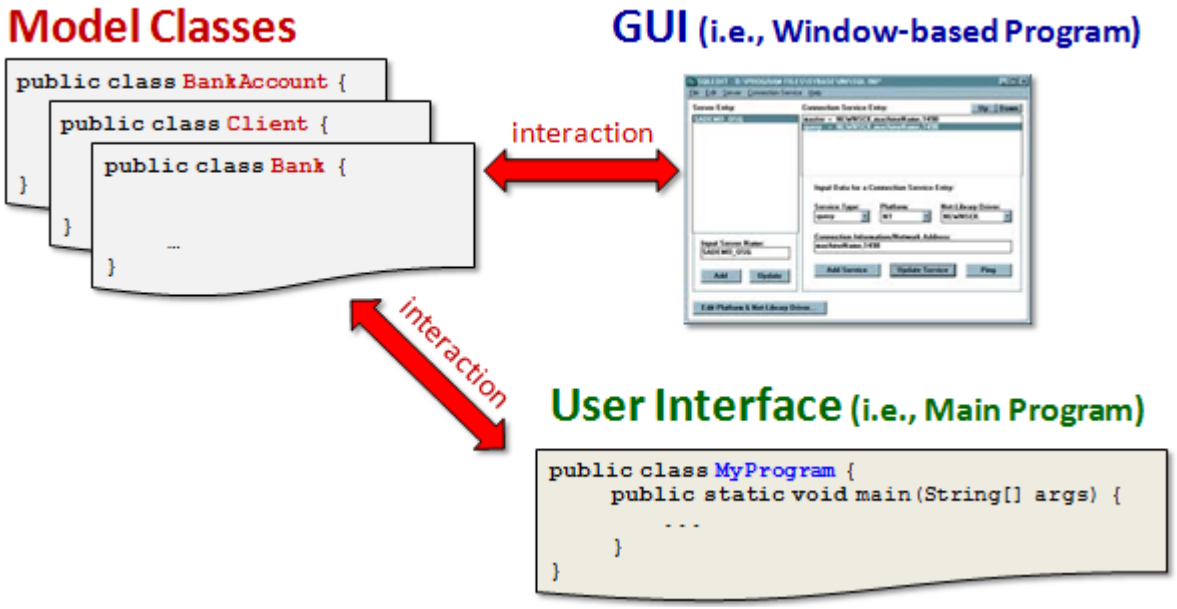
The model is always developed separately from the user interface. In fact, it should not assume any knowledge about the user interface at all (e.g., model classes should not assume that **System.out.println()** is available).

> *The **user interface** is the part of the application that is attached to the model which handles interaction with the user and does NOT deal with the business logic.*

The user interface always makes use of the model classes and often causes the model to change according to user interaction. The changes to the model are often reflected back (visually) on the user interface as a form of immediate feedback.

> *A **graphical user interface (GUI)** is a user interface that makes use of one or more windows to interact with the user.*

A GUI is often preferred over text-based user interfaces because it is more natural ... more like real-world applications that we are used to using. Imagine, for example, if the internet was only text-based with no buttons, text fields, drop-down lists, images, etc..



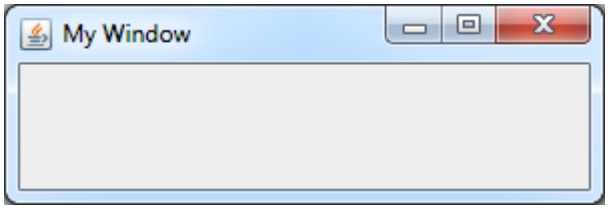
So, it is important to understand that there should always be a separation in your code between the *model* classes and the *user interface* classes. That will allow you to share the same model classes with different interfaces.

In JAVA, all of our main windows for our applications will be instances of the **JFrame** class. A **JFrame** will be the main starting place for our user interface-based programs. The JFrame window will contain window *components* that will allow us to interact with the user, such as buttons, text fields, lists etc...

Example:

The following code creates a simple window in JAVA. You can use this program as a template for all of your window-based applications:

```
import   javax.swing.JFrame;

public class MyApplication extends JFrame {

    public MyApplication(String title) {
        super(title);                               // Set title of window
        setDefaultCloseOperation(EXIT_ON_CLOSE);    // allow window to close
        setSize(300, 100);                          // Set size of window
    }

    public static void main(String[] args) {
        MyApplication frame;

        frame = new MyApplication("My Window");         // Create window
        frame.setVisible(true);                         // Show window
    }
}
```



Although this code for bringing up a new window can appear anywhere, we typically designate a whole class to represent the window (that is, the JAVA application). This also helps to separate the model and the user interface. So here are the steps involved with creating your own JAVA application that uses a main window (frame):

1. Create a new class (separate from model classes) to represent your application. In this case we created a **MyApplication.java** file to represent our window-based application.

2. Have this class extend **JFrame.** Make sure to *import* the **javax.swing.JFrame** package. This will allow us to inherit all of the JFrame's attributes and methods.

3. Create a constructor that sets the window title (specified as a parameter). In the constructor, set the size of the window using JFrame's **setSize()** method by specifying the width and height of the window (in pixels ... includes the frame around the window). If you do not set the size, the window will show up so small that you will only see part of the title bar.

4. Include a **main()** method as a starting point for the application that calls your constructor to make the window and then call the frame's **setVisible()** method with a value of **true** to make the window appear. When frames are created, by default they do not appear on the screen unless we call this method with a value of **true**.

Note as well that we specified for the application to **EXIT_ON_CLOSE**. This is necessary since we want the application to stop running when the window is closed. This is typical behavior for all applications that run under windowing operating system environments. What other choices do we have when the window is closed ? b We could have used:

```
// window is not closed
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);

// window is hidden...the program keeps running
setDefaultCloseOperation(HIDE_ON_CLOSE);

// window is hidden and disposed of (more later)
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

To test the application, just compile and run it as you normally do.

5.2 Components and Containers

In order for a window to be useful, it must contain various components.

> *A **window component** is an object with a visual representation that is placed on a window and usually allows the user to interact with it.*

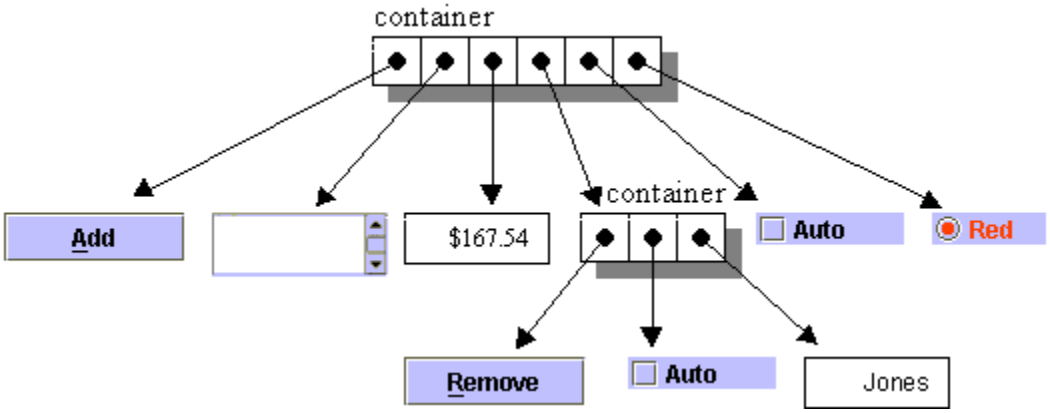
Typical window components are things such as buttons, text fields, drop down lists, scroll bars, tabbed panes, menus, etc..

Components may also be grouped together, much like adding elements to an array. In fact, most components in JAVA can contain other components as their sub-parts. This brings up the notion of a ***Container***.

> *A **container** is an object that contains components and/or other containers.*

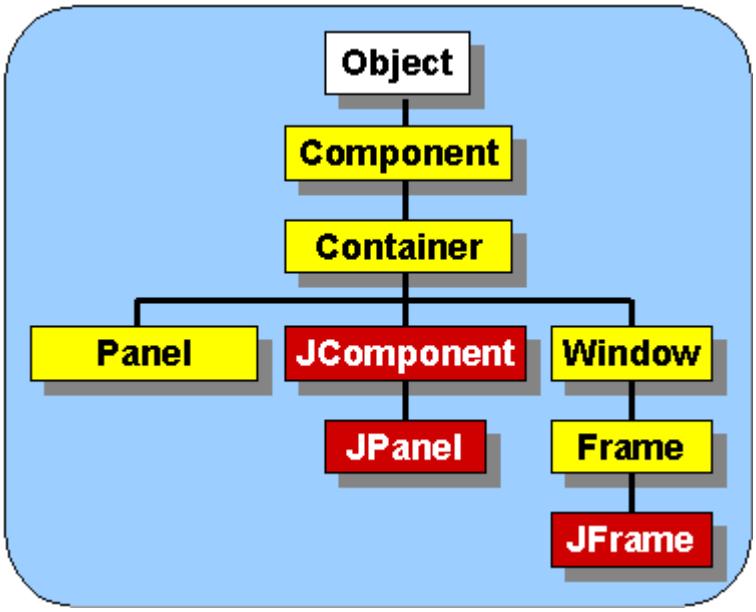
The most easily understood container in JAVA is the **JFrame**, since windows generally contain many components on them. However, as you will see, there are other useful containers such as a **JPanel**. Containers are actually *components* as well. That is, you can have containers which contain other containers.

This allows for arrangements such as a **JFrame** containing two **JPanels**, each of which contains two **JLabels** etc.. So, a container of components is conceptually "like" an array of objects:



In the original versions of JAVA, all components were stored in the **java.awt** package. This package, and its sub-packages were known as the **Abstract Windowing Toolkit** (AWT). Soon after, the JAVA guys developed a more platform-independent version in the **javax.swing** package. This package, and its sub-packages were known as the **Swing** packages. We will be making use of these newer **Swing** components.

Here is just a portion of the component hierarchy →

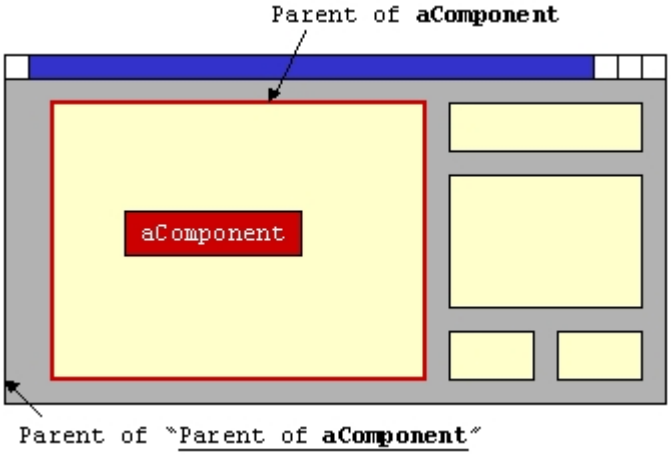


The **red** classes are classes in the swing packages, while the **yellow** ones are classes from the AWT packages:

Notice that a **JFrame** is a **Window** as well as a **Container**. **JPanels** are also **Containers** in that they too can contain many components.

In fact, **Containers** themselves are **Components** and all **JComponents** are also **Containers**. So, in Swing, everything is a **Container** ... even a button!

All components actually keep pointers to their ***parent*** container (i.e., the component that contains it). Parents of nested components are stored recursively:




```
Component   aComponent;
Container   parent;
Container   parentOfParent;

aComponent = ... ;
parent = c.getParent();
parentOfParent = p.getParent();
```

Containers themselves keep pointers to their components (i.e., an array) and we can access these using the container's **getComponent()** and **getComponents()** methods:

```
Container   aParent = ... ;
Component   c1 = aParent.getComponent(0);
Component   c2 = aParent.getComponent(1);
Component   c3 = aParent.getComponent(2);

Component[]  c = aParent.getComponents(); // get them all
```

One of the most commonly used containers is called a ***JPanel***:

*A **JPanel** is a frameless area on a window that usually contains a laid-out arrangement of other components.*

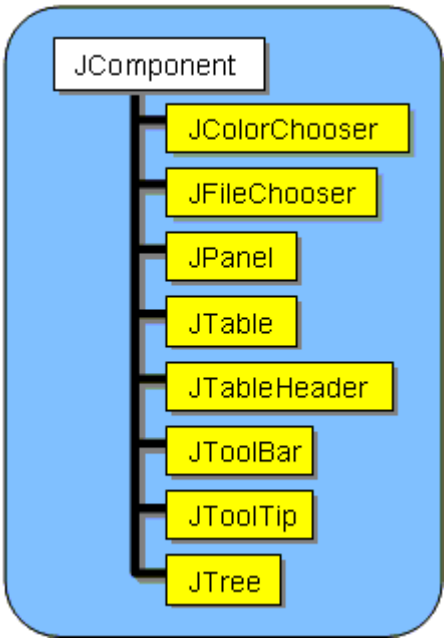
Think of a panel as a bulletin board that you can fill with components and then you can place the bulletin board anywhere as a component itself.

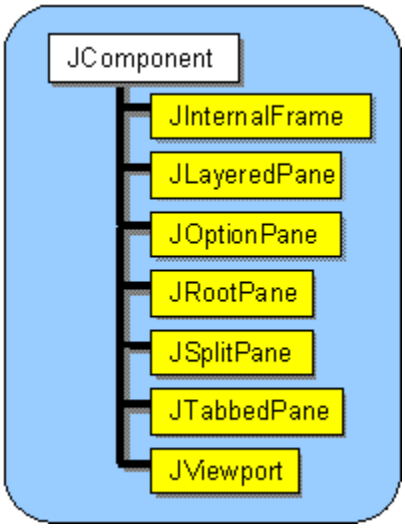
All **JFrames** have a **JPanel** at the top level to which all of our window components added. For simple "single-panel" windows, we can simply access this panel by sending the **getContentPane()** method to our **JFrame**, and then call **add()** to put components onto it:

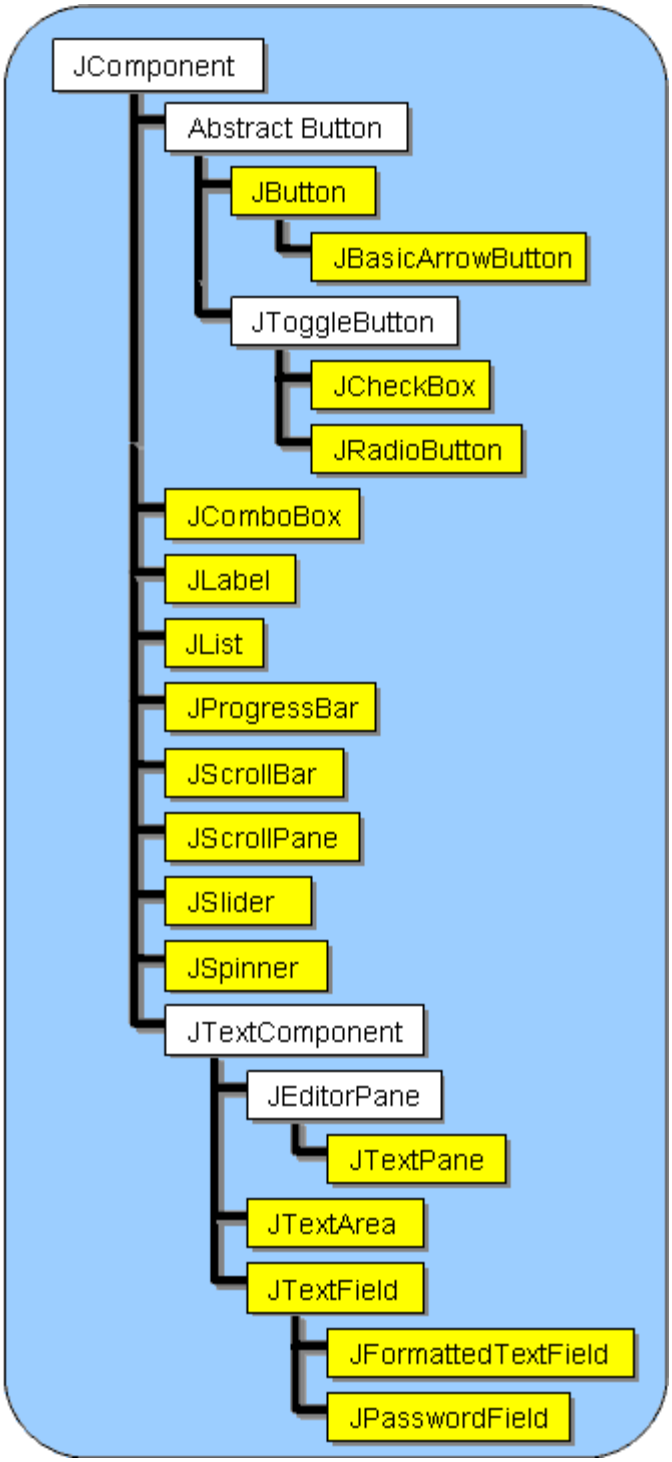
```
Component   c1 = ...;
Component   c2 = ...;
Component   c3 = ...;

JFrame frame = new JFrame("MyApplication");
frame.getContentPane().add(c1);
frame.getContentPane().add(c2);
frame.getContentPane().add(c3);
```

Note that the component hierarchy diagram shown earlier does not list all available components. We can expand the **JCompoment** hierarchy in more detail.

The three pictures on the next page (note that there was too much to show on one picture so it has been split into multiple pictures) show just some of the **JComponent** subclasses that represent commonly used components (shown in yellow) that are placed onto windows. Notice that all these **JComponents** start with a **"J"**. Also notice that there are different kinds of buttons and text-based components.



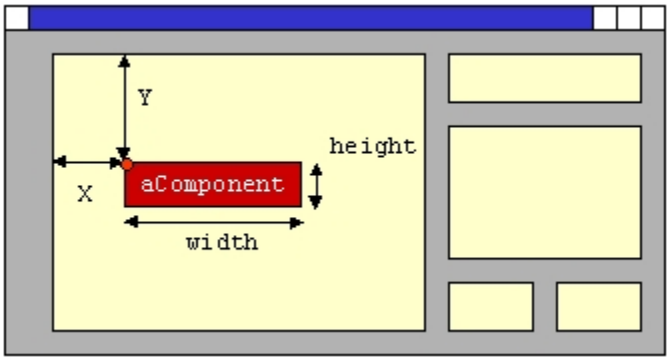




There are even more subclasses ... those dealing with menus will be shown later.

You will get to know more about these components as we use them in our examples. For now, it is a good idea to understand the common properties and functionality that all components have.

All **JComponents** have an (x, y) location as well as a width and height. The location is the (x,y) coordinate of the top/left of the component. It is a coordinate within the coordinate system defined by the top left corner of the parent container (which is often a **JPanel**):



We can access or modify these values for the component at any time using the methods shown in the code below:

```
JComponent   c = ... ;

// ask a component for its location, its width and its height
int x = c.getX();
int y = c.getY();
int w = c.getWidth();
int h = c.getHeight();

// change a component's location
c.setLocation(new Point(100, 200));

// change a component's width and height
c.setSize(100, 50);
```

There is a problem with changing sizes and locations of components by default. JAVA has "Layout Managers" (more on this later) that automatically compute component locations and sizes. If we decide to use layout managers, we can "suggest" sizes for our components using the following methods:

```
c.setMaximumSize(new Dimension(width, height));
c.setMinimumSize(new Dimension(width, height));
c.setPreferredSize(new Dimension(width, height));
```

In addition to setting the size and dimensions of our components, we can also set the background and foreground colors of the component. The background color is the color that fills in the background of the component which is behind any text that appears on the component. The foreground color is the text color. Here, for example is a window with 4 JButtons on it, each with different background and foreground colors:

The code for setting the color of a component is simple:

```
JComponent   c = ... ;

c.setBackground(Color.red);
c.setForeground(Color.black);
```

There are a few "standards" colors definitions. Here are some of them:

```
Color.black    Color.green     Color.pink
Color.blue     Color.lightGray Color.red
Color.cyan     Color.magenta   Color.white
Color.darkGray Color.orange    Color.yellow
Color.gray
```

To use the above constants, you need to import the **Color** class which is located within the **java.awt** package using: **import java.awt.Color;**

However, these are not usually the best colors to use for a nice user interface, as they may "clash", resulting in an "ugly" user interface. You will usually want to define your own colors. The **Color** class provides constructors that allow you to create your own colors by specifying the amount of Red, Green and Blue that makes up the color. This can be done by either using integers (between 0 and 255) or floats (between 0.0 and 1.0) to represent a percentage of RGB values:

```
new Color(int r, int g, int b);
new Color(float r, float g, float b);
```

Here is an example:

```
JComponent   c = ... ;

c.setBackground(new Color(100, 50, 0));  // brown
c.setForeground(new Color(70, 0, 70));   // dark purple
```

In addition to setting the color, you may also choose the Font type (i.e., type of text) and Font size that appears on your component (e.g., button, text field, etc..). The **setFont()** method is used to do this. You need to supply a **Font** object.

```
JComponent  c = ... ;
Font        f = ...;

c.setFont(f);
```

You can create a **Font** object by calling a constructor in the **Font** class (located in the **java.awt** package). Here is one constructor that requires you to supply the name of the font, the style and the point size:

```
f = new Font(String name, int sytle, int size);
```

So, as an example, here is what you could write:

```
c.setFont(new Font("SansSerif", Font.BOLD, 12));
```

For the font **name**, you can supply any font that your system has available such as:

```
"Times", "Arial", "SansSerif" or "Courier"
```

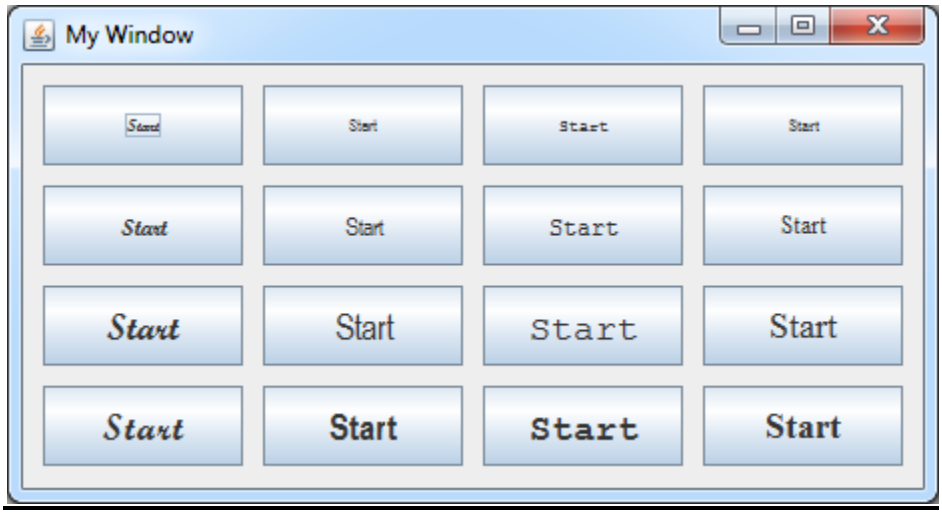
To get a list of all fonts available on your computer, you can run the following code:

```
java.awt.GraphicsEnvironment     ge;
java.awt.Font[]                  fonts;

ge = java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment();
fonts = ge.getAllFonts();
for(int i=0; i<fonts.length; i++) {
     System.out.println(fonts[i].getName());
}
```

Here are possible **styles** (notice that we can "OR" them together using JAVA's bitwise OR operator |):

```
Font.BOLD,  Font.ITALIC,  Font.PLAIN,  Font.BOLD|Font.ITALIC
```

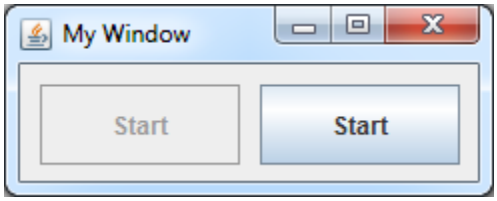
Here is an example showing buttons with various font names, style and size. The rows show font sizes 8, 12, 18 and 18, respectively. The first 3 rows have PLAIN style while the last row had BOLD style. The columns (from left to right) show font types: **Script MT Bold**, **Arial Narrow**, **Courier** and **Times New Roman**, respectively:



In addition to these visual parameters, we can also enable and disable components. A component that is disabled is "grayed out" and not able to be controlled by the user. A disabled button, for example, cannot be pressed. A disabled text field, for example, does not allow the user to type text into it.

We can enable and disable components at any time in our program by using the **setEnabled()** method as follows:

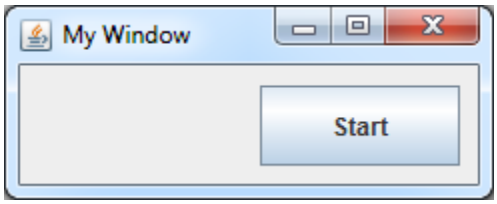
```
JComponent   c = ... ;

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



By default, all components are enabled and will remain that way unless you disable them.

Components can actually also be completely hidden from view. A **setVisible()** method is used to show or hide a component:

```
JComponent   c = ... ;

c.setVisible(false);
...
c.setVisible(true);
```



By default, all newly created components are visible (i.e., not hidden). However, **JFrames** are NOT automatically made visible ... you MUST do **setVisible(true)** to have your window appear.

There are many more attributes that we can set for components and we will investigate some more of them throughout the course.

Example:

Consider writing a program that creates the window shown in this rough diagram →

The top/left component is a **JTextField**. The bottom/left component is a **JList** and the right two components are **JButtons** to add and remove items to the list.

It is always a good idea to draw out a rough version of your user interface, identifying the top/left corner or all components as well as their dimensions and the width or all margins around the window border and between the components.

To write the code, we begin with a basic class definition. Notice the line that says **setLayout(null).** This allows us to manually place the components wherever we want to on the screen. Notice as well that we set the size of the window according to the dimensions in the picture with an extra 10 pixels wide to account for the window frame and an extra 25 pixels

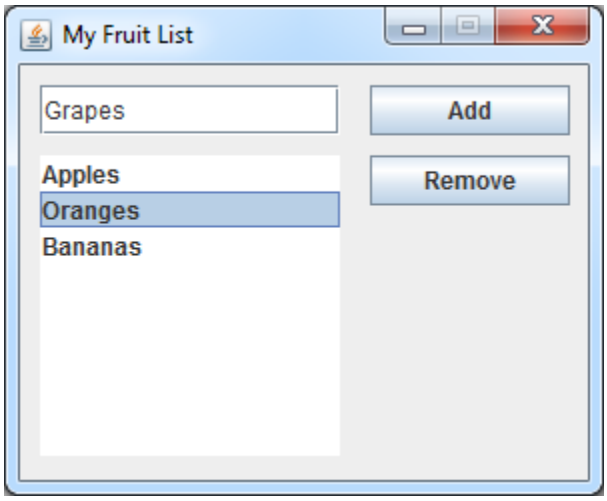
high to account for the title bar of the window. We also made use of the **setResizable(false)** so that the window cannot be resized after we create it.

```
import javax.swing.*;

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

        // Choose to lay out components manually
        getContentPane().setLayout(null);

        // ... Add components here (see below) ...

        // Set program to stop when window closed
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(290, 230); // manually computed sizes
        setResizable(false);
    }

    public static void main(String[] args) {
       JFrame frame = new FruitListApp("My Fruit List");
       frame.setVisible(true);
    }
}
```

Now, we need to place our components on the window. Just insert the following code into the middle of the above constructor:

```
// Add the text field
JTextField newItemField = new JTextField("Grapes");
newItemField.setLocation(10,10);
newItemField.setSize(150,25);
getContentPane().add(newItemField);

// Add the ADD button
JButton addButton = new JButton("Add");
addButton.setLocation(175, 10);
addButton.setSize(100,25);
getContentPane().add(addButton);

// Add the REMOVE button
JButton removeButton = new JButton("Remove");
removeButton.setLocation(175,45);
removeButton.setSize(100,25);
getContentPane().add(removeButton);

// Add the JList
String[] fruits = {"Apples", "Oranges", "Bananas"};
JList fruitList = new JList(fruits);
fruitList.setLocation(10,45);
fruitList.setSize(150,150);
getContentPane().add(fruitList);
```

Here is what the window will look like when we run the code →



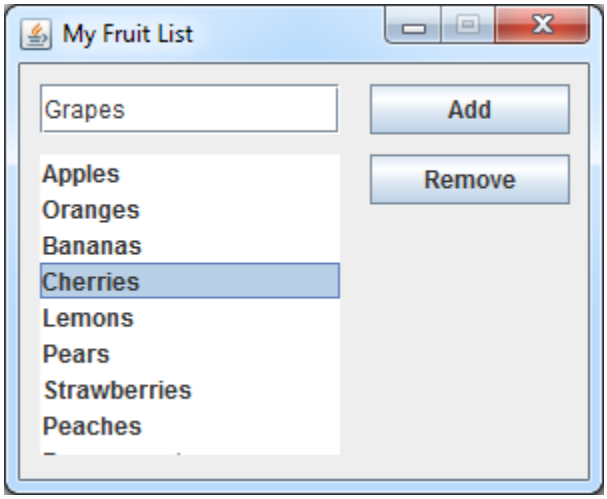
You will notice that the text supplied in the **JTextField** constructor is the initial text that will appear in the text field when the window opens.

Also, the text supplied for the **JButtons** is the text that will appear on the button itself.

Lastly, notice that we can supply a list of items to place in the **JList** initially as an array of **String** objects. In fact, we can supply any objects for this list, but for now we will simply use Strings.

The code above adds a list with 3 items in it initially. If, however, the list had many items in it, the items cannot all be shown. For example, here is what the list would look like if we added these fruits:

```
String[] fruits = {
   "Apples", "Oranges", "Bananas",
   "Cherries", "Lemons", "Pears",
   "Strawberries", "Peaches",
   "Pomegranates", "Nectarines",
   "Apricots"};
```



Notice that the list cannot display more than 8 items due to its small size. In such as situation, we need scroll bars on our list so that the user can scroll to find particular items.

JAVA has a **JScrollPane** class that allows us to place components within it such that scroll bars are automatically placed in a way that allows scrolling in both the horizontal and vertical directions.

To use the scroll pane, we call the **JScrollPane** constructor with 3 parameters. The first parameter is the **JList** that we want to be able to scroll through (e.g., **fruitList** in our example). The next two parameters are the scroll policies for the vertical and horizontal scroll bars. For each scroll bar we must choose one of the following three policies for the vertical scroll bar:

```
ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER
ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED
```

... and choose one of these for the horizontal scroll bar:

```
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

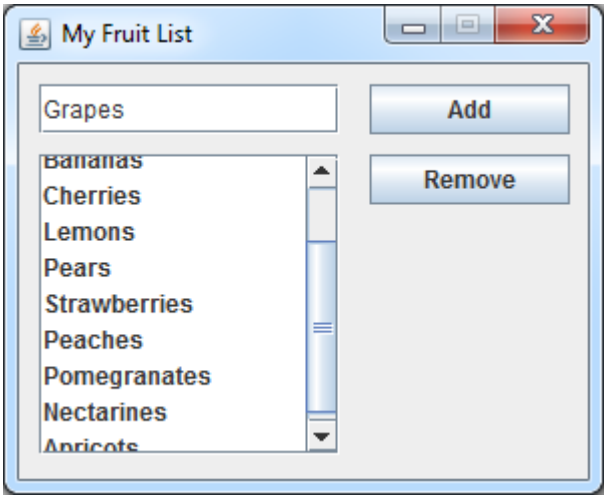
The `VERTICAL_SCROLLBAR_AS_NEEDED` option will show no scroll bar unless one is needed.

That means, if there are not enough items in the list to require scrolling, the vertical scroll bar will not appear. Similarly, for horizontal scrolling, if all our items are short strings, the `HORIZONTAL_SCROLLBAR_AS_NEEDED` option will not show the scroll bar unless a longer item appeared in the list that cannot fit horizontally in the list.

As a result here is the new code:

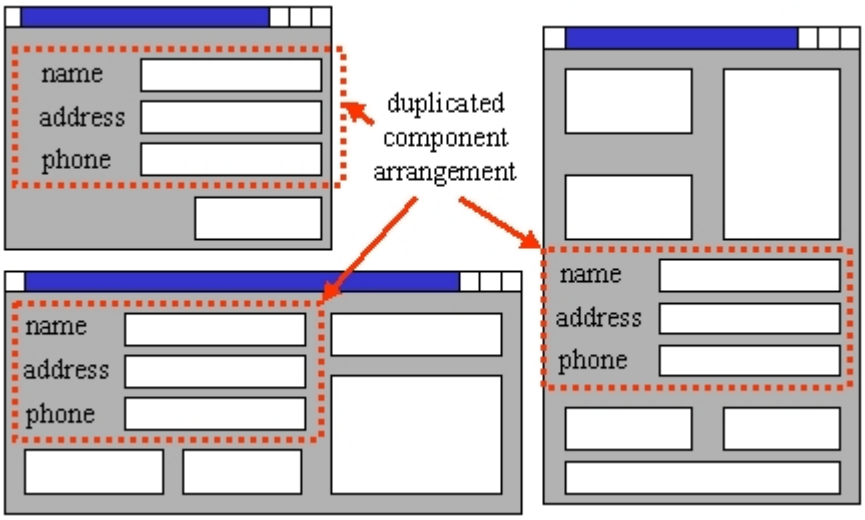
```
String[] fruits = {"Apples", "Oranges", "Bananas", "Cherries",
                   "Lemons", "Pears", "Strawberries", "Peaches",
                   "Pomegranates", "Nectarines", "Apricots"};
JList fruitList = new JList(fruits);

JScrollPane scrollPane = new JScrollPane(fruitList,
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);

scrollPane.setLocation(10,45);
scrollPane.setSize(150,150);
getContentPane().add(scrollPane);
```

And here is the resulting window:



5.3 Grouping Components Together

It is a very good idea to keep your window components organized. It is often the case that an arrangement of components may be similar (or duplicated) within different windows.

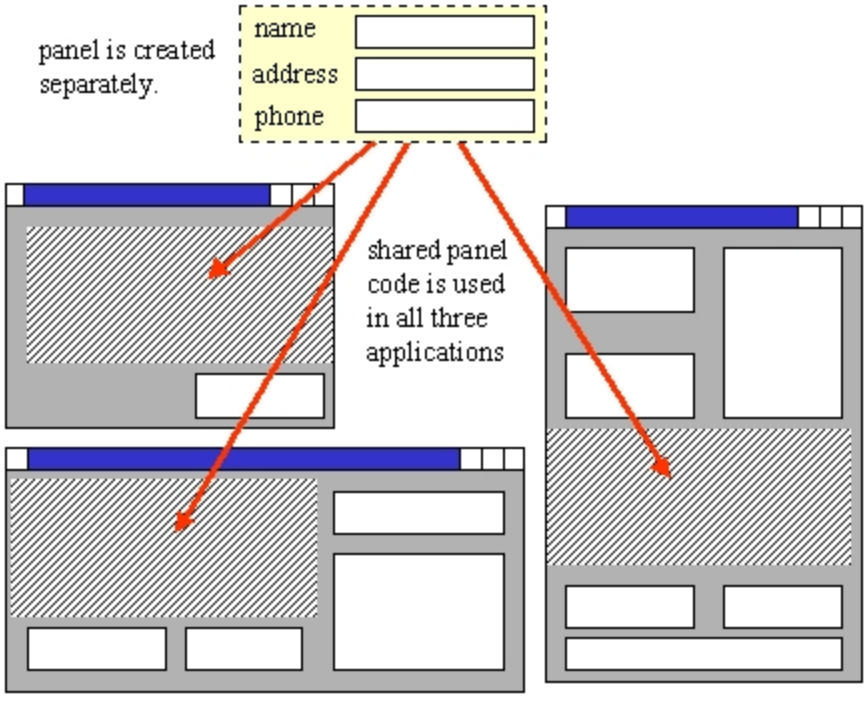


For example, an application may require a name, address and phone number to be entered at different times in different windows →

It is a good idea to share component layouts among the similar windows within an application so that the amount of code you write is reduced.

To do this, we often lay out components onto a **JPanel** and then place the **JPanel** on our window. We can place the created panel on many different windows with one line of code ... this can dramatically reduce the amount of GUI code that you have to write.

So, you will often want to create separate **JPanels** to contain groups of components so that you can move them around (as a group) to different parts of a window or even be shared between different windows. The code to do this simply involves creating our **JPanel** with its appropriate component arrangement and then adding the **JPanel** to the **JFrame**.



On the next page is an example that takes two **JPanels** with 2 components each on them and then adds them to a **JFrame**. Note that some coding details have been left out purposely:

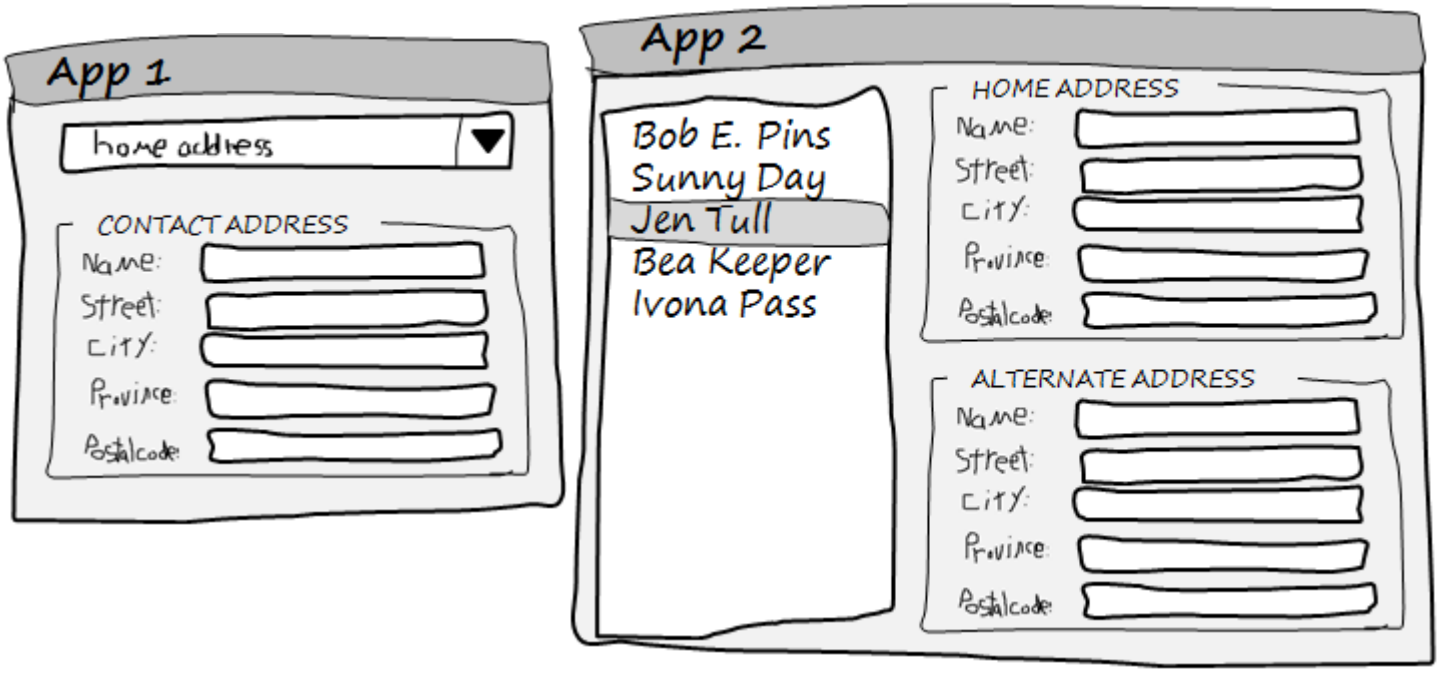
```
Component   c1, c2, c3, c4, c5;
JPanel      p1, p2;

/* ... code omitted for creating the components and panels ... */

JFrame frame = new JFrame("MyApplication");
p1.add(c1);
p1.add(c2);
p2.add(c3);
p2.add(c4);
frame.getContentPane().add(p1);
frame.getContentPane().add(p2);
frame.getContentPane().add(c5);
```

Notice how components c1 and c2 are added to panel p1 and then c3 and c4 are added to panel p2. Then these panels are simply added to the frame's content pane as if they were simply components. The following example is more specific.

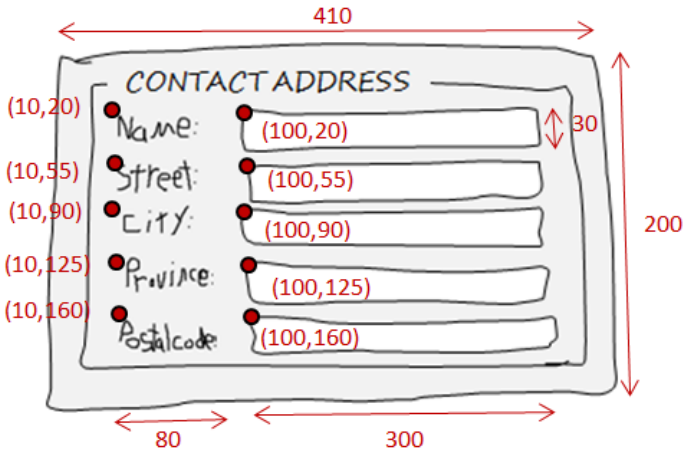
Example:

Consider another example in which a **JPanel** is used in more than one window. We will create a simple panel called **AddressPanel** that contains 5 labels and 5 text fields for allowing the user to enter a name and address as shown here →

The panel contains 5 **JTextField** objects that allow the user to fill-in an address. It also has 5 **JLabel** objects (which are simply pieces of text) to indicate the kind of data expected for each text field. Lastly, there is a nice border around the **JPanel** which has the title CONTACT ADDRESS. This panel will not be its own window. Instead, it will be a panel inside of two other windows that look as shown here. Notice that each application has the same kind of **AddressPanel**, except that the border's title varies.



We will need to compute the locations and sizes for each component. Note that the CONTACT ADDRESS is not a **JLabel** but is actually part of the panel's border, as you will soon see. Here are the dimensions →



To begin the code, we will want to define an **AddressPanel** class. It will be a special kind of **JPanel**, and so it should be a subclass of **JPanel**. The code should have the following framework:

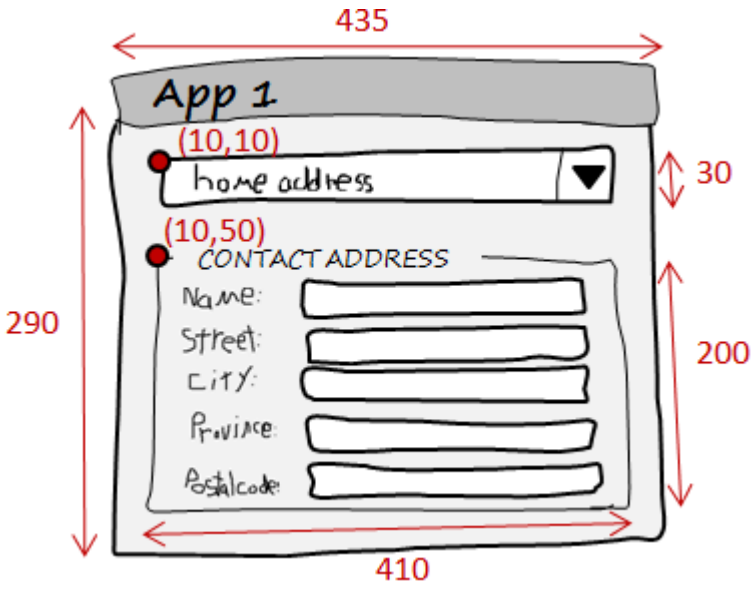
```
import javax.swing.*;

public class AddressPanel extends JPanel {
    public AddressPanel(String title) {
        // Choose to lay out components manually
        setLayout(null);

        // Make a border around the outside with the given title
        setBorder(BorderFactory.createTitledBorder(title));

        // ... Add components here (see below) ...

        setSize(410, 200);
    }
}
```

You may notice that the **AddressPanel** takes a constructor that allows the user to pass in a String to be the title on the border. This title is used in the call to **setBorder()** in order to make the appropriate border with that title. The components are then added to the window one-by-one as follows:

```
        // Add the Name, Street, City, Province and PostalCode labels
        JLabel aLabel = new JLabel("Name:");
        aLabel.setLocation(10, 20);
        aLabel.setSize(80,30);
        add(aLabel);

        aLabel = new JLabel("Street:");
        aLabel.setLocation(10, 55);
        aLabel.setSize(80,30);
        add(aLabel);

        aLabel = new JLabel("City:");
        aLabel.setLocation(10, 90);
        aLabel.setSize(80,30);
        add(aLabel);

        aLabel = new JLabel("Province:");
        aLabel.setLocation(10, 125);
        aLabel.setSize(80,30);
        add(aLabel);
```

```
        aLabel = new JLabel("Postal Code:");
        aLabel.setLocation(10, 160);
        aLabel.setSize(80,30);
        add(aLabel);

        // Add the name textfield
        JTextField nameField = new JTextField();
        nameField.setLocation(100, 20);
        nameField.setSize(300,30);
        add(nameField);

        // Add the street textfield
        JTextField streetField = new JTextField();
        streetField.setLocation(100, 55);
        streetField.setSize(300,30);
        add(streetField);

        // Add the city textfield
        JTextField cityField = new JTextField();
        cityField.setLocation(100, 90);
        cityField.setSize(300,30);
        add(cityField);

        // Add the province textfield
        JTextField provinceField = new JTextField();
        provinceField.setLocation(100, 125);
        provinceField.setSize(300,30);
        add(provinceField);

        // Add the postal code textfield
        JTextField postalField = new JTextField();
        postalField.setLocation(100, 160);
        postalField.setSize(300,30);
        add(postalField);
```

The **AddressPanel** itself is not a runable application (i.e., there is no **main** method). So we will now make our App1 application to test it out. Here are the dimensions for the application →



Notice that the **AddressPanel** is now "treated" as a single component and is placed on the main window by specifying its location. We do not need to specify the size of the **AddressPanel**, since this was defined within the **AddressPanel** constructor and is fixed.

The topmost component is called a **JComboBox** and it represents what is known as a drop-down list. It is similar to a **JList**, except that only the selected item is shown and the remaining items can be show by pressing the black arrow.

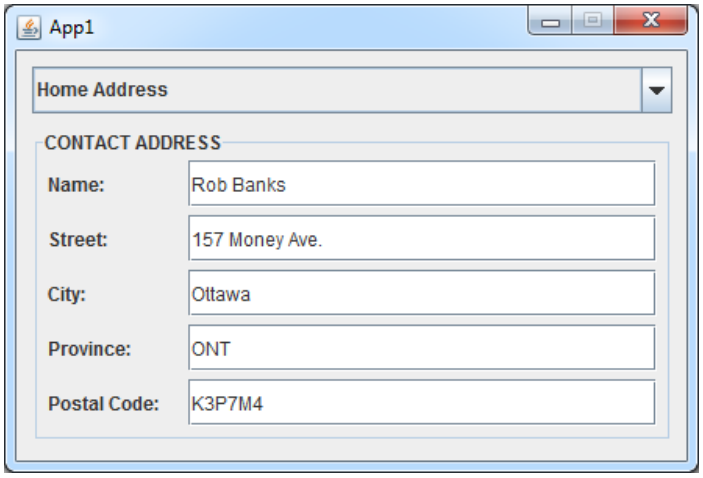
A **JComboBox** can be created by specifying an array of objects that are to appear in the list and passing this in as a parameter to the constructor as follows:

```
String[]   addresses = {"Home Address", "Work Address", "Alternate Address"};
JComboBox  addressBox1 = new JComboBox(addresses);
```

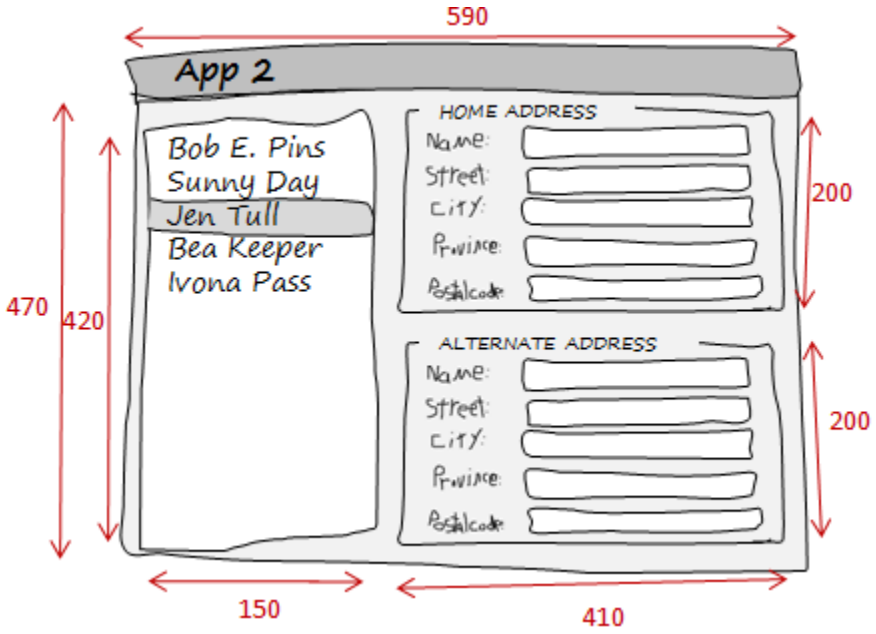
Here is the code to create the application. Take note of how the **AddressPanel** is now used as if it were a single, simple component:

```
import javax.swing.*;

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

        // Choose to lay out components manually
        getContentPane().setLayout(null);

        // Add the drop-down list
        String[] addresses = {"Home Address", "Work Address",
                              "Alternate Address"};
        JComboBox addressBox1 = new JComboBox(addresses);
        addressBox1.setLocation(10,10);
        addressBox1.setSize(410,30);
        getContentPane().add(addressBox1);

        // Now add an AddressPanel
        AddressPanel myPanel = new AddressPanel("CONTACT ADDRESS");
        myPanel.setLocation(10,50);
        getContentPane().add(myPanel);

        // Set program to stop when window closed
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(435, 290); // manually computed sizes
        setResizable(false);
    }

    public static void main(String[] args) {
       JFrame frame = new OneApp("App1");
       frame.setVisible(true);
    }
}
```

And here is the finished product →



Here are the dimensions for the 2nd application:

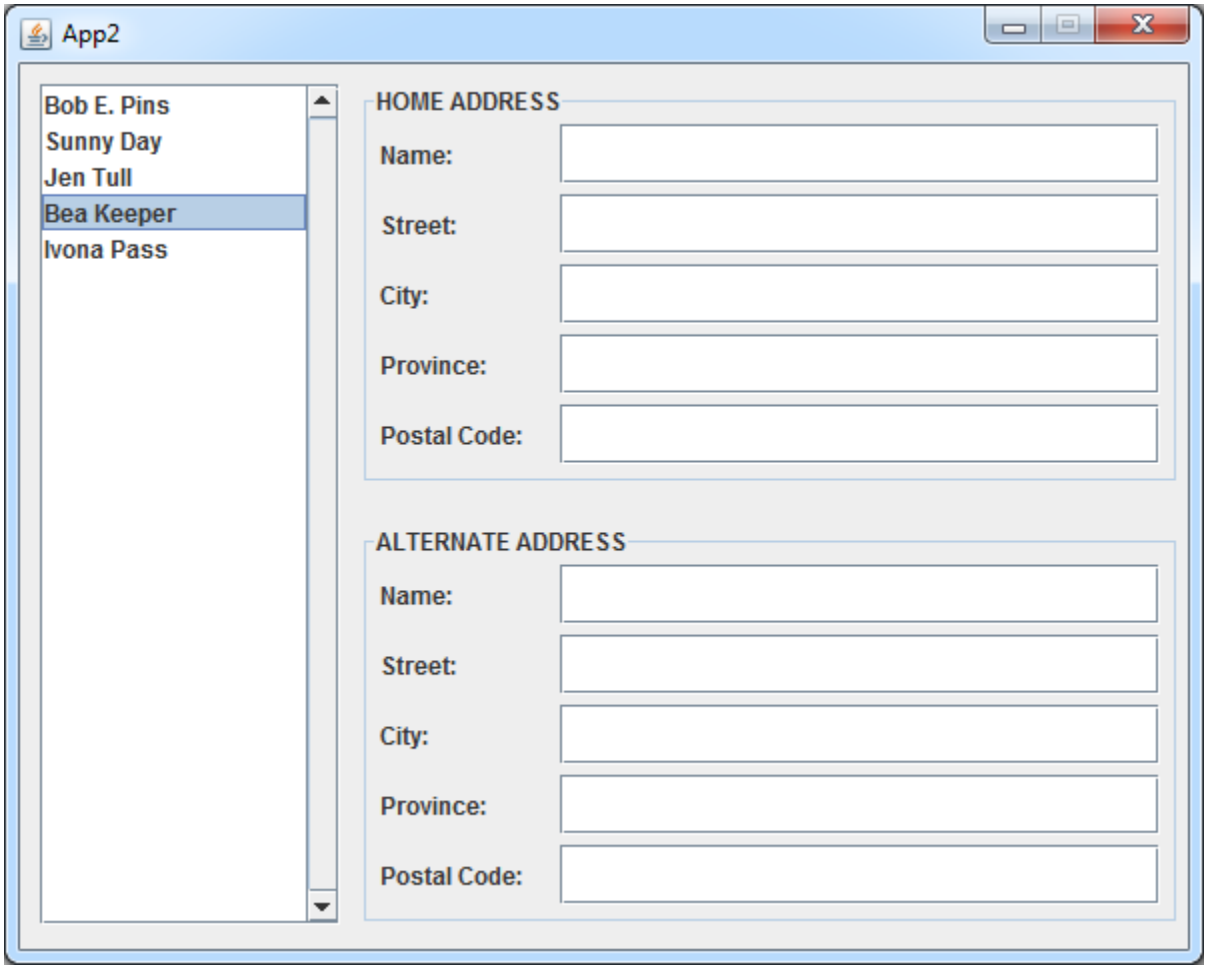


Here is the code. Again, notice how the **AddressPanel** is used twice in the same window with a different title:

```
import javax.swing.*;
public class TwoApp extends JFrame {
    public TwoApp(String name) {
        super(name);

        // Choose to lay out components manually
        getContentPane().setLayout(null);

        // Add the list of names
        String[] names = {"Bob E. Pins", "Sunny Day", "Jen Tull",
                          "Bea Keeper", "Ivona Pass"};
        JList aList = new JList(names);
        JScrollPane scrollPane = new JScrollPane(aList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        scrollPane.setLocation(10,10);
        scrollPane.setSize(150,420);
        getContentPane().add(scrollPane);

        // Now add an AddressPanel
        AddressPanel myPanel1 = new AddressPanel("HOME ADDRESS");
        myPanel1.setLocation(170,10);
        getContentPane().add(myPanel1);

        // Now add an another AddressPanel
        AddressPanel myPanel2 = new AddressPanel("ALTERNATE ADDRESS");
        myPanel2.setLocation(170,230);
        getContentPane().add(myPanel2);
```

```
            // Set program to stop when window closed
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setSize(590, 470); // manually computed sizes
            setResizable(false);
        }

        public static void main(String[] args) {
           JFrame frame = new TwoApp("App2");
           frame.setVisible(true);
        }
    }
```

And ... the result:



5.4 Event Handling

Now that we understand the basics of laying out various components on our graphical user interfaces, we need to discuss how to allow the user to interact with the window and make things happen.

Recall these definitions from the previous course:

> *An **event** is something that happens in the program based on some kind of triggering input which is typically caused (i.e., **generated**) by user interaction such as pressing a key on the keyboard, moving the mouse, or pressing a mouse button.*

> *An **event handler** is a procedure that specifies the code to be executed when a specific type of event occurs in the program.*

In the previous course, you may have written event handlers for handling events for when the mouse button was pressed, released or clicked as well as when the mouse was moved or dragged or when a key on the keyboard was pressed, released or clicked. As an example you may have written event handlers like this:

```
void mousePressed() {
  if (dist(x,y,mouseX,mouseY) < RADIUS)
    grabbed = true;
}

void mouseReleased() {
  if (grabbed) {
    direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = false;
}
```

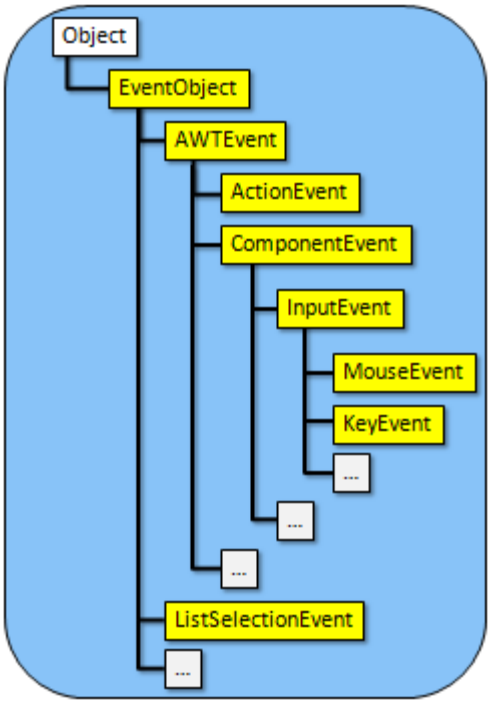
When writing GUIs we need to write specific event handlers in order for our application to respond to button clicks, allow selecting items from a list, allow typing into a text field etc...

There are two terms that we need to define:

> *The **source** of an event is the component for which the event was generated (i.e., when handling button clicks, the **Button** is the **source**).*

> *A **listener** is an event handler (i.e., also known as a **callback** procedure).*

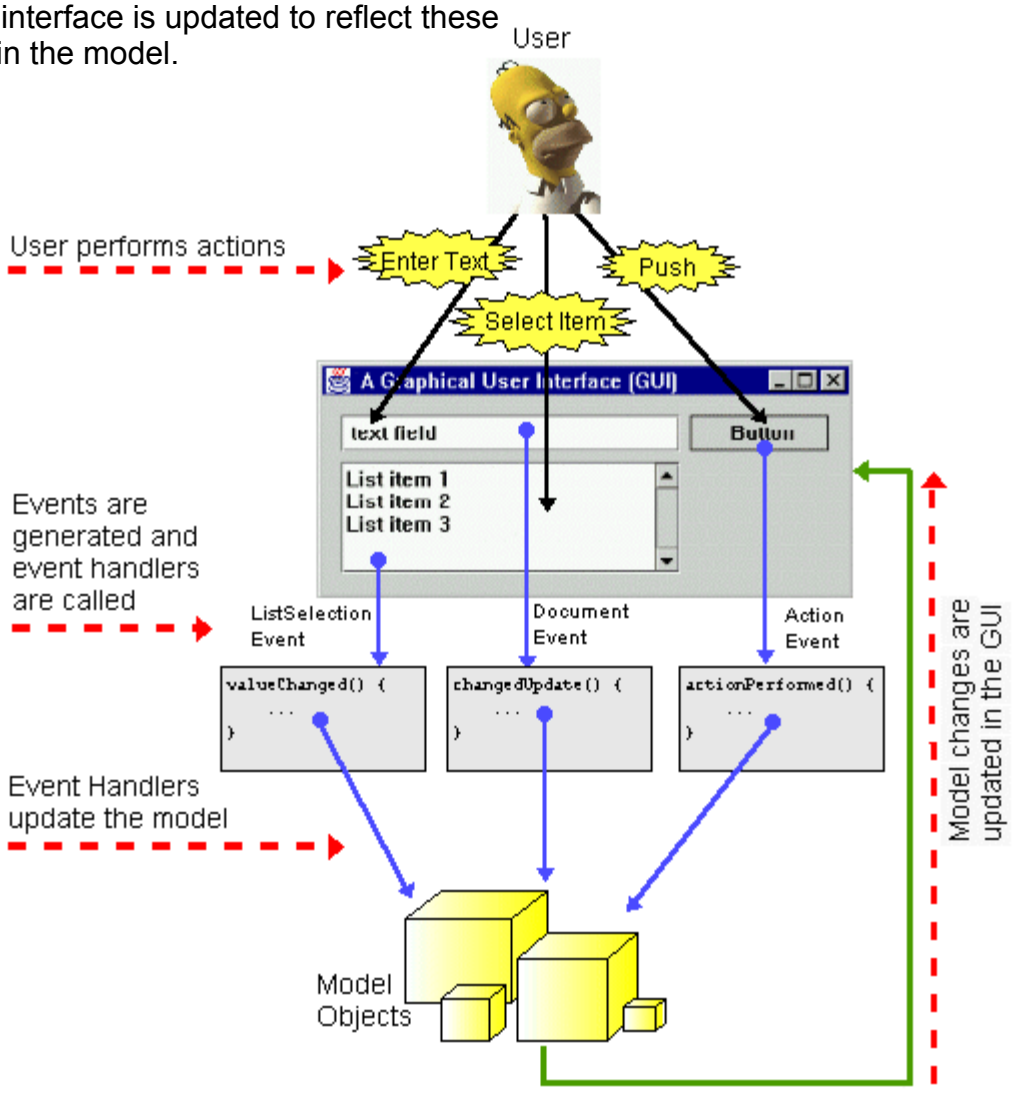
In JAVA, all **Events** are represented by a distinct class. There are many kinds of events, each having its own unique class. Here is a partial hierarchy showing some of the events that we will use in the course →



You do not need to memorize these, but you will become familiar with them during the course.

These events are generated when the user interacts with the user interface as follows:

1. The user causes an event by clicking a button, entering text, selecting a list item etc..
2. JAVA calls the appropriate event handler.
3. The event handling code changes the model in some way.
4. The user interface is updated to reflect these changes in the model.



Therefore, to perform event handling in JAVA, you must first identify the event that you want to handle. Then you need to write the appropriate event handlers. For each event, there is a corresponding *interface* in JAVA with a list of methods that you can write in order to handle the appropriate event in a meaningful way.

Below is a table of commonly-used events along with their Listener interfaces. This list basically tells you which methods need to be written in order to handle the kind of event that you are interested in. For a more complete description of these (and other) events, listeners and their methods, see the JAVA API specifications.

Event	Interface to Implement
ActionEvent - generated when button pressed, menu item selected, enter key pressed in a text field or from a timer event	`public interface ActionListener { public void actionPerformed(ActionEvent e); }`
DocumentEvent - generated when changes have been made to a text document such as insertion, removal in an editor	`public interface DocumentListener { public void changedUpdate(DocumentEvent e); public void insertUpdate(DocumentEvent e); public void removeUpdate(DocumentEvent e); }`
ListSelectionEvent - generated when selecting (i.e., click or double click) a list item	`public interface ListSelectionListener { public void valueChanged(ListSelectionEvent e); }`
WindowEvent - generated when open/close, activate/deactivate, iconify/deiconify a window	`public interface WindowListener { public void windowOpened(WindowEvent e); public void windowClosed(WindowEvent e); public void windowClosing(WindowEvent e); public void windowActivated(WindowEvent e); public void windowDeactivated(WindowEvent e); public void windowIconified(WindowEvent e); public void windowDeiconified(WindowEvent e); }`
KeyEvent - generated when pressing and/or releasing a key while within a component	`public interface KeyListener { public void keyPressed(KeyEvent e); public void keyReleased(KeyEvent e); public void keyTyped(KeyEvent e); }`
MouseEvent - generated when pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component	`public interface MouseListener { public void mouseClicked(MouseEvent e); public void mouseEntered(MouseEvent e); public void mouseExited(MouseEvent e); public void mousePressed(MouseEvent e); public void mouseReleased(MouseEvent e) ; }`
MouseEvent - generated when moving mouse within a component while button is up or down	`public interface MouseMotionListener { public void mouseDragged(MouseEvent e); public void mouseMoved(MouseEvent e); }`

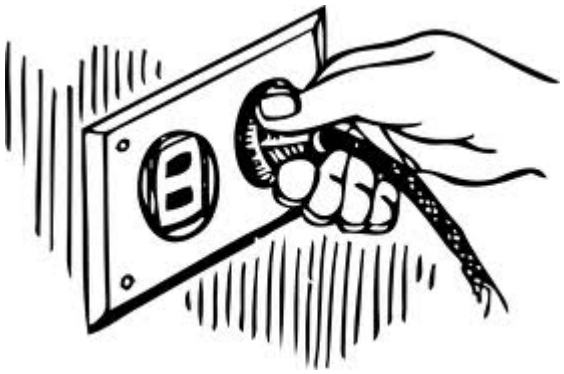
So what does all of this mean ? It means, for example, that if you want to handle a button press in your program, you need to write an **actionPerformed()** method:

```
public void actionPerformed(ActionEvent e) {
    //Do what needs to be done when the button is clicked
}
```

If you want to have something happen when the user presses a particular key on the keyboard, you need to write a **keyPressed()** method:

```
public void keyPressed(KeyEvent e) {
    //Do what needs to be done when a key is pressed
}
```

Once we decide which events we want to handle and then write our event handlers, we then need to ***register*** the event handler. This is like "**plugging-in**" the event handler to our window. In general, many applications can *listen for* events on the same component. So when the component event is generated, JAVA must inform everyone who is listening. We must therefore tell the component that we are listening for (or waiting for) an event. If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler). So, when a component wants to signal/fire an event, it sends a specific message to all listener objects that have been registered (i.e., anybody who is "listening"). For every event, therefore, that we want to handle, we must not only write the listener (i.e., event handler) but also **register** that listener.



To help you understand this notion of registering, imagine signing up on a webpage somewhere to receive an email notification when some event occurs (e.g., when something goes on sale, or getting an email bill-statement at the end of the month). When we sign up, we are essentially registering for (or listening to) any updates that may occur as a result of the event.

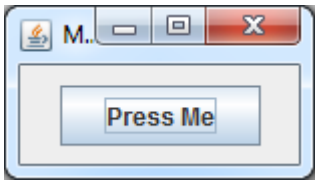
To **register** for an event (i.e., enable it), we need to merely add the listener (i.e., your event handler) to the component by using an **addXXXListener()** method (where XXX depends on the type of event to be handled). Here are some examples:

```
aButton.addActionListener(anActionListener);
aJPanel.addMouseListener(aMouseListener);
aJFrame.addWindowListener(aWindowListener);
```

Here **anActionListener**, **aMouseListener** and **aWindowListener** can be instances of **any class that implements the specific Listener interface**.

So, for example, if you wanted to have your application handle a button press, you can make your application itself be the **ActionListener** as follows:

```
import java.awt.event.*; // Need this for ActionEvent and ActionListener
import javax.swing.*;    // Need this for JFrame and JButton

public class SimpleEventTest extends JFrame implements ActionListener {
    public SimpleEventTest(String name) {
        super(name);

        getContentPane().setLayout(null);

        JButton aButton = new JButton("Press Me");
        aButton.setLocation(20,10);
        aButton.setSize(100, 30);
        getContentPane().add(aButton);

        // Plugin button event handler using THIS class as the listener
        // (i.e., tell JAVA to call the actionPerformed() in THIS class)
        aButton.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250, 90);
    }
    // Must write this method now since SimpleEventTest
    // implements the ActionListener interface
    public void actionPerformed(ActionEvent e) {
        System.out.println("I have been pressed");
    }

    public static void main(String[] args) {
        JFrame frame = new SimpleEventTest("Making a Listener");
        frame.setVisible(true);
    }
}
```



Sometimes it is necessary to remove a listener (i.e., disable it). As we will see later, we often disable a listener while we are making changes to update the user interface so that additional events do not get generated automatically.

You can "unregister" from an event (i.e., disable the listener), by merely removing it using a **removeXXXListener()** method. Here are some examples:

```
aButton.removeActionListener(anActionListener);
aJPanel.removeMouseListener(aMouseListener);
aJFrame.removeWindowListener(aWindowListener);
```

It is important that you remove the listener that was previously added. Therefore, it is often necessary to store the listener in a variable so that it can be removed later:

```
ActionListener    buttonHandler = this;
...
aButton.addActionListener(buttonHandler);
...
aButton.removeActionListener(buttonHandler);
...
```

In some situations, we only want to handle one or two types of user interactions and therefore want to only write the event handlers needed to deal with that type of interaction. For example we may want to handle a **windowOpened()** event handler to do something when the window is first opened, but we may not care about when the window is closed, activated, iconified, etc..

Unfortunately, if we write code in the style as we did above by implementing the **WindowListener** interface, we would be forced to write 7 event handlers (since JAVA forces you to write ALL the methods defined in an interface that we implement):

```
public class WindowEventTest extends JFrame implements WindowListener {
    public WindowEventTest (String name) {
        ...
        // Plugin window event handler
        this.addWindowListener(this);
        ...
    }

    // Unfortunately, we now have to write 7 methods as follows,
    // (even though we really only want to write one):

    public void windowOpened(WindowEvent e) {
        System.out.println("Window has been opened");
    }
    public void windowClosed(WindowEvent e) { /* leave blank */ }
    public void windowClosing(WindowEvent e) { /* leave blank */ }
    public void windowActivated(WindowEvent e) { /* leave blank */ }
    public void windowDeactivated(WindowEvent e) { /* leave blank */ }
    public void windowIconified(WindowEvent e) { /* leave blank */ }
    public void windowDeiconified(WindowEvent e) { /* leave blank */ }

    public static void main(String[] args) { ... }
}
```

Writing these extra "empty" event handling methods is a lot of extra code writing that just wastes time and makes the code more confusing. It does seem a little silly to have to write 6 blank methods when we do not even want to handle these other kinds of events. The JAVA guys recognized this inconvenience and solved it using the notion of ***Adapter*** classes.

> *An **adapter class** is a class that is used to implement an interface having a set of dummy methods.*

For each listener interface that has more than one method specified, there exists an adapter class with a corresponding name:

- **MouseListener** has **MouseAdapter**
- **MouseMotionListener** has **MouseMotionAdapter**
- **DocumentListener** has **DocumentAdapter**
- **WindowListener** has **WindowAdapter**
- ...and so on.

ActionListener and **ListSelectionListener** do NOT have an

adapter class since they only define one method each.

Here, for example is the **WindowAdapter** class:

```
public abstract class WindowAdapter implements WindowListener {
   public void windowOpened(WindowEvent e) {};
   public void windowClosed(WindowEvent e) {};
   public void windowClosing(WindowEvent e) {};
   public void windowActivated(WindowEvent e) {};
   public void windowDeactivated(WindowEvent e) {};
   public void windowIconified(WindowEvent e) {};
   public void windowDeiconified(WindowEvent e) {};
}
```

Adapter classes are provided for convenience sake to help us avoid writing empty methods. We can write subclasses that **extend** adapter classes, thereby inheriting the blank methods. Therefore, we would only need to write the event handlers that we are interested in, allowing the dummy methods to be inherited.

Unfortunately, looking at our previous **WindowEventTest** program, we cannot simply extend **WindowAdapter** since the code extends **JFrame** already and JAVA only allows us to extend a single class.

As a solution, we could create a new ***inner* class** just for the event handler and then use that:

```
import java.awt.event.*;   // Need this for WindowEvent and WindowListener
import javax.swing.*;      // Need this for JFrame

public class WindowEventTest2 extends JFrame {
    public WindowEventTest2 (String name) {
        super(name);

        // Plugin window event handler by creating a separate class
        // that extends WindowAdapter so that only one method needs
        // to be written.   This is called an "inner class", which
        // must have default access (e.g., not public nor private).
        class MyWindowHandler extends WindowAdapter {
            public void windowOpened(WindowEvent event) {
                System.out.println("Window has been opened");
            }
        }

        this.addWindowListener(new MyWindowHandler());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 300);
    }
    public static void main(String[] args) {
        JFrame frame = new WindowEventTest2("Inner Class Example");
        frame.setVisible(true);
    }
}
```

This is the first time that we created a class within another class explicitly.

*An **inner class** is a class declared entirely within the body of another class or interface.*

Interestingly, when you create an inner class, the JAVA compiler will create an additional class file (in this case it is called **WindowEventTest2$1MyWindowHandler.class**). As you write more user interfaces, you will find many such class files created... each identified by a **$** character in the name.

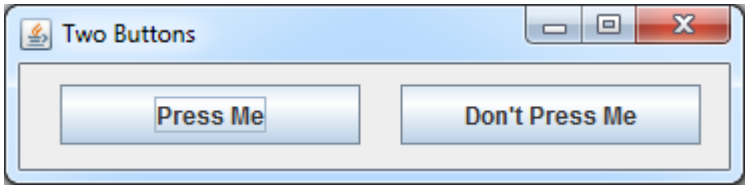
To reduce clutter in your program, JAVA allows another shorter syntax for creating inner classes. It is a way of defining a class without specifying a name for the class. Here is the syntax:

```
import java.awt.event.*;   // Need this for WindowEvent and WindowListener
import javax.swing.*;      // Need this for JFrame

public class WindowEventTest3 extends JFrame {
    public WindowEventTest3 (String name) {
        super(name);

        // Plugin window event handler by creating an anonymous class
        // that extends WindowAdapter.
        this.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent event) {
                System.out.println("Window has been opened");
            }
        });
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250, 90);
    }
    public static void main(String[] args) {
        JFrame frame = new WindowEventTest3("Anonymous Class Example");
        frame.setVisible(true);
    }
}
```

This syntax actually creates and makes an instance of an inner class as a subclass of **WindowAdapter**. The class has no name, it is considered to be an ***anonymous*** class. This code actually creates an instance of the anonymous class and returns it to us. It is weird syntax. The .class file produced from this anonymous class will be titled: **WindowEventTest3$1.class**

This idea of using anonymous classes is the simplest and most compact code for adding event handlers when only one method from a listener interface is needed. We will use adapter classes often in our examples.

Example:



How could we write a program that was able to distinguish between two different buttons on the window ?

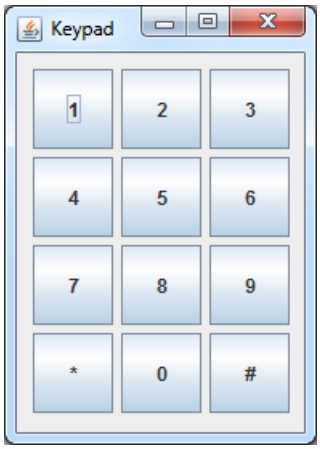
Adding the buttons to the window is easy. To connect their event handlers, we have two choices. We can either:

1) write separate event handlers for each button, or
2) write a single event handler for both buttons.

In this example, we will choose the first option. Here is the code with separate event handlers:

```
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;       // Needed for JFrame and JButton

public class TwoButtonsApp extends JFrame {
      public TwoButtonsApp(String title) {
            super(title);
            getContentPane().setLayout(null);

            JButton button1 = new JButton("Press Me");
            button1.setLocation(20,10);  button1.setSize(150, 30);
            getContentPane().add(button1);

            JButton button2 = new JButton("Don't Press Me");
            button2.setLocation(190,10);  button2.setSize(150, 30);
            getContentPane().add(button2);

            // Add the first button's event handler
            button1.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        System.out.println("That felt good!");
                  }});

            // Add the second button's event handler
            button2.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        System.out.println("Ouch! Stop that!");
                  }});

            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setSize(370,90);
      }

      public static void main(String args[]) {
            TwoButtonsApp frame =  new TwoButtonsApp("Two Buttons");
            frame.setVisible(true);
      }
}
```

Example:



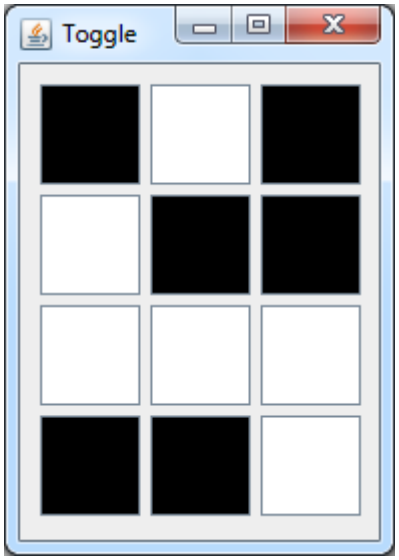
The previous example showed a good solution if we only have a small number of buttons. However, if we have more buttons, it is often good to have them share the same event handler. One way to do this is to have the **JFrame** implement **ActionListener** and then point all the buttons to it. We will create the following keypad using one event handler →

To do this, we will have the main application implement **ActionListener** and point all buttons to that same listener. The event handler will be written as a separate method (i.e., outside of the constructor), so we will need to store the buttons as instance variables (i.e., object attributes) so that we can access the buttons from both the constructor AND the event handler:

```
import java.awt.event.*;      // Needed for ActionListener and ActionEvent
import javax.swing.*;         // Needed for JFrame and JButton

public class MultipleButtonsApp extends JFrame implements ActionListener {
   // This stores all buttons
   JButton[][] buttons;

   public MultipleButtonsApp(String title) {
      super(title);
      getContentPane().setLayout(null);

      buttons = new JButton[4][3];
      String[] buttonLabels = {"1","2","3","4","5","6","7","8","9","*","0","#"};
      for(int row=0; row<4; row++) {
         for (int col=0; col<3; col++) {
            buttons[row][col] = new JButton(buttonLabels[row*3+col]);
            buttons[row][col].setLocation(10+col*55, 10+row*55);
            buttons[row][col].setSize(50,50);
            buttons[row][col].addActionListener(this);
            getContentPane().add(buttons[row][col]);
         }
      }

      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(195,275);
   }

   // This is the single event handler for all the buttons
   public void actionPerformed(ActionEvent e) {
      System.out.println("Button " + e.getActionCommand() + " was pressed." );
   }

   public static void main(String args[]) {
      MultipleButtonsApp frame =  new MultipleButtonsApp("Keypad");
      frame.setVisible(true);
      }
}
```

You may notice that we have called the **getActionCommand()** method for the **ActionEvent** that is passed in as a parameter in the event handler. This method retrieves the text from the button that was pressed.

Example:



Another way of determining the button that was pressed is to use the **getSource()** method for the **ActionEvent**. This will return the component that generated the event (i.e., the actual **JButton** object representing the button that was pressed). This is particularly useful if we have a grid of buttons that have no labels on them. Consider this application where there are 12 buttons arranged again in a grid as shown here, such that all go to the same event handler →

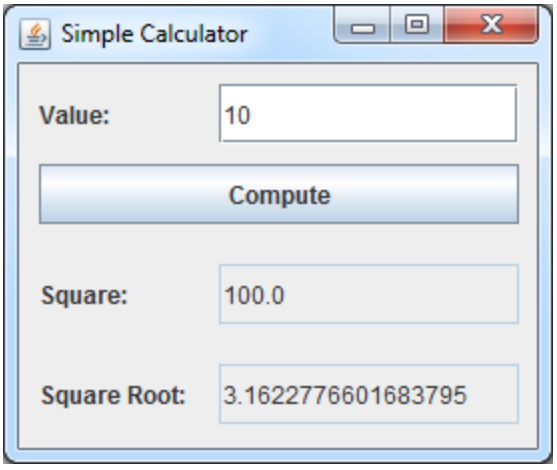
Now when a button is pressed, we cannot look at the text on the button since there is no text ... just different background colors. We will use the **getSource()** method and compare the pressed button with each button in the array. Once we find the button that matches, we will know its row and column and will be able to toggle the background of that button accordingly. Here is the code:

```
import java.awt.*;          // Needed for Color
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;       // Needed for JFrame and JButton

public class ToggleButtonsApp extends JFrame implements ActionListener {
   JButton[][]   buttons;   // This stores all buttons

   public ToggleButtonsApp (String title) {
      super(title);
      getContentPane().setLayout(null);

      buttons = new JButton[4][3];
      for(int row=0; row<4; row++) {
         for (int col=0; col<3; col++) {
            buttons[row][col] = new JButton();
            buttons[row][col].setLocation(10+col*55, 10+row*55);
            buttons[row][col].setSize(50,50);
            buttons[row][col].addActionListener(this);

            // Pick a random color for the button
            if (Math.random() < 0.5)
               buttons[row][col].setBackground(Color.black);
            else
               buttons[row][col].setBackground(Color.white);
            getContentPane().add(buttons[row][col]);
         }
      }
```

```
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(195,275);
   }

   // This is the single event handler for all the buttons
   public void actionPerformed(ActionEvent e) {
      // Find the row and column of the pressed button
      for(int row=0; row<4; row++) {
         for (int col=0; col<3; col++) {
            if (e.getSource() == buttons[row][col]) {
               System.out.println("You pressed the button at row: " +
                                  row + ", column: " + col + ".");
               // Now toggle the button's color
               if (buttons[row][col].getBackground() == Color.black)
                  buttons[row][col].setBackground(Color.white);
               else
                  buttons[row][col].setBackground(Color.black);
            }
         }
      }
   }

   public static void main(String args[]) {
      ToggleButtonsApp frame = new ToggleButtonsApp("Toggle");
      frame.setVisible(true);
   }
}
```

Example:

Now let us consider an example of an application that makes use of **JTextFields**. We will create a simple application that allows the user to type in a number into one text field, then press a button and have the "square" of that number appear in another text field and the "square root" of the number appear in yet another text field as shown here →



In this example, we will only be handling one event ... that of the user pressing the **Compute** button. We will need to extract the text data from the **Value** field. The **getText()** method in the **JTextField** class allows us to get the text (as a **String** object) that lies in the field. We will need to convert this **String** into a number, such as a **float**, in order to perform computations with it.

In JAVA, we can extract a **float** from a **String** by using the following strategy:

```
float x = Float.parseFloat(aString);
```

This code will convert the **String** (called **aString** in the example) into a **float** (called **x** in the example). There are actually similar ways to convert to other types. Here are some more:

```
int     x = Integer.parseInt(aString);
double  x = Double.parseDouble(aString);
boolean x = Boolean.parseBoolean(aString);   // "true" to true
```

Once we have the value as a number, we can compute the square and root and then we simply need to put the results into the other two text fields. We do that using **setText()** where we supply a **String** with the result in it. The simplest way to convert a number into a String is to append the number to an empty String object in JAVA as follows:

```
aTextField.setText("" + x);
```

This will work for any number **x**, regardless of whether it is an **int**, **float**, **double** , etc..

One last point ... we will probably want to disable editing in the last two text fields so that the user cannot type into them, since they are "output only" fields. We use the **setEditable(false)** method to do this. Here is the completed code:

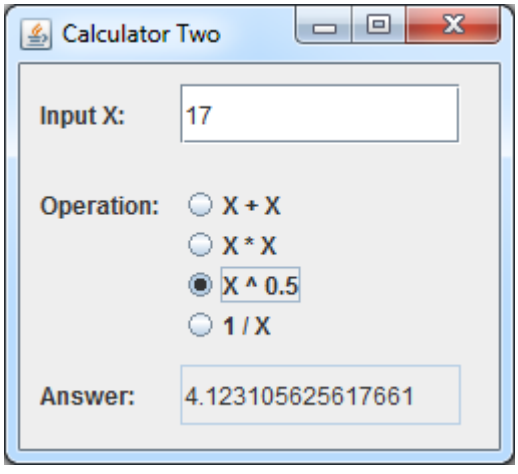
```
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;       // Needed for JFrame, JButton and JTextField

public class CalculatorApp extends JFrame {
   // Text fields to hold the user data and the computed data
   JTextField    valueField, squareField, rootField;

   public CalculatorApp(String title) {
      super(title);
      getContentPane().setLayout(null);

      // Add the value label and text field
      JLabel  label = new JLabel("Value:");
      label.setLocation(10,10);   label.setSize(100, 30);
      getContentPane().add(label);

      valueField = new JTextField();
      valueField.setLocation(100,10);   valueField.setSize(150, 30);
      getContentPane().add(valueField);

      // Add the compute button
      JButton computeButton = new JButton("Compute");
      computeButton.setLocation(10,50);   computeButton.setSize(240, 30);
      getContentPane().add(computeButton);

      // Add the square label and text field
      label = new JLabel("Square:");
      label.setLocation(10,100);   label.setSize(100, 30);
      getContentPane().add(label);

      squareField = new JTextField();
      squareField.setLocation(100,100);   squareField.setSize(150, 30);
      squareField.setEditable(false);
```

```
      getContentPane().add(squareField);

      // Add the square root label and text field
      label = new JLabel("Square Root:");
      label.setLocation(10,150);   label.setSize(100, 30);
      getContentPane().add(label);

      rootField = new JTextField();
      rootField.setLocation(100,150);   rootField.setSize(150, 30);
      rootField.setEditable(false);
      getContentPane().add(rootField);

      // Handle the button click
      computeButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            if (valueField.getText().length() > 0) {
               float value = Float.parseFloat(valueField.getText());
               squareField.setText("" + value * value);
               rootField.setText("" + Math.sqrt(value));
            }
         }});

      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(275,230);
   }
   public static void main(String args[]) {
      CalculatorApp frame = new CalculatorApp("Simple Calculator");
      frame.setVisible(true);
   }
}
```

You may have noticed that we did an error-check for the case where no text was in the text field. That is because the **parseFloat()** method generates an ugly error message if the **String** passed in is empty.

Example:

Here is another example of a calculator that can do more operations. It has been set up using a set of **JRadioButtons** which work exactly as **JButtons** do, except that we will add them to a **ButtonGroup** so that only one of the buttons is able to be selected at a time... just like an old-fashioned radio.



To make this work, we will add similar text fields as we did in the previous example. We will also create an array of **JRadioButtons** and have all of them go to the same event handler. Once again, we will search

through the button array to find out which button was pressed and then perform a computation accordingly. In JAVA, a **ButtonGroup** object is used to group buttons together so that only one is on in the group at a time. We simply add **JRadioButtons** to the same button group to get this desired behavior. Here is the code:

```
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;       // Needed for JFrame, JRadioButton and JTextField

public class CalculatorTwoApp extends JFrame implements ActionListener {
   JTextField         valueField, answerField;
   JRadioButton[]     buttons;

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

      getContentPane().setLayout(null);

      // Add the value label and text field
      JLabel  label = new JLabel("Input X:");
      label.setLocation(10,10);   label.setSize(100, 30);
      getContentPane().add(label);

      valueField = new JTextField();
      valueField.setLocation(80,10);   valueField.setSize(140, 30);
      getContentPane().add(valueField);

      // Add the "operation type" radio buttons to the window
      // and to a ButtonGroup so that one is on at a time
      label = new JLabel("Operation:");
      label.setLocation(10,55);   label.setSize(100, 30);
      getContentPane().add(label);

      ButtonGroup operations = new ButtonGroup();
      buttons = new JRadioButton[4];
      String[] buttonLabels = {"X + X", "X * X", "X ^ 0.5", "1 / X"};
      for (int i=0; i<4; i++) {
         buttons[i] = new JRadioButton(buttonLabels[i], false);
         buttons[i].setLocation(80, 60 + i*20);
         buttons[i].setSize(150, 20);
         getContentPane().add(buttons[i]);
         operations.add(buttons[i]);
         buttons[i].addActionListener(this);
      }

      // Add the answer label and text field
      label = new JLabel("Answer:");
      label.setLocation(10,150);   label.setSize(100, 30);
      getContentPane().add(label);

      answerField = new JTextField();
      answerField.setLocation(80,150);
      answerField.setSize(140, 30);
      answerField.setEditable(false);
      getContentPane().add(answerField);
```

```
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(255,230);
   }

   // Handle a radio button click
   public void actionPerformed(ActionEvent e) {
      int value = Integer.parseInt(valueField.getText());

      // Find the number of the button that was clicked
      int buttonNumber = 0;
      for (buttonNumber=0; buttonNumber<4; buttonNumber++) {
         if (buttons[buttonNumber] == e.getSource())
            break;
      }

      // Perform the calculation
      double result=0;
      switch (buttonNumber) {
         case 0: result = value + value; break;
         case 1: result = value * value; break;
         case 2: result = Math.sqrt(value); break;
         case 3: result = 1 / (double)value; break;
      }

      // Show the answer
      answerField.setText("" + result);
   }

   public static void main(String args[]) {
      CalculatorTwoApp frame = new CalculatorTwoApp("Calculator Two");
      frame.setVisible(true);
   }
}
```

Note as well that the **JCheckBox** works similar to the **JRadioButton**, except that normally **JRadioButtons** should have only one on at a time, while **JCheckBoxes** may normally have many on at a time. Here is how the window would look if **JCheckBoxes** were used instead (although keep in mind that in this application, it doesn't make sense to have more than one button on at a time). For **JCheckBoxes**, you should NOT add them to a **ButtonGroup**.