COMP1406/1006

Design and Implementation of Computer Applications

Course Notes

Notes maintained by Mark Lanthier (2006 version)

COMP1406 - Course Contents

1 User Interfaces

What's in This Set of Notes ?

The most dominant part of this course is related to designing and implementing complete applications that have user interfaces. In our previous course, our user interface was either keyboard input or test programs. In this set of notes we will look at some user interface terminology and learn how to keep our user interface separate from our model classes as well as examine a simple text-based user interface.

Here are the individual topics found in this set of notes (click on one to go there):

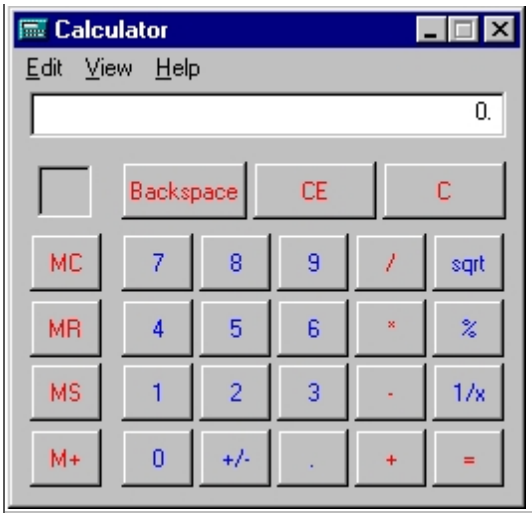
1.1 User Interface Terminology

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. Sometimes the interface has physical/hardware interactive components (e.g., a bank machine or bank teller) and sometimes the components are software-based (e.g., menus, buttons, text fields).

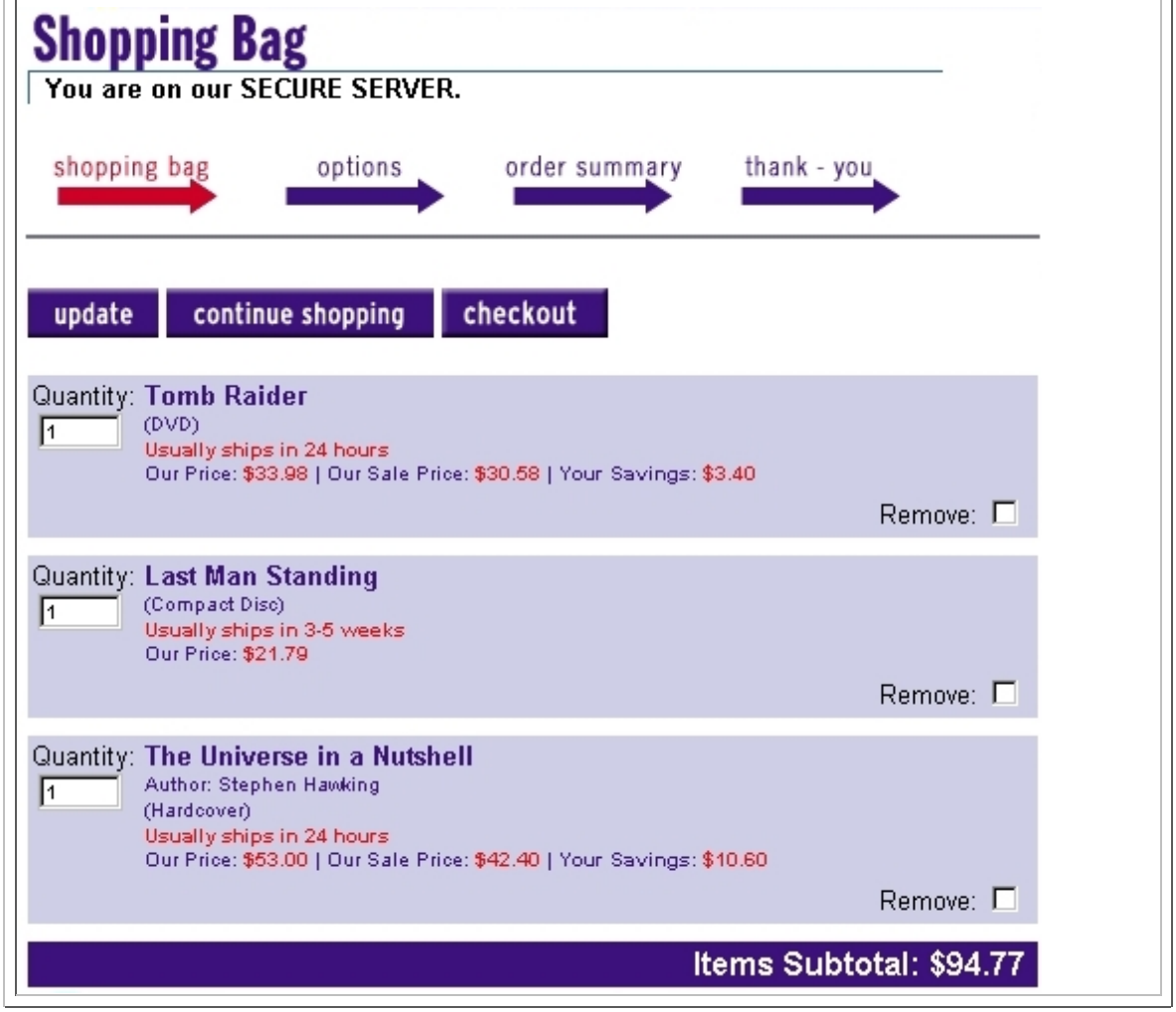
a **Bank** has BankTellers as well as BankMachines, calculators have physical components:



a software application has buttons/text fields/menus:



an **On-line Shopping Cart** has webpages containing forms, text fields, lists, buttons:



In this course, we will consider software-based user interfaces (such as the software calculator shown above).

Some programs/applications do not really have any interactive user interface. For example, our **Team/League** example from COMP1405/1005 had a **main** method as a "pretend" user interface. We simply ran the code and it spewed output onto the console. We did not get to interact much at all with it, except to say "Go!" by running it.

What about simple text-based interfaces ? Recall the style of the basic text-based user interfaces that we have seen in COMP1405/1005:

```
public class AverageTest {
    public static void main(String args[]) {
        java.util.Scanner  keyboard = new java.util.Scanner(System.in);
        int sum = 0.0;
        for (int i=0; i<5; i++) {
            System.out.println("Enter the number " + i + ":");
            sum += keyboard.nextInt();
        }
        System.out.print("The average is " + sum / 5.0);
    }
}
```

Here we received input from the keyboard, did a calculation, and then printed out the result. Later we wrote code that did not require any user input from the keyboard, but instead used simple test methods to simulate some "fixed scenario":

```
public static void main(String args[]) {
    League aLeague = new League("NHL");

    aLeague.addTeam(new Team("Ottawa Senators"));
    aLeague.addTeam(new Team("Montreal Canadians"));
    aLeague.addTeam(new Team("Toronto Maple Leafs"));
    aLeague.addTeam(new Team("Vancouver Cannucks"));

    aLeague.recordWinAndLoss("Ottawa Senators", "Toronto Maple Leafs");
    aLeague.recordWinAndLoss("Montreal Canadians", "Toronto Maple
Leafs");
    aLeague.recordWinAndLoss("Ottawa Senators", "Vancouver Cannucks");
    aLeague.recordTie("Vancouver Cannucks", "Toronto Maple Leafs");
    aLeague.recordWinAndLoss("Toronto Maple Leafs", "Montreal
Canadians");

    System.out.println("\nHere are the teams:");
    aLeague.showTeams();
}
```

Nevertheless, all of our programs so far have been ***Java Applications*** which:

- can be invoked from command line (e.g., **java MyApplication**), or with a button click when using **JCreator,**
- are always executed by the JAVA Interpreter (i.e., JVM), and
- must have **main()** method in order to run.

Something was fundamentally common between all our programs:

- they all had underlying classes representing objects that were specific to the application (e.g., the **Team** and **League** classes)
- the testing code (which was our "user interface") always made use of these underlying objects

As it turns out, these underlying objects altogether are called the ***Model*** of the application.

A ***Model***:

- may consist of one or more classes representing the *'business logic'* part of the application
- represents the underlying system on which a user interface is attached
- is developed separately from the user interface
- should not assume any knowledge about the user interface (e.g., may not assume that a **System.out** is available).

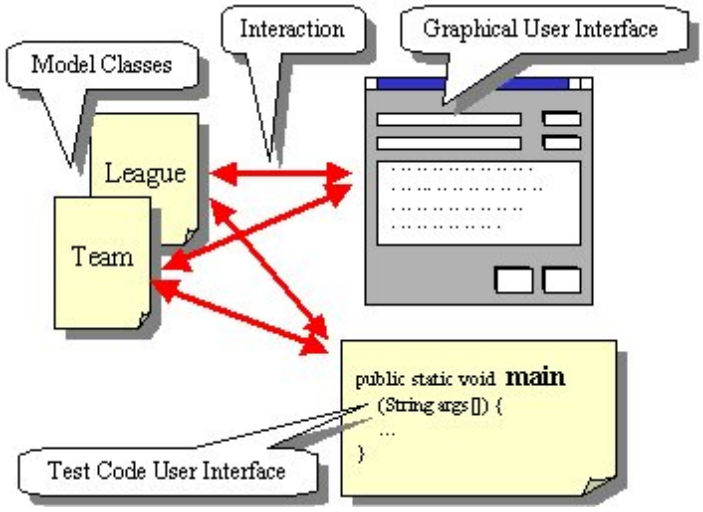
The ***User Interface***:

- is "attached to" a model, so we often develop the model first

- handles user interaction and does NOT deal with the business logic
- "uses" the model classes' methods
- often causes the model to change according to user interaction and these model changes are often reflected back on the user interface

A ***Graphical User Interface (GUI)***:

- is a user interface that has one or more windows (e.g., such as the calculator application shown above)
- is often preferred over text-based interfaces (more natural ... more like real-world)
    - imagine the internet as being only text-based.



We often split up the user interface as well into two separate pieces called the ***view*** and the ***controller***:

A ***View*** is:

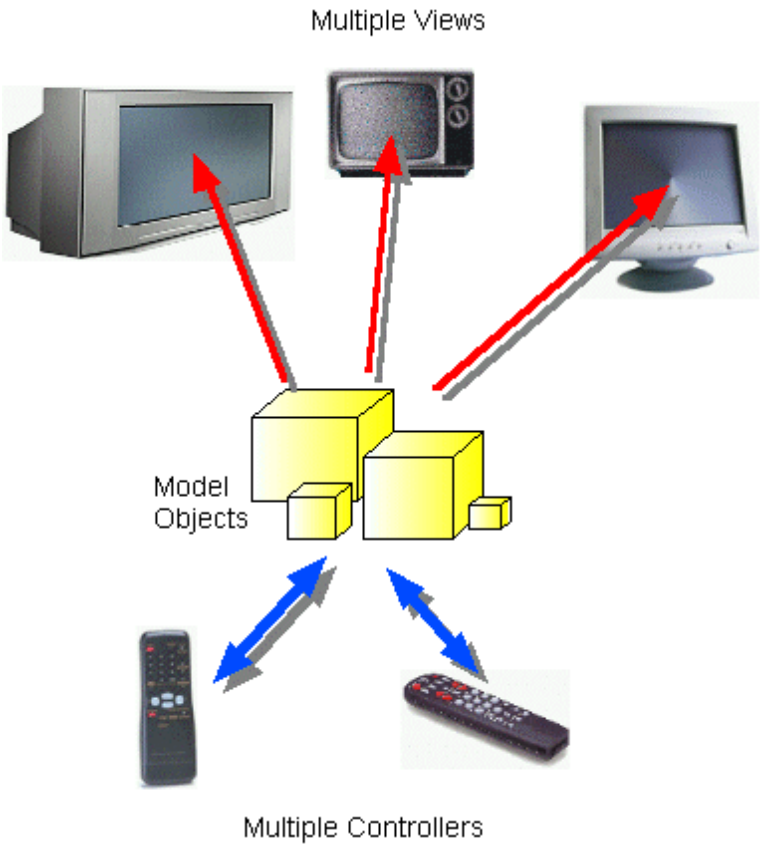
- the portion of the user interface code that specifies how the model is shown visually (i.e., its appearance).

A ***Controller*** is:

- the portion of the user interface code that specifies how the model interacts with the view (i.e., its behaviour).
- code that serves as a "mediator" between the model and the view.

It is ALWAYS a good idea to separate the model, view and controller (MVC):

- code is cleaner and easier to follow when the view and controller are separated
- we may want to have multiple views or controllers on the same model.
- we may want to have multiple models for the same user interface.



1.2 A Simple Text-Based User Interface

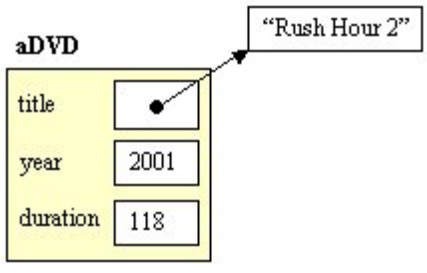
Let us now look at an example of how to separate our model from our user interface. We will create an application that allows us to maintain a small database to store the DVDs that we own. We must think of what we want to be able to do with the application. These are the ***requirements***:

- We want to keep the DVD's title, year and duration (i.e., how long the DVD lasts) when played (e.g., 120 minutes).
- We should be able to **add** new DVDs to our collection as well as **remove** them (if we lose or sell them).
- We should be able to **list** our DVDs sorted by title.

So, that's it. It will be a simple application. Let us begin by creating the model, since we did this already a few times in COMP1405/1005.
What objects do we need to define ?

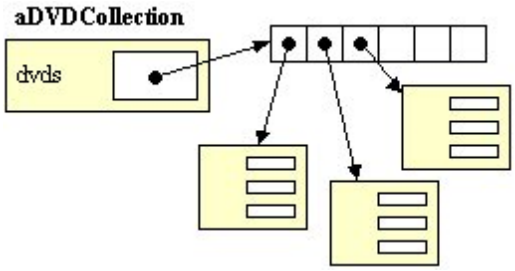
- a DVD object (maintains title, year, duration)
- an object to represent the collection of DVDs (similar to our **League** class in COMP1405/1005).

Here is a simple DVD class:



```
public class DVD {
    private   String title;
    private   int year;
    private   int duration;

    public DVD () { this("", 0, 0); }
    public DVD (String newTitle, int y, int minutes) {
        title = newTitle;
        year = y;
        duration = minutes;
    }

    public String getTitle() { return title; }
    public int getDuration() { return duration; }
    public int getYear() { return year; }
    public void setTitle(String t) { title = t; }
    public void setDuration(int d) { duration = d; }
    public void setYear(int y) { year = y; }

    public String toString() {
        return ("DVD (" + year + "): " + title + " with length: " +
duration);
    }
}
```

There is not much to this class. Now what about the DVD collection itself ? It should probably look something like this in its most basic form:




```
import java.util.*;
public class DVDCollection {
    private ArrayList<DVD>   dvds;

    public DVDCollection() { dvds = new ArrayList<DVD>(); }

    public ArrayList<DVD> getDvds() { return dvds; }
    public String toString() { return ("DVD Collection of size " +
dvds.size()); }

    public void add(DVD aDvd) { dvds.add(aDvd); }
    public boolean remove(String title) {
        for (DVD aDVD: dvds) {
            if (aDVD.getTitle().equals(title)) {
                dvds.remove(aDVD);
                return true;
            }
        }
        return false;
    }
}
```

Notice when adding a DVD to the collection, we simply add it to the ArrayList. When removing, it is more convenient to have a method that removes according to title. The **remove()** method therefore takes a string and searches for the DVD with that title.

OK. So now we can test the adding/removing and listing with a **main()** method as follows:

```
public static void main (String[] args) {
    DVDCollection c = new DVDCollection();
    c.add(new DVD("Star Wars", 1978, 124));
    c.add(new DVD("Java is cool", 2002, 93));
    c.add(new DVD("Mary Poppins", 1968, 126));
    c.add(new DVD("The Green Mile", 1999, 148));
    c.remove("Mary Poppins");
    // List the DVDs
    for (DVD aDVD: c.getDvds())
        System.out.println(aDVD);
}
```

Let us now make a user interface for this model. We should make a new class for this. We will make a simple text-based interface. It should perhaps bring up a menu and repeatedly prompt for a user choice. Here is what we will make appear on the screen:

```
Welcome to the Dvd Collection User Interface
--------------------------------------------
1.  Add DVD
2.  Delete DVD
3.  List DVDs
4.  Exit

Please make a selection:
```

If the user makes an invalid selection, we print an error message out. This is considered to be our main menu. Once we make a selection and the action has been performed, this menu will be displayed again. Here is the user interface process:

1. Display a menu with choices
2. Wait for the user to make a selection
3. Perform what needs to be done from that selection
    1. possibly more input is required
    2. the user may want to quit the program
    3. computations may be made, output may be displayed
4. Go back up to step 1.

We will design our code so that the main user interface holds onto a DVD collection as its model:

```
public class DVDUI {
    private   DVDCollection   model;

    public DVDUI() { model = new DVDCollection(); }
    public DVDCollection getModel() { return model; }

    public void showMainMenu() {
        System.out.println("1.  Add DVD");
        System.out.println("2.  Delete DVD");
        System.out.println("3.  List DVDs");
        System.out.println("4.  Exit");
        System.out.println("\nPlease make a selection");
    }

    public static void main (String[] args) {
        System.out.println("Welcome to the Dvd Collection User
Interface");

System.out.println("-----------------------------------------");

        new DVDUI().showMainMenu();
    }
```

```
}
```

Notice that the instance variable is named **model** which holds onto the **DVDCollection** which the interface is attached to. We could have picked any name for this variable, but we chose **model** here to help you understand that the rest of the code represents the user interface.

The model is now "plugged-into" the user interface through this instance variable. However, we will want to make use of the model's methods. Let us now handle the keyboard input and make the main menu repeat until 4 is entered:

```
public void showMainMenu() {
    while(true) {
        System.out.println("1.  Add DVD");
        System.out.println("2.  Delete DVD");
        System.out.println("3.  List DVDs");
        System.out.println("4.  Exit");
        System.out.println("\nPlease make a selection");
        int   selection = new java.util.Scanner(System.in).nextInt();
        switch(selection) {
            case 1:   /* Handle the adding of DVDs   */ break;
            case 2:   /* Handle the removing of DVDs */ break;
            case 3:   /* Handle the listing of DVDs  */ break;
            case 4:   System.exit(0);
            default:  System.out.println("Invalid Selection");
        }
    }
}
```

If we run our code now, we will notice that it repeatedly brings up the main menu, waits for a response from the user and then displays the menu again. This repeats until the user selects option 4 to quit the program.

So we now have:

- The **Model**: which is the DVD collection class and the DVD class
- The **View**: which is the menu system

We now need to create the **Controller** part of the program which is responsible for linking the model to the view through proper interaction. The controller decides what happens to the model when the user interacts with the program through the view. So we just need to handle the menu choices. We can create helper methods so that the main method stays simple. We can call these helper methods from the switch statement:

```
switch(selection) {
   case 1:   addDVD();    break;
   case 2:   deleteDVD(); break;
   case 3:   listDVDs();  break;
   case 4:   System.exit(0);
   default:  System.out.println("Invalid Selection");
}
```

We must now decide what needs to be done in each of the helper methods:

- The **addDVD()** method should:
  - make a new DVD and add it to the collection (i.e., to the model)
  - allow the user to enter the title, year and duration of the DVD

```
private void addDVD() {
     DVD    aDVD = new DVD();
     System.out.println("Enter DVD Title:  ");
     aDVD.setTitle(new java.util.Scanner(System.in).nextLine());
     System.out.println("Enter DVD Year (e.g., 2001):");
```

```
            aDVD.setYear(new java.util.Scanner(System.in).nextInt());
            System.out.println("Enter DVD Duration (minutes):");
            aDVD.setDuration(new java.util.Scanner(System.in).nextInt());
            model.add(aDVD);
        }
```

- The **deleteDVD()** method should:
  - prompt the user for the name of the DVD to remove
  - remove the DVD from the collection (i.e., from the model)

```
    private void deleteDVD() {
        System.out.println("Enter DVD Title:  ");
        model.remove(new java.util.Scanner(System.in).nextLine());
    }
```

- The **listDVDs()** method should:
  - list all the DVDs in the collection (i.e., model) to the console.

```
    private void listDVDs() {
        for (DVD aDVD: model.getDvds())
            System.out.println(aDVD);
    }
```

The above code for listing DVDs displays them in the order that they were added. What about having them displayed sorted by title ? To do this, we can make use of the sort method in the Collections class:

```
private void listDVDs() {
    java.util.Collections.sort(model.getDvds());
    for (DVD aDVD: model.getDvds())
        System.out.println(aDVD);
}
```

Of course, this implies that our DVDs implement the **Comparable** interface ... which means that they have to have a **compareTo()** method. We will need to make the following changes to our **DVD** class definition:

```
public class DVD implements Comparable {

    ...
    public int compareTo(Object obj) {
        if (obj instanceof DVD) {
            DVD aDVD = (DVD)obj;
            return title.compareTo(aDVD.title);
        }
       return 0;
    }
    ...
}
```

So sorting by title merely compares the string titles using the **String** class's **compareTo()** method.

Click here for the combined code of everything we have so far.

We can make finishing touches on the application by:

- ensuring that the output displayed on the screen is pleasant on the eyes (e.g., properly tabbed)
- ensuring that our interface handles all likely errors gracefully (e.g., deleting a DVD that doesn't exist)

The BEST person to try out a user interface is a small child !! They'll be sure to find problems that you never imagined could occur. For example, our interface currently crashes when we enter a

non-integer menu choice, dvd year or dvd duration. We can fix this by catching the **InputMismatchExceptions** that may occur. We can modify the while loop in our main menu method as follows:

```
while(true) {
    System.out.println("1.  Add DVD");
    System.out.println("2.  Delete DVD");
    System.out.println("3.  List DVDs");
    System.out.println("4.  Exit");
    System.out.println("\nPlease make a selection");
    try {
        int selection = new java.util.Scanner(System.in).nextInt();
        switch(selection) {
            case 1:   addDVD();    break;
            case 2:   deleteDVD(); break;
            case 3:   listDVDs();  break;
            case 4:   System.exit(0);
            default:  System.out.println("Invalid Selection");
        }
    }
    catch(java.util.InputMismatchException e) {
        System.out.println("Invalid Selection");
    }
    System.out.println("\n");
}
```

We will also want to change our **addDVD()** method similarily:

```
private void addDVD() {
    DVD    aDVD = new DVD();
    System.out.println("Enter DVD Title:  ");
    aDVD.setTitle(new java.util.Scanner(System.in).nextLine());
    int entered = -1;
    do {
        System.out.println("Enter DVD Year (e.g., 2001):");
        try {
            entered = new java.util.Scanner(System.in).nextInt();
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Invalid Year");
        }
    }
    while (entered == -1);
    aDVD.setYear(entered);

    entered = -1;
    do {
        System.out.println("Enter DVD Duration (minutes):");
        try {
            entered = new java.util.Scanner(System.in).nextInt();
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Invalid Duration");
        }
    }
    while (entered == -1);
    aDVD.setDuration(entered);
    model.add(aDVD);
}
```

We should also display a nice message to provide feedback to the user upon deletion:

```
private void deleteDVD() {
    System.out.println("Enter DVD Title:  ");
    String title = new java.util.Scanner(System.in).nextLine();
```

```
        boolean success = model.remove(title);
        if (success)
            System.out.println("\nDVD: " + title + " was deleted successfully
\n");
        else
            System.out.println("\n*** Error: Could not find DVD: " + title +
"\n");
    }
```

I am sure you can think of many ways to improve this code. Here are the important things to remember from this code:

- The model was created separately from the user interface
- The user interface was created with a particular model in mind (in this case a DVDCollection)
- The model was "tied to" the user interface through an instance variable in the user interface class
- The user interface accesses and modifies the model by calling **public** methods in the model classes

---

2 Applications and LayoutManagers

What's in This Set of Notes?

We look here at how to create a graphical user interface (i.e., with windows) in JAVA. Creating GUIs in JAVA requires adding components onto windows. We will find out how to do this as well as look at an interesting JAVA feature called a "LayoutManager" that automatically arranges components on the window. This allows us to create simple windows without having to worry about resizing issues.

Here are the individual topics found in this set of notes (click on one to go there):

2.1 Creating a Basic GUI Application

Recall that a ***Graphical User Interface*** (GUI) is a user interface that has one or more windows.

A ***frame:***

- is the JAVA terminology for a window (i.e., its a window frame)
- represented by the **Frame** and **JFrame** classes in JAVA
- used to show information and handle user interaction
- has no security restrictions ... can modify files, perform I/O, open network connections to other computers, etc..

The following code creates a basic window frame in JAVA and shows it on the screen:

```
javax.swing.JFrame frame = new javax.swing.JFrame("This appears at the
top of the window");
frame.setSize(300, 100);
frame.setVisible(true);
```

Here is what it looks like:

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.
2. Make this class extend **JFrame** (or **Frame** if you want to use the older AWT classes ... more on this later).
3. Create a constructor method that sets the window title (specified as a parameter) and any other settings as well such as background color.
4. Include a **main()** method as a starting point for the application which is similar to the above code.

So here is the template that you should use:

```
import javax.swing.*; //needed to use swing components e.g. JFrame
public class FirstApplication  extends JFrame {
    public FirstApplication(String title) {
        super(title);        // Set the title of the window
        setDefaultCloseOperation(EXIT_ON_CLOSE);  // allow window to
close
        setSize(300, 100);   // Set the size of the window
    }
    public static void main(String args[]) {
         // Instantiate a FirstApplication object so you can display it
         FirstApplication frame =  new
FirstApplication("FirstApplication Example");
         frame.setVisible(true); // Show the window now
    }
}
```

Note that we make use of the following instance method for **JFrame** objects to set the size of the frame:

```
setSize(int frameWidth, int frameHeight);
```

If we do not set the size, the window shows up so small that we only see part of the title bar.

When frames are created, they do not appear on the screen. They are essentially hidden. To make the window visible and give control to your application window, we use the **setVisible** method:

```
setVisible(boolean isVisible);
```

which either shows or hides the window according to the boolean value supplied.

Note 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 behaviour for all applications that run under windowing operating system environments.

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

What happens ? ... A new application window should come up with the title that we specified.
Try changing the size of the window.

What happens if we set visibility to **false** ? ... The application starts but nothing is displayed. Press <CNTRL>C to stop it.

What other choices do we have when the window is closed ?

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);
```

It is possible to alter the look and feel of our application's windows in JAVA. We can have our windows look like standard windows applications, like standard cross-platform JAVA applications or any look and feel that we would like. To do this, we merely need to add a few lines to our code to tell the user interface manager (UIManager) that we would like all of our windows to have a certain standard look and feel to them. Here is what we need to add BEFORE we make our frame:

```
    try {
        UIManager.setLookAndFeel(aLookAndFeel);
    } catch(Exception e) {}
        JFrame.setDefaultLookAndFeelDecorated(true);
    // ... now make our frame as usual
```

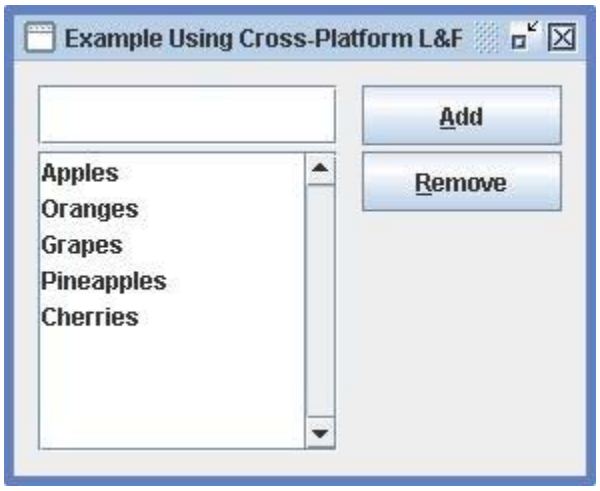
Here, **aLookAndFeel** can be any "LookAndFeel" available in Java. Here are just some examples:

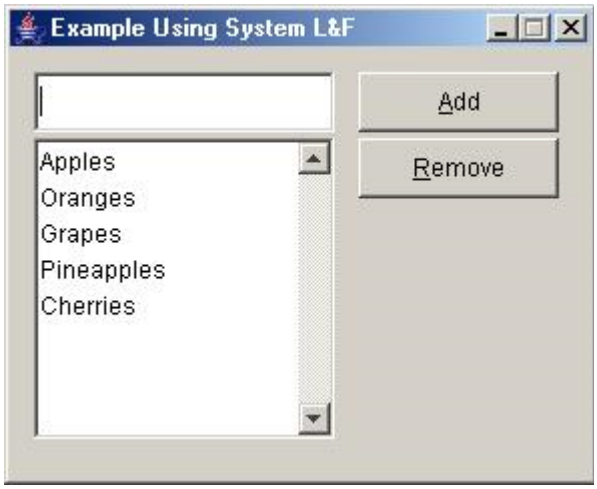
- `UIManager.getCrossPlatformLookAndFeelClassName()`
- `UIManager.getSystemLookAndFeelClassName()`
- `"com.sun.java.swing.plaf.motif.MotifLookAndFeel"`

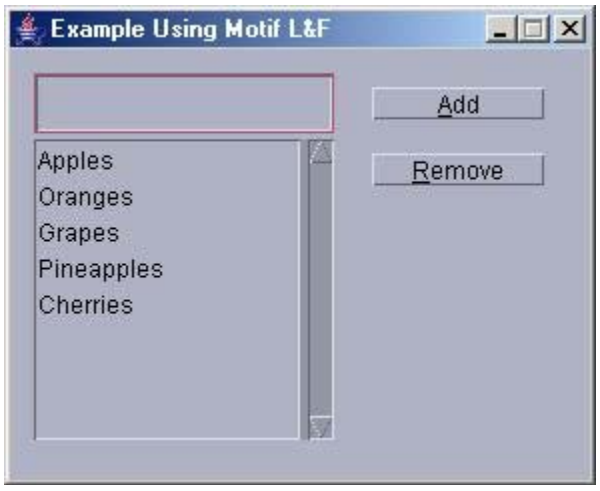
To illustrate, consider an example of a simple window (which we will show how to construct later). We can add the following lines of code to the main method in order to choose a particular look and feel:

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

    public NoLayoutExample(String name) {
        // Code for building the window ... we will see this later
    }

    public static void main(String[] args) {
        try {

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
        } catch(Exception e) {}
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new NoLayoutExample("Example Using Cross-Platform L&F");
        frame.setVisible(true);
    }
}
```

Here are some snapshots of the application using the cross-platform l&f, the System (i.e., windows) l&f and the motif l&f:







If you have some spare time on your hands, you can try experimenting with other looks and feels ... you can even create your own ;).

2.2 Components and Containers

Our windows that we make will need to have various components on them. ***Components*** are objects on our window that have a visual representations (e.g., labels, lists, scroll bars, buttons, text fields, menubars, menus). They may allow the user to interact with them (e.g., lists, scroll bars, buttons, text fields, menus).

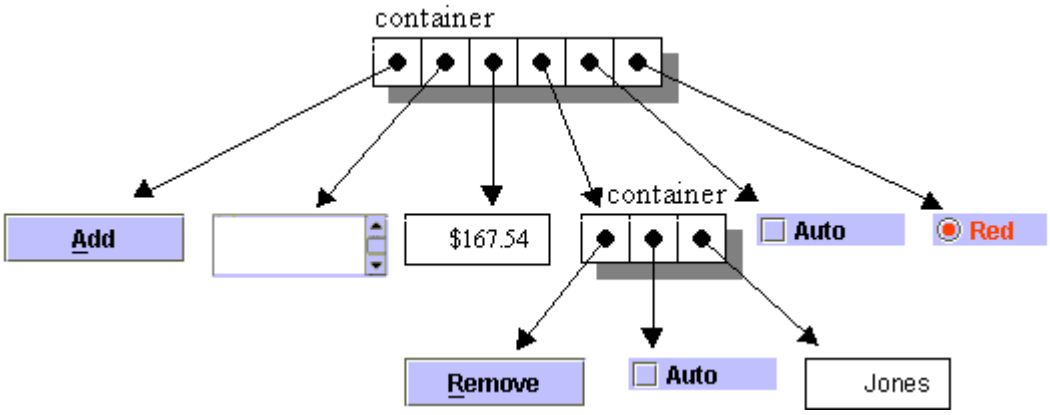
Components may also be grouped together, much like adding elements to an ArrayList. In fact, most

components in JAVA can contain other components as their sub-parts. This brings up the notion of a ***Container***.

Containers:

- can contain other components (e.g., a window, **JPanel**, or **JApplet**).
- are actually components as well (e.g., we can have containers which contain other containers)
- can have their components automatically laid out using a ***LayoutManager*** (more on this later)

So, a container of components is conceptually "like" an ArrayList of Objects:



There are two main sets of visual components and containers for user interface design in JAVA:

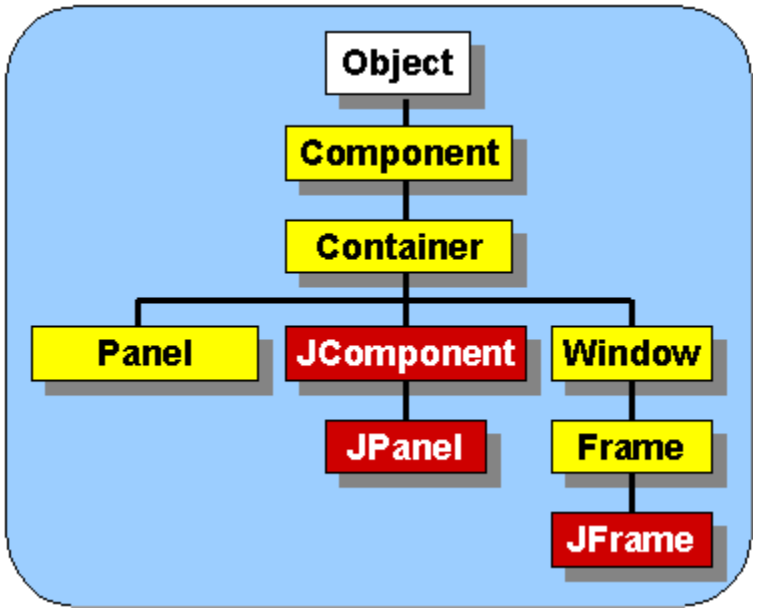
i) AWT (Abstract Window Toolkit)

- the original window components in JAVA
- components are contained in `java.awt` package
- subclasses of **java.awt.Container** can contain **java.awt.Components**
- has component names such as **Button** and **Label**

ii) Swing

- extends the AWT by adding a set of components, the **JComponents**, and a group of related support classes.
- since **Containers** are themselves **Components**, they can be nested arbitrarily deep. This allows for arrangements such as a frame containing two panels, each of which contains two labels etc..
- Swing takes containment one step further.
  - All **JComponents** are subclasses of **java.awt.Container**.
  - This allows Swing components, such as **JLabel**, to contain other components (either AWT or Swing).

Here is just a portion of the component hierarchy. The **red** classes are Swing classes, while the **yellow** ones are AWT classes:



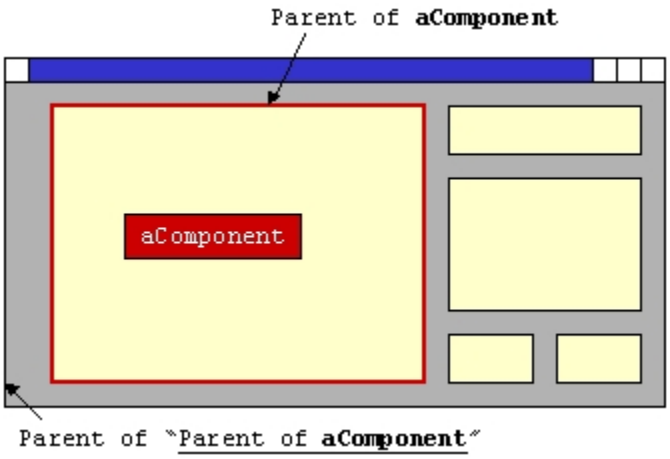
There are some interesting things to note:

- **Panels** are **Containers** (i.e. they clearly contain many components).
- **Containers** are **Components** (i.e. recursive definition).
- All **JComponents** are also **Containers** (i.e., in Swing, everything is a **Container**)
- A **JFrame** is a **Window** which is a **Container**.

Now, how is everything held together ?

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

```
Component   aComponent = ... ;
Container parent = aComponent.getParent();
Container parentOfParent = parent.getParent();
```



- Containers keep pointers to their components (i.e., an array):

```
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 ***Panel****:*

A **Panel** is:

- the simplest container class
- is itself contained (i.e., it itself is contained within a container)
- provides space in which an application can attach any other component (including other panels)
- does not create a separate window of its own like **Frame**.

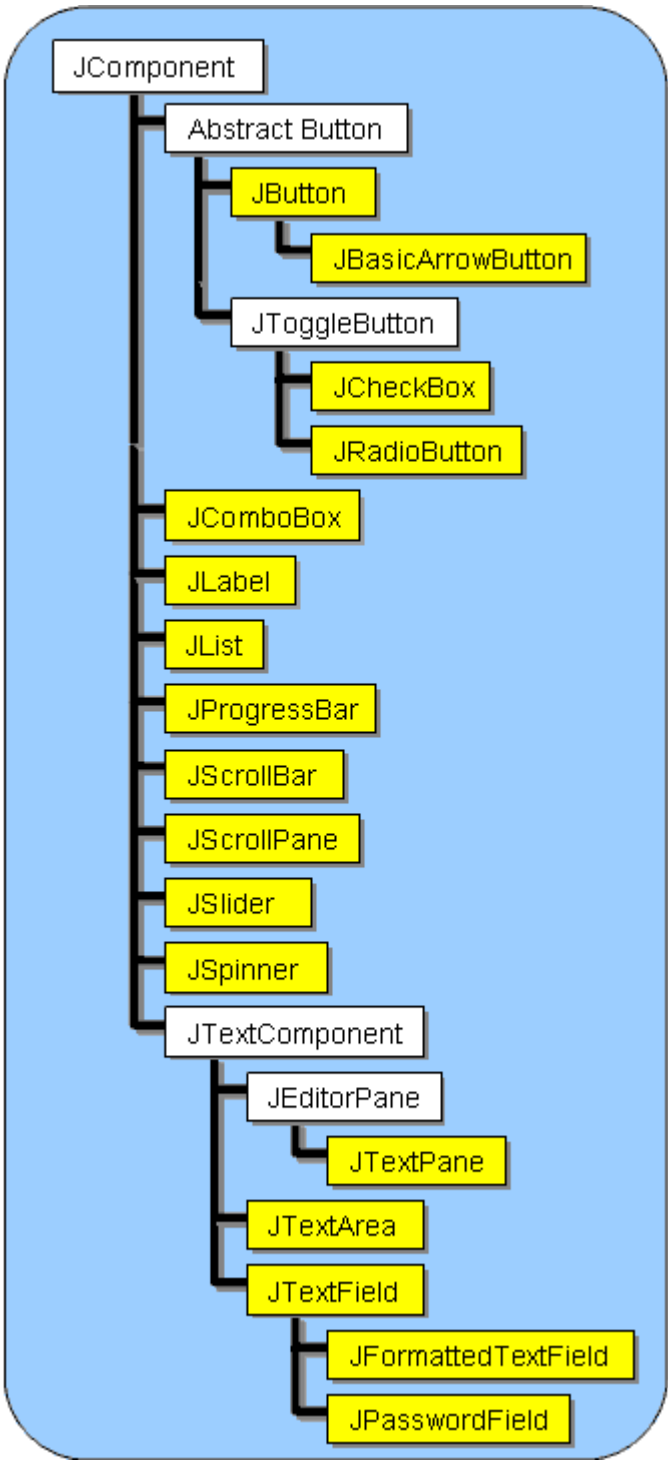
Think of a panel as a bulletin board that you can fill with components and then you can place the bulletin board anywhere as a component itself.

All **JFrames** have a **JPanel** at the top level to which everything is 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:

```
JFrame frame = new JFrame("MyApplication");
frame.getContentPane().add(aComponent);
frame.getContentPane().add(anotherComponent);
frame.getContentPane().add(yetAnotherComponent);
```

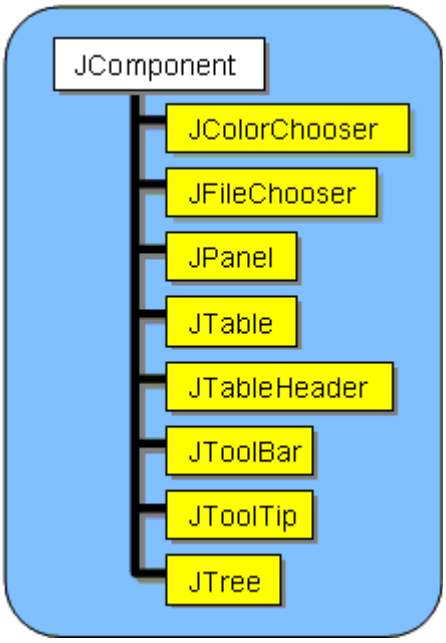
We will see more on this in our upcoming examples.

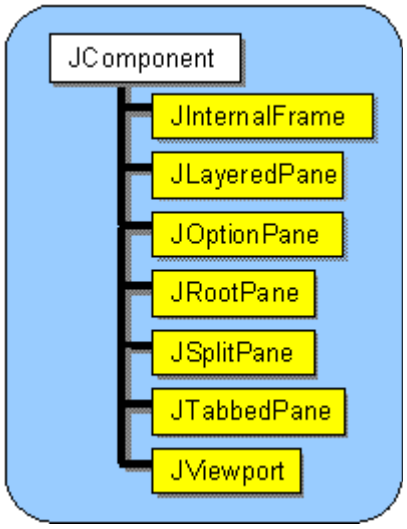
Note that the diagram above did not expand the **Component** hierarchy ... there are also standard AWT components that are similar to those of the **Swing** set. Now let us look at the Swing **JCompoment** hierarchy in more detail. Here are some of the **JComponent** subclasses for common components (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.

In addition to these classes, there are some more advanced classes (some of which we will look at later). Below are two more pieces of the **JComponent** hierarchy (note that there was too much to show on one picture so it has been split into multiple pictures):

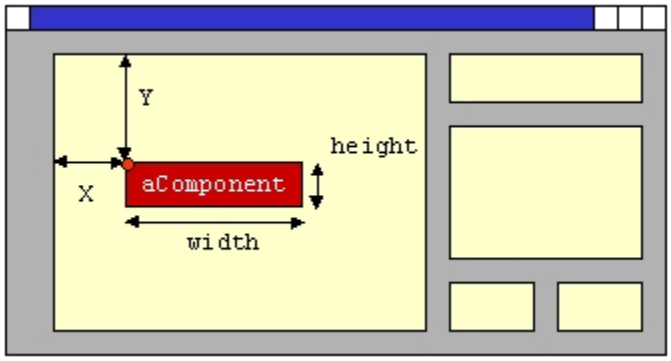




There are even more subclasses ... those dealing with menus will be shown later.

All **JComponents** have the following state:

1. **Location, Width and Height (in pixels):**
   - location is an (x,y) coordinate with respect to top left corner of parent container:



We can access/modify this information from the component at any time:

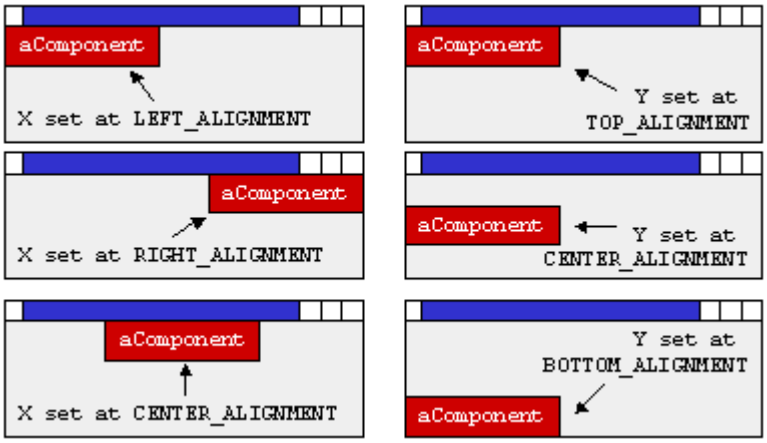
```
JComponent   c = ... ;
// ask a component for its location
int x = c.getX();
int y = c.getY();
// ask a component for its width or height
int w = c.getWidth();
int h = c.getHeight();

// 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" that automatically compute component locations and sizes. We can disable these layout managers or make use of them (as we will see later). If we decide to use layout managers, we can "suggest" sizes for our components using these (for some **ints** x and y), but this does not always work :(:

```
c.setMaximumSize(new Dimension(x, y));
c.setMinimumSize(new Dimension(x, y));
c.setPreferredSize(new Dimension(x, y));
```

2. Vertical and Horizontal Alignment

- options are left, right, top, bottom or center (as specified by class constants):




```
JComponent   c = ... ;
// we can set a component's X alignment to one of
// LEFT_ALIGNMENT, RIGHT_ALIGNMENT, CENTER_ALIGNMENT
c.setAlignmentX(Component.LEFT_ALIGNMENT);

// we can set a component's Y alignment to one of
// TOP_ALIGNMENT, BOTTOM_ALIGNMENT, CENTER_ALIGNMENT
c.setAlignmentY(Component.TOP_ALIGNMENT);
```

As we will see, these attributes are only useful when we use layout managers. Otherwise we can simply set the exact locations with **`setLocation()`**.

3. Background and Foreground Colors:

The background is the "fill" color behind the text, while the foreground is usually used as the text color.

```
JComponent   c = ... ;
c.setBackground(Color.red);
c.setForeground(Color.White);
```

There are many 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
```

You can also make your own **Color** object by specifying the amount of red, green and blue in them as integers (between 0 and 255) or as floats (between 0.0 and 1.0):

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

You can use the **`getRGB()`** method to return an int representing the color of a component like this:

```
myColor.getRGB();
```

4. **Font Types and Font Sizes:**

   You can choose the type of font on your component (e.g., button, text field etc.):

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

   Here is the format for making Font objects:

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

   Here are some examples of typefaces (names):

   ```
   "Times", "Serif", "SansSerif", "Courier",
   ```

   Here are possible Styles (notice that we can "OR" them together):

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

   The **getAllFonts()** method of the **`GraphicsEnvironment`** class returns an array of all font faces available in the system.

5. **Ability to be Enabled/Disabled:**

   Sometimes we want to disable a component so that it cannot be selected or controlled by the user. We can enable and disable components at any time on our program.

   ```
   JComponent   c = ... ;
   c.disable();
   ...
   c.enable();
   ```

   While a component is disabled, it is "greyed out", but cannot be used.

6. **Ability to be Hiden/Shown:**

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

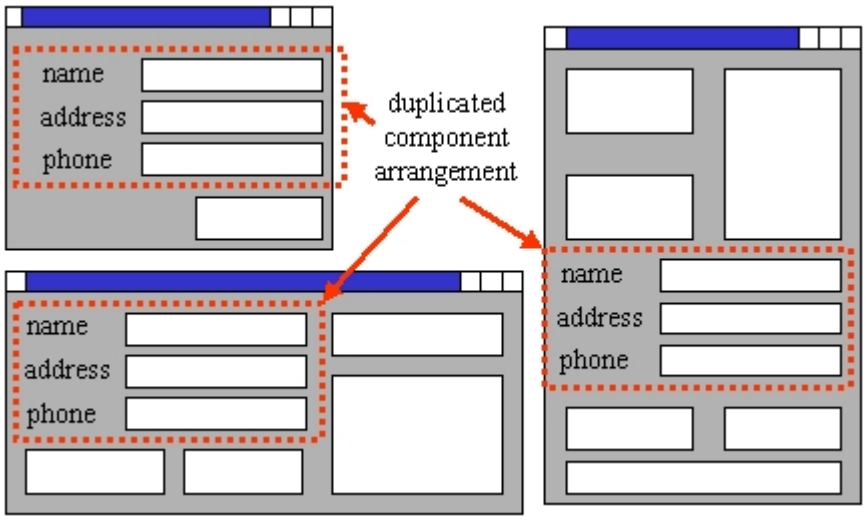
   While a component is hidden, it is invisible. By default, components are automatically visible (not **JFrames** though)

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

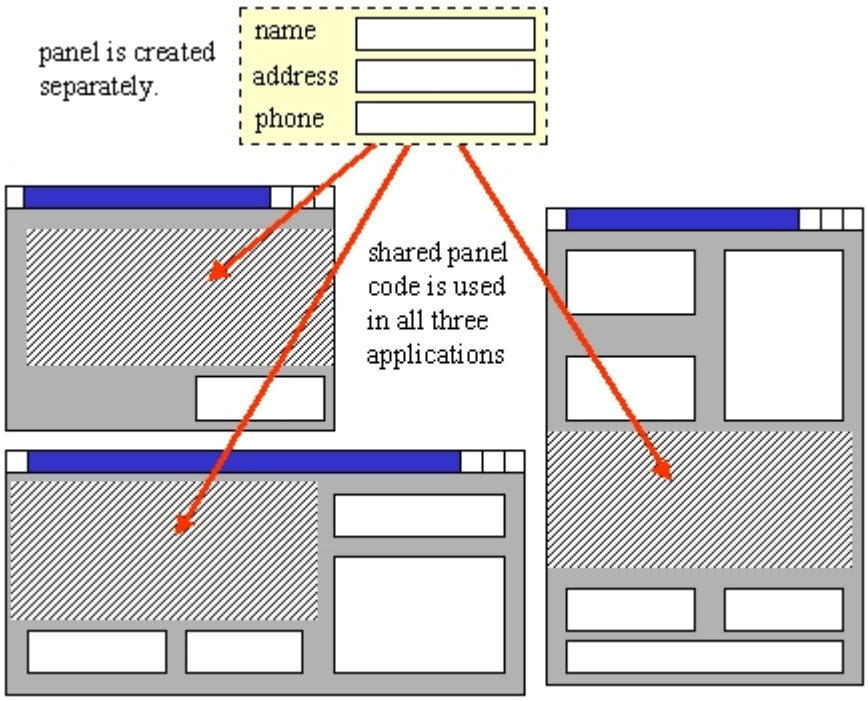
Using JPanels in a Basic Application

An application's window may contain many components. 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.



To do this, we often lay out components onto a panel and then place the panel 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.



We will follow this approach. The typical steps for creating an application are as follows:

1. create a subclass of **JPanel** containing your application components (i.e., add components one at a time to the panel)
2. create subclass of **JFrame** and add your panel to it.
3. test it to see if it looks the way you want it to.

Although we could have simply added our components directly to a **JFrame**, this will allow us to re-use the **JPanel** (perhaps in some future application). We will be seeing many examples later in which we do not make a separate **JPanel**. The reason for doing the examples without a **JPanel** is so that the example code is simpler to explain in the notes.

Here is an example in which we create a simple application with two classes. The first class is a special kind of panel (we will make it a subclass of **JPanel**). We will add two labels and 2 buttons to

the panel. The second class is a frame that will hold the panel and represent the main application. Here is the panel file that specifies the objects to be placed on the window:

```
import javax.swing.*;
import java.awt.*; //needed to use components setting methods (e.g.,
colors, fonts)
public class PanelWithFourComponents extends JPanel {

    public PanelWithFourComponents() {
        // Create and add a simple JLabel to the panel
        JLabel plainLabel = new JLabel("Plain Small Label");
        add(plainLabel);

        // Create a 2nd JLabel with a 32pt bold italic Serif font
        // and a "brain.gif" picture to the left of the text.
        // Make the label have a red background and white text.
        JLabel fancyLabel = new JLabel("Fancy Big Label");
        fancyLabel.setFont(new Font("Serif", Font.BOLD | Font.ITALIC,
32));
        fancyLabel.setIcon(new ImageIcon("brain.gif"));
        fancyLabel.setHorizontalAlignment(JLabel.RIGHT);
        fancyLabel.setBackground(Color.red);
        fancyLabel.setOpaque(true);
        fancyLabel.setForeground(Color.white);
        add(fancyLabel);

        // Create a JButton
        JButton button1 = new JButton("Button");
        button1.setBackground(Color.blue);
        button1.setForeground(Color.yellow);
        add(button1);

        // Create a 2nd JButton, this one with an icon
        JButton button2 = new JButton("Brain", new
ImageIcon("brain.gif"));
        button2.setBackground(SystemColor.control);
        add(button2);

        // Set the background color of the panel
        setBackground(Color.green);
    }
}
```

Notice the following:

- We added two labels ... the first was simple text ... the second had a fancy font, coloring and an icon attached to it.
- We also added two buttons.
- ".gif" (possibly animated gifs) or ".jpg" files can be shown in the application by using the **ImageIcon** class.
- Different fonts can be used on labels, we can even **bold** them, make them *italics*, and change their size.
- Both buttons and labels can have an associated icon which can be aligned left/right of the text.
- **`SystemColor.control`** lets us specify the standard operating system control color (in windows, it is gray).
- Objects are merely added one-by-one to the panel ... we will explain later how they are arranged.

Here is the main application file:

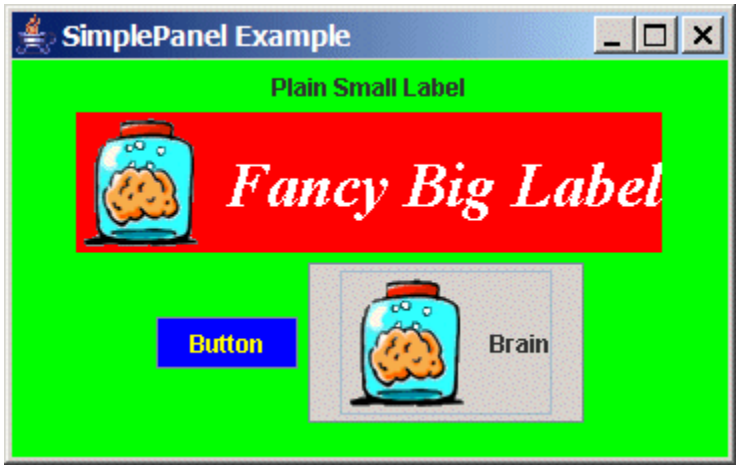
```
import javax.swing.*;
public class SimplePanelTestFrame extends JFrame {
```

```
    public SimplePanelTestFrame(String title) {
        super(title);  // Must be first line

        add(new PanelWithFourComponents());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(650, 200);
    }

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

Note the following:

- We added the panel that we created earlier to the frame.
- The frame of the window also has its own background color, but this is hidden by the green panel which is on top of it.

We will have to make sure that the **brain.gif** file is in the directory from where our code is running.



2.3 Layout Managers

As we all know ... JAVA was developed for the internet and JAVA applications were meant to be run 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.

- A **`LayoutManager`** is an interface (**java.awt.LayoutManager**)
- It defines methods necessary for a class to be able to arrange **Components** within a **Container**
- There are 8 useful layout classes that implement **LayoutManager**:
    - **FlowLayout**
    - **BorderLayout**
    - **CardLayout**
    - **GridLayout**
    - **GridBagLayout**
    - **BoxLayout**
    - **OverlayLayout** (not described in these notes)
    - **SpringLayout** (not described in these notes)

- Layouts are set for a panel using the **`setLayout()`** method. If set to **null**, then no layout manager is used.

So why should we use a LayoutManager ?

- 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

Before we look at some of these layout managers, we will first see how to lay components out without using them.

2.3.1 Null Layout

We can actually set the layout for our window panel to be **null** so that we have full control on the exact locations of all objects. This means that we are NOT using a layout manager. So, we must arrange the components on our own.

The **advantages** of not having any layout manager are:

- You can specify exactly where you want to place the components.
- Much simpler than some of the layout managers such as **GridBagLayout**

The **disadvantages** are:

- You do not get to specify any resizing behaviour, so your window does not resize properly !!
- Usually, your component locations rely on other component locations, so moving a single component may cause you to want to move other components around again.
- You MUST pre-plan your entire window (this is actually be a good thing to do anyway) by specifying the exact locations of components and figure out the spacing between the components that you would like to use.

To choose to use no layout manager, we just send the following message to a panel:

```
setLayout(null);
```

We must then specifically place all of our components by using the following:

```
//set the location of the component within its container
void setLocation(int x, int y);

//set the dimensions of the component
void setSize(int width, int height);
```

Things to note:

- The locations and sizes are in pixels.
- Locations are always with respect to the top left corner of the container which is location (0,0).
- Also, the top left corner of the component will appear at the specified location.

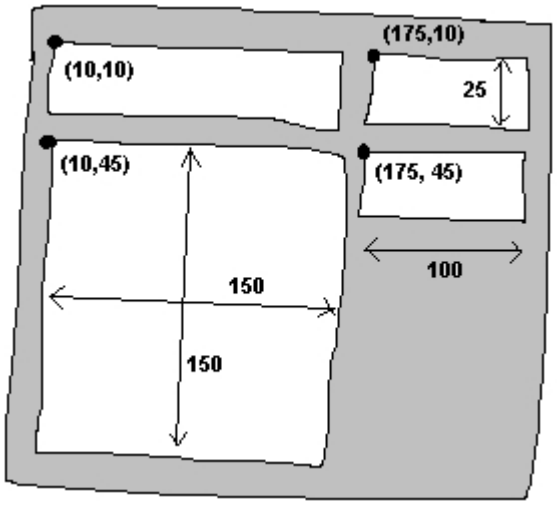
Components are added to containers by using the **`add()`** method as before:

```
anyContainer.add(aComponent);
```

Example

Here is an example showing how to build a simple window with a text field, a list, and two buttons.

First, we need to sketch out the window on paper:



Here is the code:

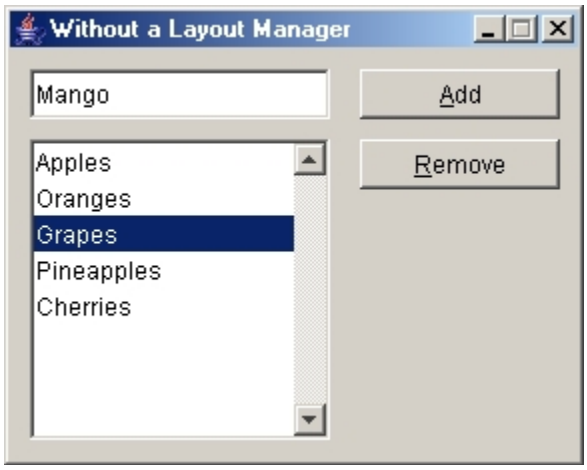
```
import javax.swing.*;
public class NoLayoutExample extends JFrame {
    public NoLayoutExample(String name) {
        super(name);

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

        //The text field
        JTextField newItemField = new JTextField();
        newItemField.setLocation(10,10);
        newItemField.setSize(150,25);
        add(newItemField);

        //The Add button
        JButton addButton = new JButton("Add");
        addButton.setMnemonic('A');
        addButton.setLocation(175, 10);
        addButton.setSize(100,25);
        add(addButton);

        //The List
        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);
        scrollPane.setLocation(10,45);
        scrollPane.setSize(150,150);
        add(scrollPane);

        //The Remove button
        JButton removeButton = new JButton("Remove");
        removeButton.setMnemonic('R');
        removeButton.setLocation(175,45);
        removeButton.setSize(100,25);
        add(removeButton);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(290, 230); // manually computed sizes
        setResizable(false);
    }

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

```
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch(Exception e) {}
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new NoLayoutExample("Without a Layout Manager");
        frame.setVisible(true);
    }
}
```

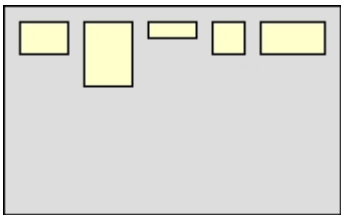
Note the following:

- The ***mnemonic*** for a button allows you to press **ALT** along with the character to select the button. It acts as if you clicked the button.
- You can put something in a **JTextField** using one of the following:
  - `newItemField = new JTextField("some text");`
  - `newItemField.setText("any text at all");`
- You can put something in a **JList** using the following:
  - `String[] stuff = {"Apples", "Oranges", "Grapes", "Plums"};`
    `itemsList = new JList(stuff);`
- JLists are usually placed within a **JScrollPane** so that you can have scroll bars automatically on them.
- We set the window to be non-resizable. Since our components will not move or grow anyway, there is no use in allowing a larger window.

Here is what the window looks like when running our code:



2.3.2 FlowLayout



- components are arranged horizontally from left to right, like lines of words in a paragraph.
- if no space left on current "line", components flow to next line
- components are centered horizontally on each line by default
- often used to arrange buttons in a panel

There are three constructors:

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

- align may be any one of three class constants: **LEFT**, **RIGHT**, **CENTER**

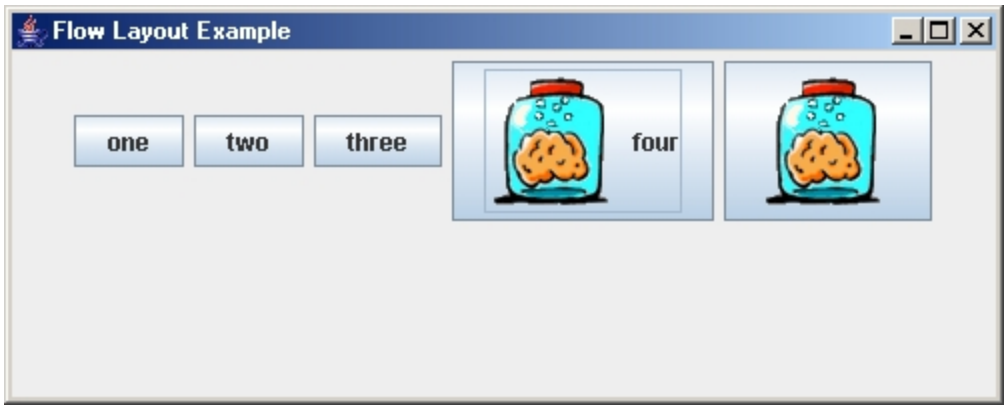
- specifies how components are justified
- **hGap** and **vGap** specify the horizontal and vertical pixels between components

Example

Here is a simple example that adds 5 buttons (two with icons) to a panel which uses a **FlowLayout**.

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

     public FlowLayoutManagerExample (String title) {

          super(title);

          setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
          add(new JButton("one"));
          add(new JButton("two"));
          add(new JButton("three"));
          add(new JButton("four", new
          ImageIcon("brain.gif")));
          add(new JButton(new ImageIcon("brain.gif")));

          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setSize(500, 200);

     }
     public static void main(String args[]) {
         FlowLayoutManagerExample frame = new
     FlowLayoutManagerExample("Flow Layout Example");
        frame.setVisible(true);
     }

}
```

Here is the result obtained when the application is run:

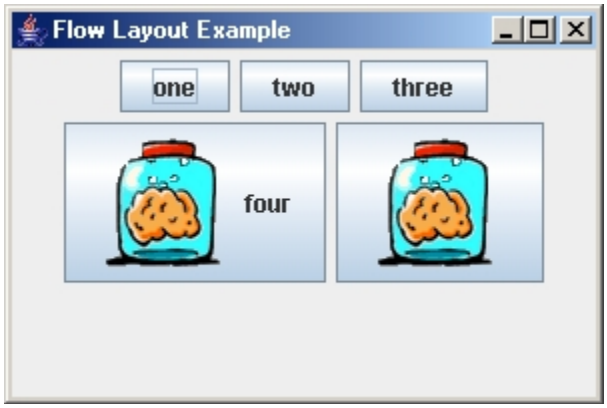


Notice

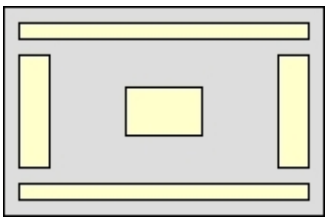
- we are using the default cross-platform look and feel, which makes the buttons look "metallic".
- **FlowLayout** (and all the other layout managers) are in the **java.awt** package, so we imported **java.awt.***
- the components are merely added left to right until no more room is available.
- the components are centered on the window; although we could have instead used **FlowLayout.LEFT** or **FlowLayout.RIGHT** in the constructor to have the left or right aligned, respectively.
- There is a 5 pixel gap between the components, which we could have made larger or smaller in

the constructor.

Try resizing the window to 300x200. The components will be rearranged accordingly:



2.3.3 BorderLayout



- This is the default layout for a JFrame
- Divides the container into regions: north, south, east, west and center.
- Regions are specified using class constants (`BorderLayout.NORTH`, `BorderLayout.SOUTH`, etc....)
- A single component (which may itself be a container) fills each region

There are two useful constructors:

```
public BorderLayout()
public BorderLayout(int hgap, int vgap); //allows spacing between
components
```

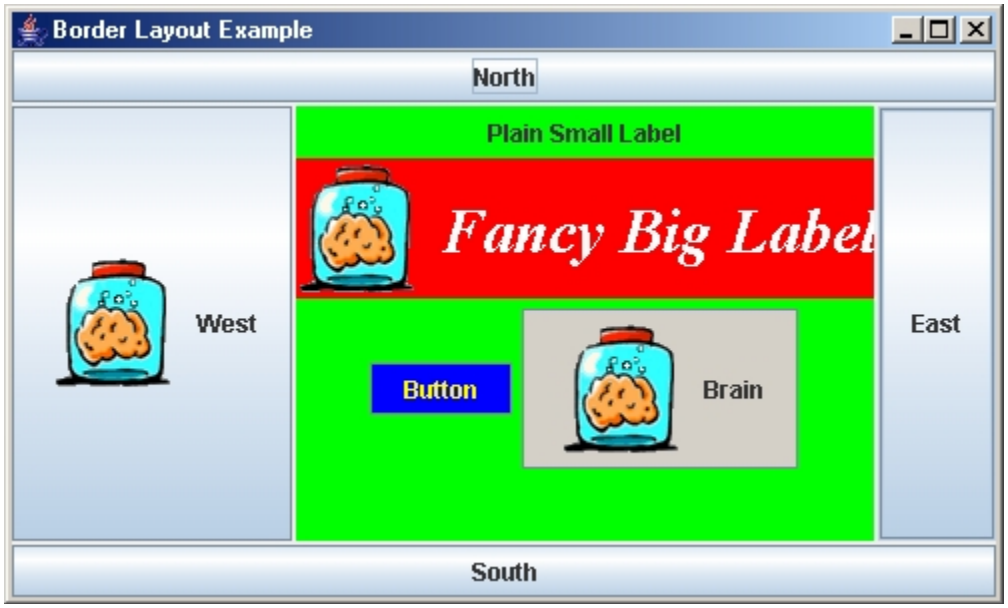
Example

In this example, we add 4 buttons along the north, south, east and west of the window. We also make use of our previous code which created a panel with 2 labels and 2 buttons and add this to the center of the pane, just for fun.

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

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

          // the JFrame already has a border layout by
          default, but
          // by doing this, we also get to specify a 2 pixel
          gap (i.e.,
          // margin) between components.
          setLayout(new BorderLayout(2,2));

          add(BorderLayout.NORTH, new JButton("North"));
          add(BorderLayout.SOUTH, new JButton("South"));
          add(BorderLayout.EAST, new JButton("East"));
          add(BorderLayout.WEST, new JButton("West", new
          ImageIcon("brain.gif")));
```

```
            add(BorderLayout.CENTER, new
            PanelWithFourComponents());

            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setSize(500, 300);

        }
        public static void main(String args[]) {
            BorderLayoutManagerExample frame = new
        BorderLayoutManagerExample("Border Layout Example");
            frame.setVisible(true);
        }

    }
```

Here is the result:

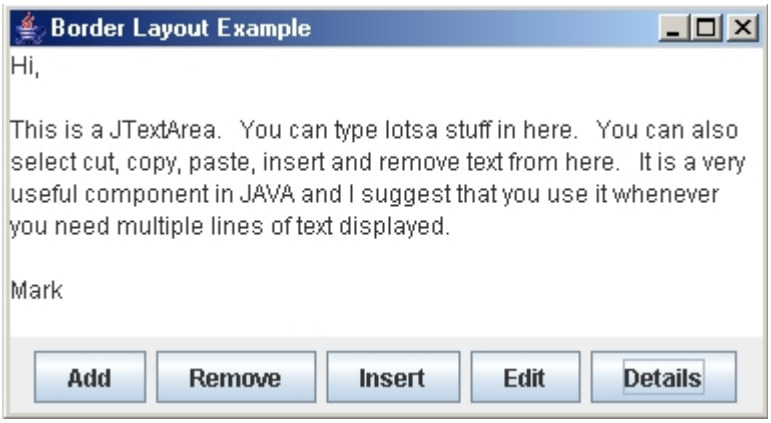


Note that you do NOT have to put something in all of the border locations. You may simply want to just use the top/bottom or perhaps left/right/bottom. A common situation is to have a panel of components along the bottom or side of a window.

For example, by using this code (which makes a panel of buttons on the bottom of the window):

```
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"));
add(BorderLayout.SOUTH, buttonPanel);
add(BorderLayout.CENTER, new JTextArea());
```

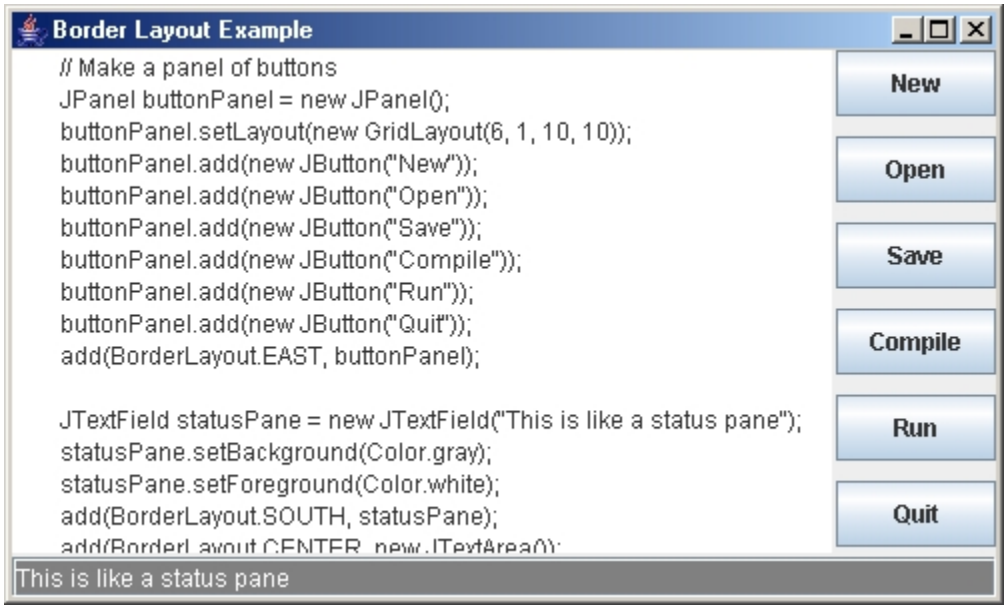
we obtain this window:



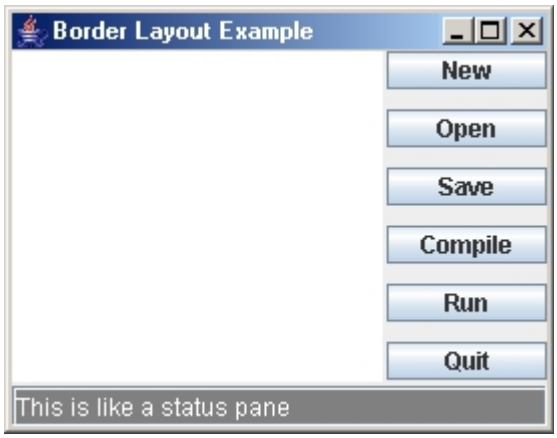
Or, perhaps we would like to place some buttons on the right side of the window and maybe a status pane or progress bar on the bottom. The following code does this (while making use of a GridLayout ... which we will talk about soon):

```
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"));
add(BorderLayout.EAST, buttonPanel);

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

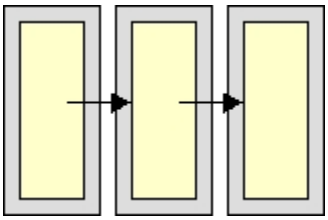
produces this window:



As you can see below, the window still resizes nicely automatically:



2.3.4 CardLayout



- Layout manager for a container....each component is treated as a "card"
- Displays components one a time, only one is visible at a time
- Often used for swapping panels of components in and out
- Used for managing a set of panels that present themselves as a stack of tabbed folders
- User can interact with cards like a **slide show**

There are two constructors:

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

Stacking methods include the following:

```
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)
```

Example

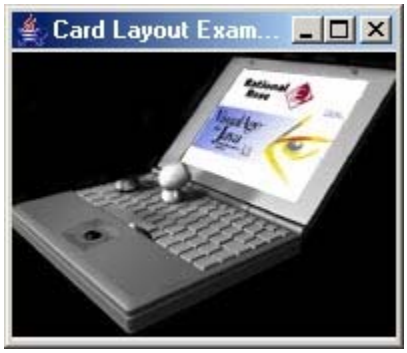
This example creates a window with three cards which are displayed using a **CardLayout** manager. Since only one card can be shown at a time, only one image appears on the window. We can select the image by using the **show()** method for the card layout manager to specify which one to show. Later in the notes, we will see how to hook up some buttons to get the cards showing as a slide show when buttons are pressed.

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

     private CardLayout cardLayoutManager;
     public CardLayoutManagerExample(String title) {

          super(title);

          CardLayout layoutManager = new CardLayout(0,0);
          setLayout(layoutManager);

          // Add (and give names to) components using the
          layout manager
```

```
            JLabel first = new JLabel(new
            ImageIcon("trilobot.jpg"));
            JLabel second = new JLabel(new
            ImageIcon("laptop.jpg"));
            JLabel third = new JLabel(new
            ImageIcon("satelite.jpg"));
            add("first", first);
            add("second", second);
            add("third", third);

            // Pick the component to show, in this case, the
            first
            layoutManager.show(getContentPane(), "first");

            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setSize(200,172);

      }

      public static void main(String args[]) {
            CardLayoutManagerExample frame =  new
      CardLayoutManagerExample("Card Layout Example");
           frame.setVisible(true);
      }

}
```

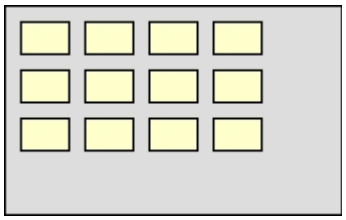
Notice that the **show()** method requires us to pass in something called the "contentPane". This is actually the "hidden" **JPanel** of the **JFrame** on which everything is placed. Here is what the window looks like when run:

If we were to have changed the 2nd parameter in the **show()** method to "second" or "third", we would get the other images showing instead:



Note that we are simply showing cards that have a single **JLabel** on them (which has a picture on it). However, remember that you can actually place any **JComponent** or **Container** (e.g., **JPanel**) on the card. So, for example, we could have added our P**anelWithFourComponents** as a single card.

2.3.5 GridLayout



- Lays out its components into a rectangular grid
- Often used for calendar or spreadsheet type user interfaces
- Components must be added row by row, left to right

There are two constructors:

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns, int hGap, int vGap)
            throws IllegalArgumentException;
```

Example

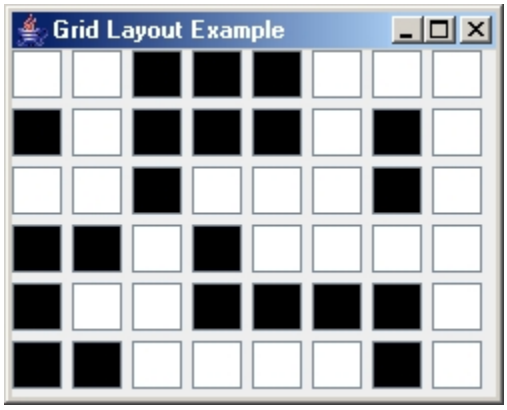
We have already seen in our BorderLayout manager example that we can create a simple grid of buttons. There we created a 6 row, 1 column panel of buttons which were all equally sized:

```
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"));
```

Now let us create a window with a 6x8 grid of components on them. For simplicity, we will use all JButtons and randomly set their colors to white or black. We can easily modify the code to have images on the buttons or text, or even to replace the buttons with arbitrary components or panels.

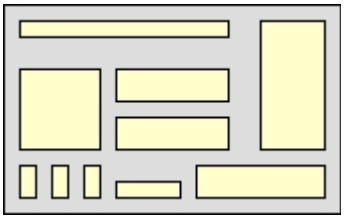
```
import java.awt.*;
import javax.swing.*;
public class GridLayoutManagerExample extends JFrame {

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

        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);
                add(b);
            }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250, 200);
    }
    public static void main(String args[]) {
        GridLayoutManagerExample frame = new
GridLayoutManagerExample("Grid Layout Example");
        frame.setVisible(true);
    }
}
```

Here is the result:



Notice that the buttons are all evenly sized and evenly spaced. There is some extra margin along the right and bottom as leftover space that cannot be evenly distributed among the buttons. This example shows the entire window using the **GridLayout**, but remember that any **JPanel** can use this layout so this grid can be applied to a panel that is only one of many components in a window.

2.3.6 GridBagLayout

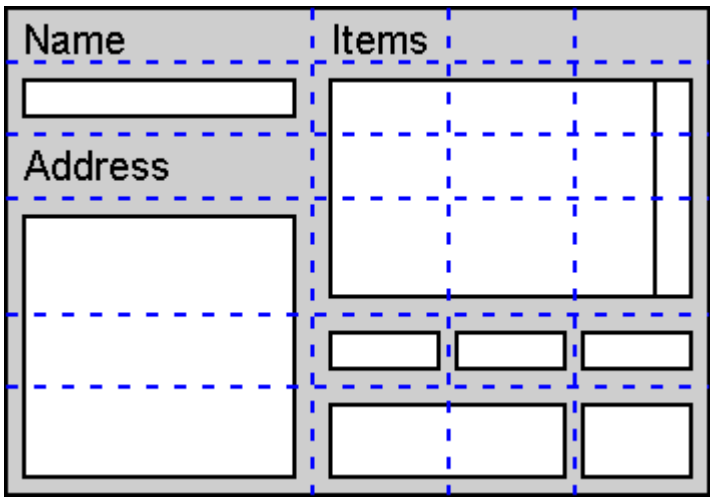


- For complicated layout needs (typically very useful ... since we have full control).
- Also arranges components in a grid, but the exact grid size is not explicitly defined.
- Rows and columns of grid may have different sizes (see image below)
- Components can occupy (i.e., span across) multiple rows and columns (see image below)
- For each component, there is an elaborate set of constraints for determining how much space is used by the component

We will create ***GridBagConstraint*** objects.

- They are used to package together a set of constraints 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.

Here is an example of a window showing the breakdown of the components onto a grid:

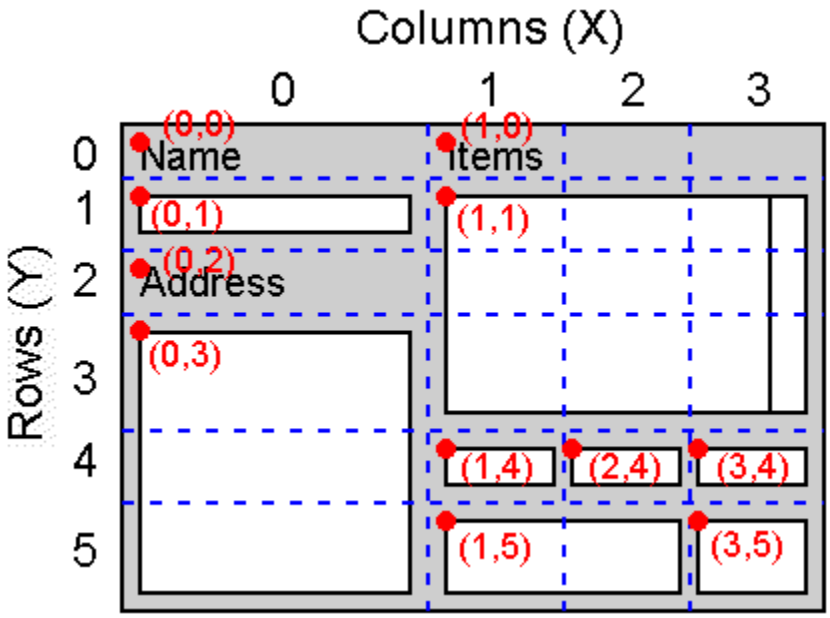


We will now take a look at the many constraints which we can use for each component:

- **gridx, gridy**

  Specifies the grid **cell** (column=x and row=y) that the upper left of the component will be displayed in, where the upper-left-most grid ***cell*** has address **gridx** = 0, **gridy** = 0.

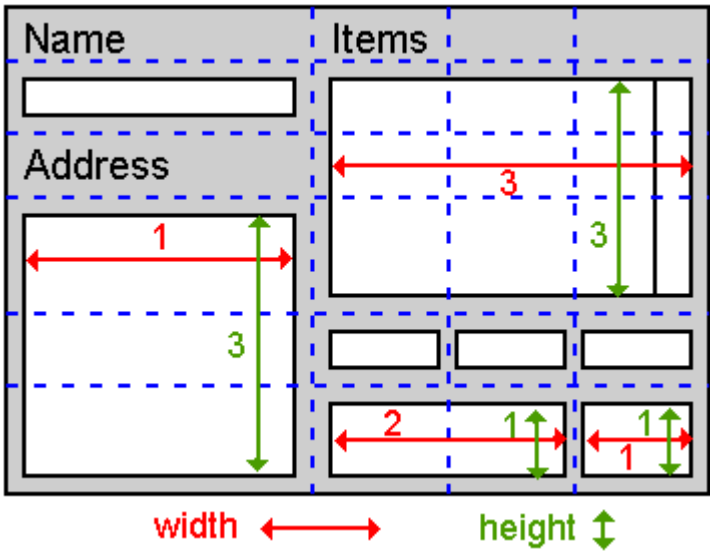
  - Use **`GridBagConstraints.RELATIVE`** (the default value) to specify that the component should be placed just to the right of (for **gridx**) or just below of (for **gridy**) the component that was added to the container just before this component was added.



- **gridwidth, gridheight**

  Specifies the number of columns (**gridwidth**) and rows (**gridheight**) that the component will occupy. The default value is 1.

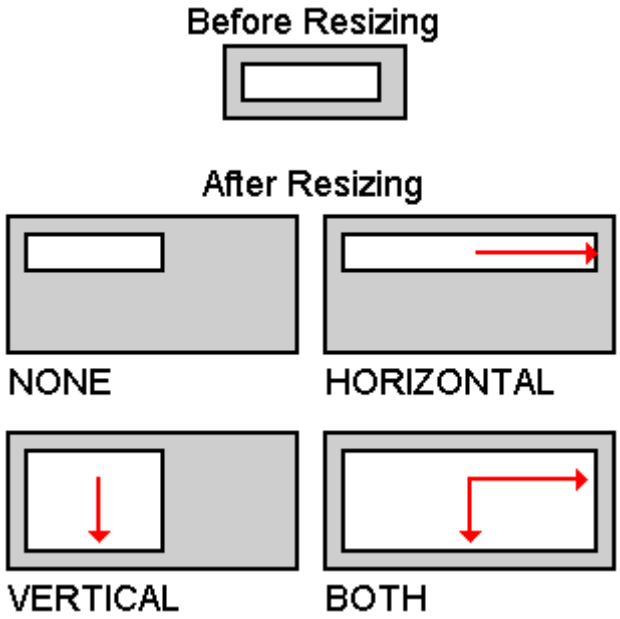
  - Use **`GridBagConstraints.REMAINDER`** to specify that the component be the last one in its row (for **gridwidth**) or column (for **gridheight**).



- **fill**

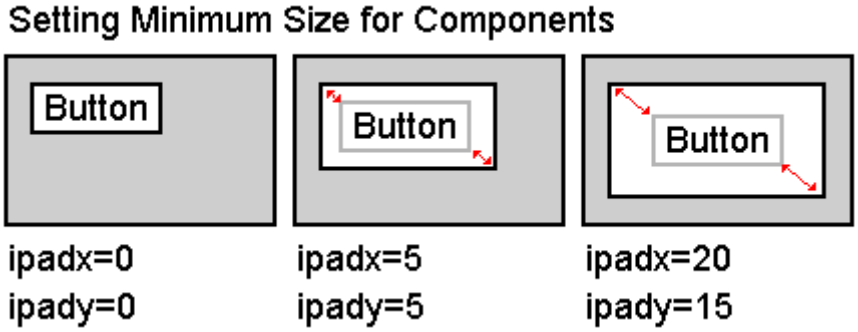
Used in resizing when the component's display area is larger than the component's requested size to determine how to resize the component. Possible values are:

- **`GridBagConstraints.NONE`** (the default - the component will not grow in either direction)
- **`GridBagConstraints.HORIZONTAL`** (make the component wide enough to fill its display area horizontally, but don't change its height),
- **`GridBagConstraints.VERTICAL`** (make the component tall enough to fill its display area vertically, but don't change its width), and
- **`GridBagConstraints.BOTH`** (make the component fill its display area entirely).



- **ipadx, ipady**

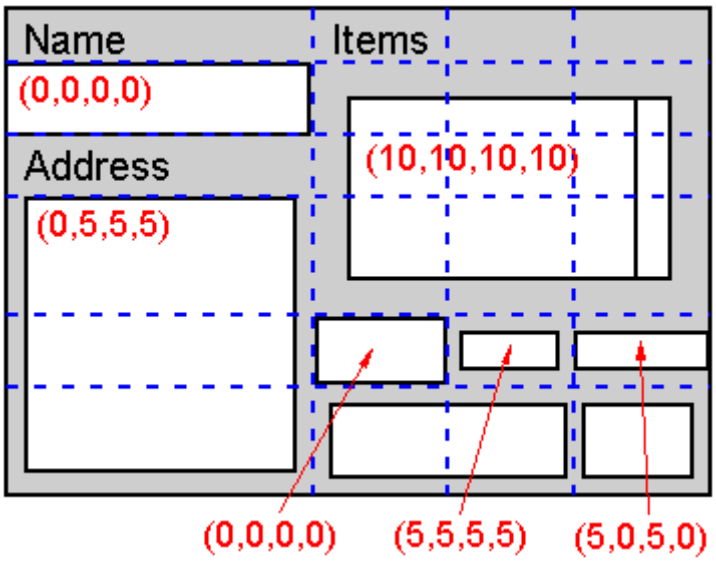
Specifies the component's internal padding (spacing around the component) within the layout, how much to add to the minimum size of the component. Since the GridBagLayout manager lays out the components such that the grid sizes are not specified explicitly, each component is given a minimum size. Sometimes, we would like a component to have a larger minimum size. Using these constraints we can set the width of the component to be at least its minimum width plus (**ipadx** * 2) pixels (since the padding applies to both sides of the component). Similarly, the height of the component will be at least the minimum height plus (**ipady** * 2) pixels.



- **insets**

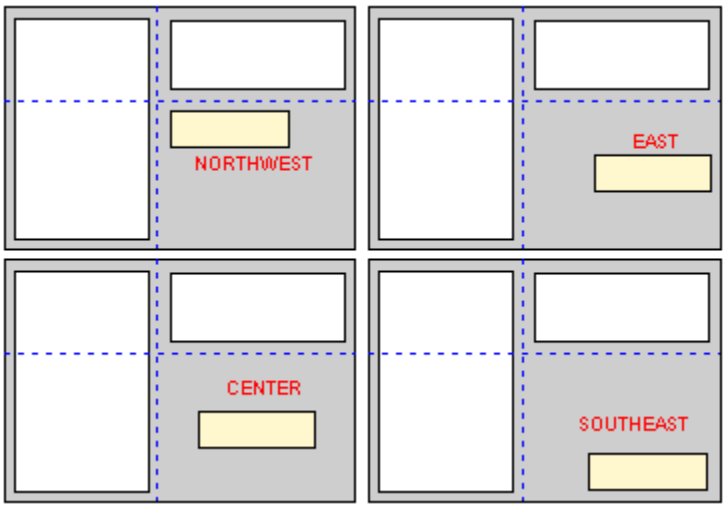
Specifies the component's external padding, the minimum amount of space between the component and the edges of its display area. To do this, we make an instance of the

Insets class (e.g. `new Insets(10, 10, 10, 10)`). The order of the parameters for this constructor is top, left, bottom then right.



- **anchor**

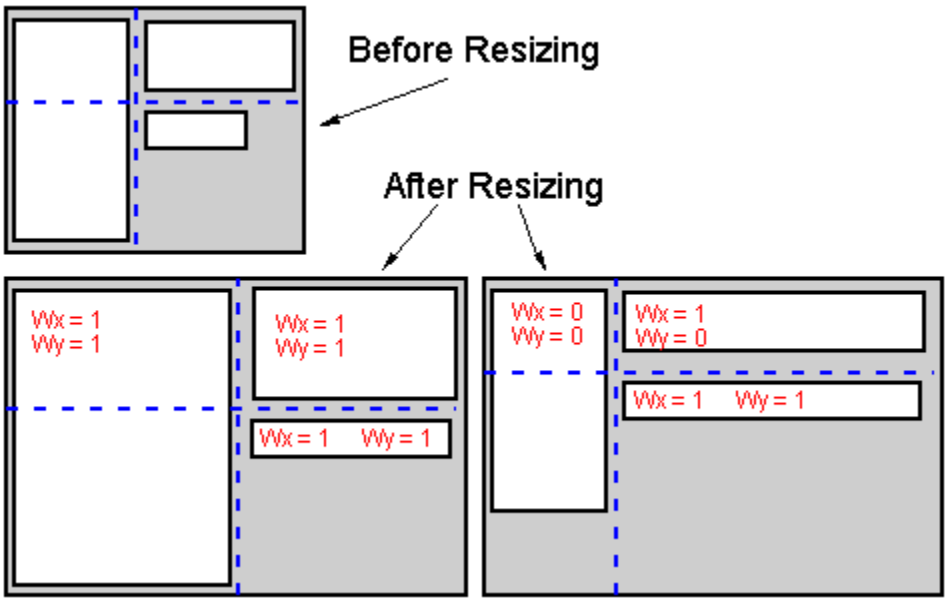
  Used when the component is smaller than its display area to determine where (within the display area) to place the component. Also, when resizing, this allows the component to be fixed at a corner or edge. Valid values are:
  - **`GridBagConstraints.CENTER`** (the default),
  - **`GridBagConstraints.NORTH`**,
  - **`GridBagConstraints.NORTHEAST`**,
  - **`GridBagConstraints.EAST`**,
  - **`GridBagConstraints.SOUTHEAST`**,
  - **`GridBagConstraints.SOUTH`**,
  - **`GridBagConstraints.SOUTHWEST`**,
  - **`GridBagConstraints.WEST`**, and
  - **`GridBagConstraints.NORTHWEST`**



- **weightx, weighty**

  Used to determine how to distribute space when resizing the window. A zero value indicates the component does not grow horizontally/vertically on its own. Components with larger weight values will occupy more of the additional space than components with

small weight values. Unless you **specify a weight for at least one component** in a row (**weightx**) and column (**weighty**), all the components clump together in the center of their container. This is because when the weight is zero (the default), the **GridBagLayout** object puts any extra space between its grid of cells and the edges of the container.

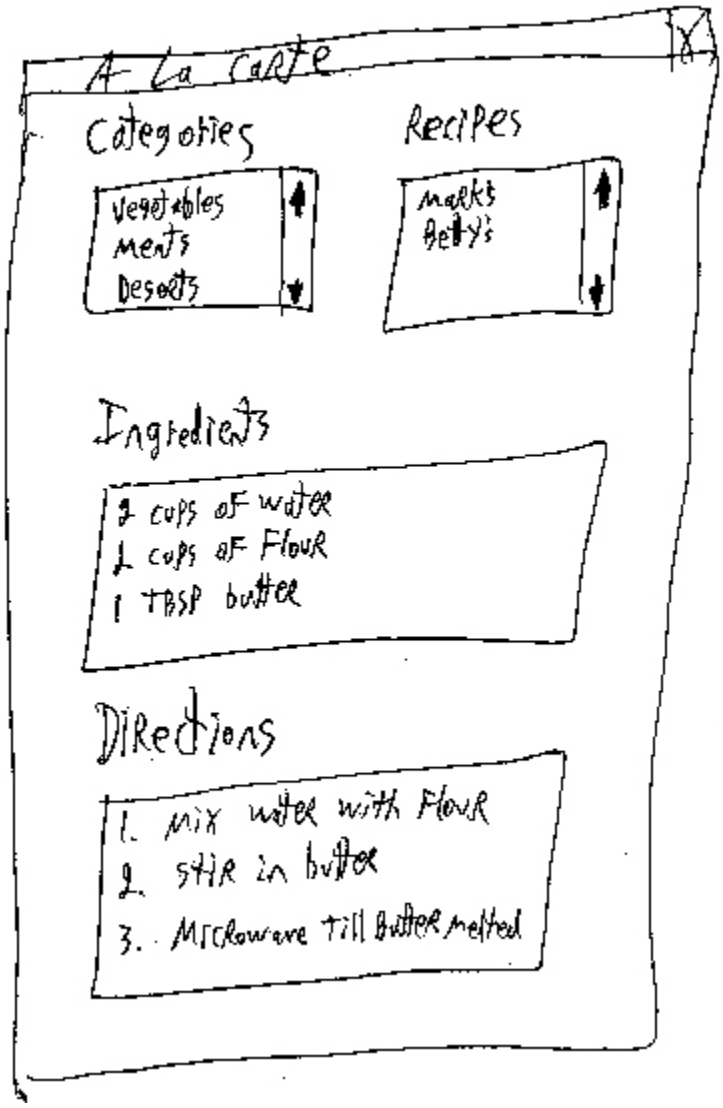


In order to fully understand the effects of these constraints, you should create an example piece of code and try changing the values. Resize the window to see the effects of your changes. Experience is best to help you understand.

Example

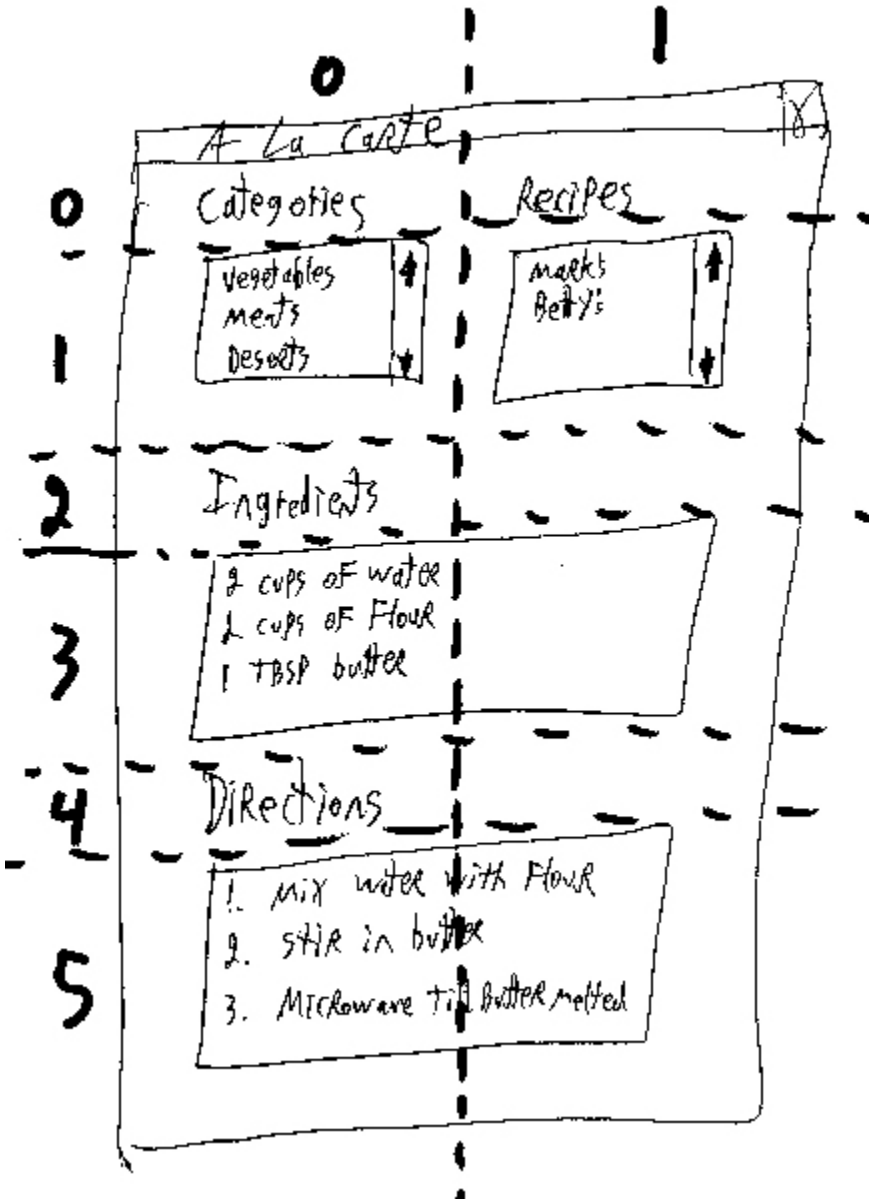
This example represents a kind of "recipe" that you can follow when using the GridBagLayout managers. The example is taken from Core Java 2 volume 1-Fundamentals, by C.S. Horstmann and G. Cornell, Sun Microsystems, 1999.

Step 1:
Sketch out the component layout on a piece of paper (i.e., the way you want the window to look). Note that it will not ALWAYS look the EXACT way that you want it to when its done, but you can get a rough idea as to the layout:



Step 2/3:

Identify the different components (i.e., labels, text fields, lists etc...) and add grid lines to your drawing such that the small components are each contained in a cell and the larger components span multiple cells. Label the rows and columns of your grid with 0,1,2,3, ... You can now read off the **`gridx`**, **`gridy`**, **`gridwidth`**, and **`gridheight`** values.



Step 4:
Now worry about the resizing issues. For each component, ask yourself whether it needs to fill its cell horizontally or vertically. If not, how do you want it aligned ? This tells you the **`fill`** and **`anchor`** parameters.

Step 5:
Set all weights to 100. However, if you want a particular row or column to always stay at its default size, set the **`weightx`** or **`weighty`** to 0 in all components that belong to that row or column. ***MAKE SURE that at least one component in each row and column has a non-zero weight!!***

Step 6:
Write the code. Carefully double-check your settings for the **GridBagConstraints**. One wrong constraint can ruin your whole layout and waste you hours of time :(. Also, it is often the case that some components act strangely when the window is resized. It sometimes seems as though your weight settings are being ignored. This is often due to the way that the **GridBagLayout** manager tries to determine starting (or preferred) sizes for the components. So, it is sometimes necessary to specify that your objects are to be treated equally in this regard. You can do this by setting the preferred size of your growable components to some similar value. For example:
`scrollPane.setPreferredSize(new Dimension(10,10));` It is likely that you may have to play a little with the insets and margins of your components to get them to the size that you want.

Step 7:

Compile, run, and enjoy. Note that your gridlines may not always be exactly where you wanted them and that your components may not be the size that you want. You may have to play around by adding additional grid lines and specifying that a component spans multiple grid cells.

Here is an example of a recipe browser build using the above steps. Look at the code and pay particular attention to the **gridx**, **gridy**, **gridwidth**, and **gridheight** values. Notice there is a lot of repeated code that can be shortened if, for example, all the **JLabels** are laid out together and their constraint settings reused.

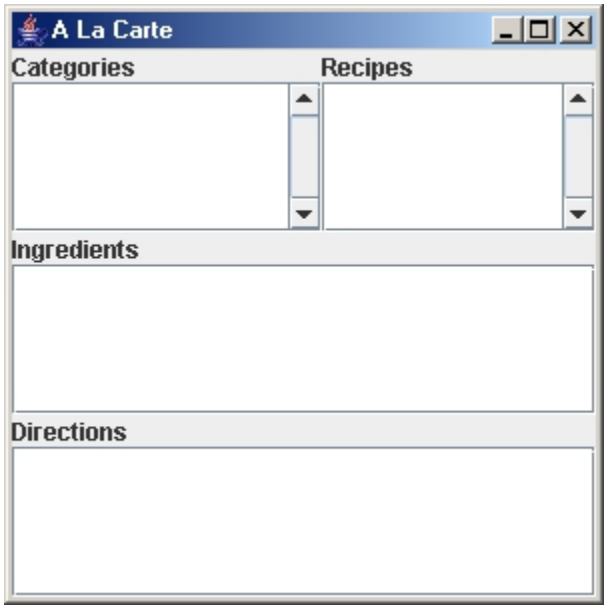
```
import java.util.*;
import java.awt.*;
import javax.swing.*;
public class RecipeBrowser extends JFrame {

     public RecipeBrowser(String name) {
          super(name);

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

          JLabel label = new JLabel("Categories");
          constraints.gridx = 0;
          constraints.gridy = 0;
          constraints.gridwidth = 1;
          constraints.gridheight = 1;
          constraints.weightx = 0; // don't grow horizontally
          constraints.weighty = 0; // don't grow vertically
          constraints.anchor = GridBagConstraints.WEST;
          layout.setConstraints(label, constraints);
          add(label);

          label = new JLabel("Recipes");
          constraints.gridx = 1;
          constraints.gridy = 0;
          constraints.gridwidth = 1;
          constraints.gridheight = 1;
          constraints.weightx = 0; // don't grow horizontally
          constraints.weighty = 0; // don't grow vertically
          layout.setConstraints(label, constraints);
          constraints.anchor = GridBagConstraints.WEST;
          add(label);

          JList categories = new JList();
          categories.setPrototypeCellValue("xxxxxxxxxx");
          JScrollPane scrollPane = new JScrollPane(categories,
              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.weightx = 1;
          constraints.weighty = 1;
          layout.setConstraints(scrollPane, constraints);
          add(scrollPane);

          JList recipes = new JList();
          recipes.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
          scrollPane = new JScrollPane(recipes,
              ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
              ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
          constraints.gridx = 1;
          constraints.gridy = 1;
```

```
            constraints.gridwidth = 1;
            constraints.gridheight = 1;
            constraints.fill = GridBagConstraints.BOTH;
            constraints.weightx = 1;
            constraints.weighty = 1;
            layout.setConstraints(scrollPane, constraints);
            add(scrollPane);

            label = new JLabel("Ingredients");
            constraints.gridx = 0;
            constraints.gridy = 2;
            constraints.gridwidth = 1;
            constraints.gridheight = 1;
            constraints.weightx = 0; // don't grow horizontally
            constraints.weighty = 0; // don't grow vertically
            layout.setConstraints(label, constraints);
            add(label);

            JTextArea ingredients = new JTextArea();
            scrollPane = new JScrollPane(ingredients,
                ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
            constraints.gridx = 0;
            constraints.gridy = 3;
            constraints.gridwidth = 2;
            constraints.gridheight = 1;
            constraints.fill = GridBagConstraints.BOTH;
            constraints.weightx = 1;
            constraints.weighty = 1;
            layout.setConstraints(scrollPane, constraints);
            add(scrollPane);

            label = new JLabel("Directions");
            constraints.gridx = 0;
            constraints.gridy = 4;
            constraints.gridwidth = 1;
            constraints.gridheight = 1;
            constraints.weightx = 0; // don't grow horizontally
            constraints.weighty = 0; // don't grow vertically
            layout.setConstraints(label, constraints);
            add(label);

            JTextArea directions = new JTextArea();
            scrollPane = new JScrollPane(directions,
                ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
            constraints.gridx = 0;
            constraints.gridy = 5;
            constraints.gridwidth = 2;
            constraints.gridheight = 1;
            constraints.fill = GridBagConstraints.BOTH;
            constraints.weightx = 1;
            constraints.weighty = 1;
            layout.setConstraints(scrollPane, constraints);
            add(scrollPane);

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

        public static void main(String[] args) {
            JFrame frame = new RecipeBrowser("A La Carte");
            frame.setVisible(true);
        }
}
```

Here are some interesting tips/points about the code above:

- If you do not want the component to resize when the window is resized (such as the "Categories" and "Recipes" labels), then you can:
  - set the **weightx** and **weighty** to zero.
  - you may still need to specify the **fill** to BOTH if you want to have this component take up its entire cell space...in case some other cell in the same row/column has caused this cell to enlarge.
  - you should anchor the labels (in this case to the left (i.e., WEST) of its grid cell).
- If you are setting the **fill** to **BOTH** for your component, then you do not need to set the **anchor**.
- The **`setPrototypeCellValue()`** method is used to specify a "typical" String that would appear in the **JList**. JAVA will use this string (along with the Font that is set for the list) to figure out how many pixels wide it should make the list.

Here is the result:



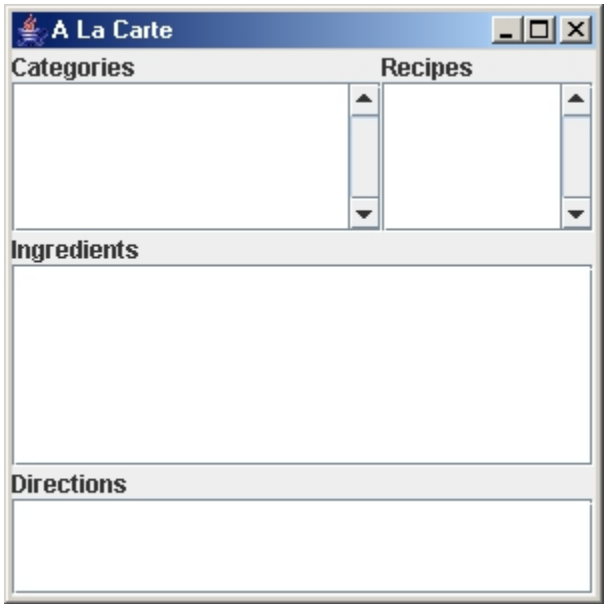
Note that we can adjust how much "space" each component takes according to the weightx and weighty settings. For example, currently we have these settings:

COMPONENT	weightx	weighty
category list	1	1
recipe list	1	1
ingredients text area	1	1
directions text area	1	1

We can change these weights to allow the categories list to be wider than the recipe list and the Ingredients area to be larger (in the y) than the lists and the directions area to be smaller (in the y) than the lists as follows:

COMPONENT	weightx	weighty
category list	2	2
recipe list	1	2
ingredients text area	1	3
directions text area	1	1

We would then get the following look:



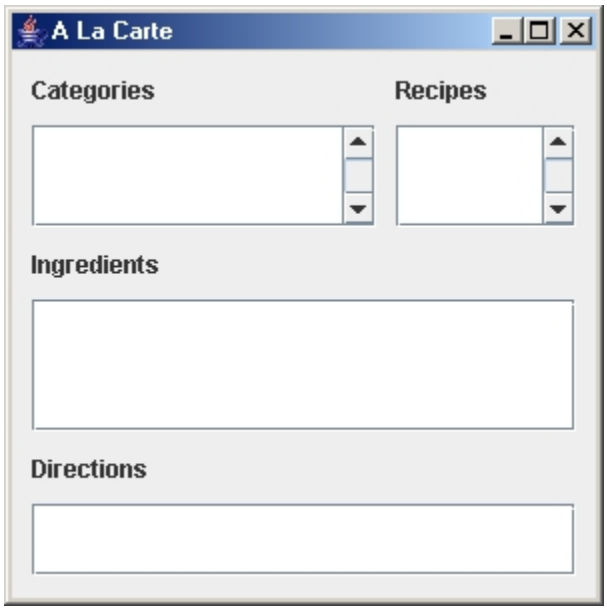
So, by playing with the weights, you can usually achieve the desired look, after some trial and error. Note that we can also make some nice margins around the window and components. We can set the insets for each component by using:

```
constraints.insets(new Insets(top, left, bottom, right));
```

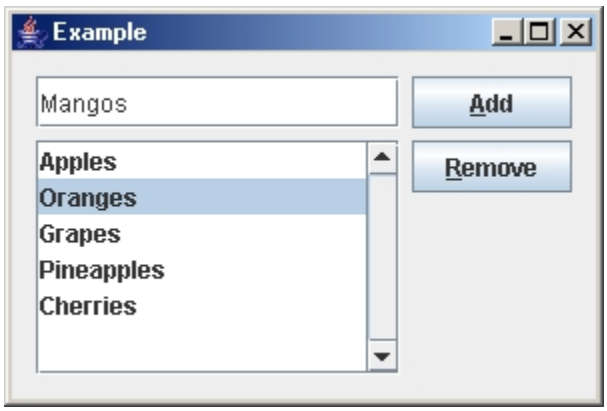
where top, left, bottom and right are set as follows for our components:

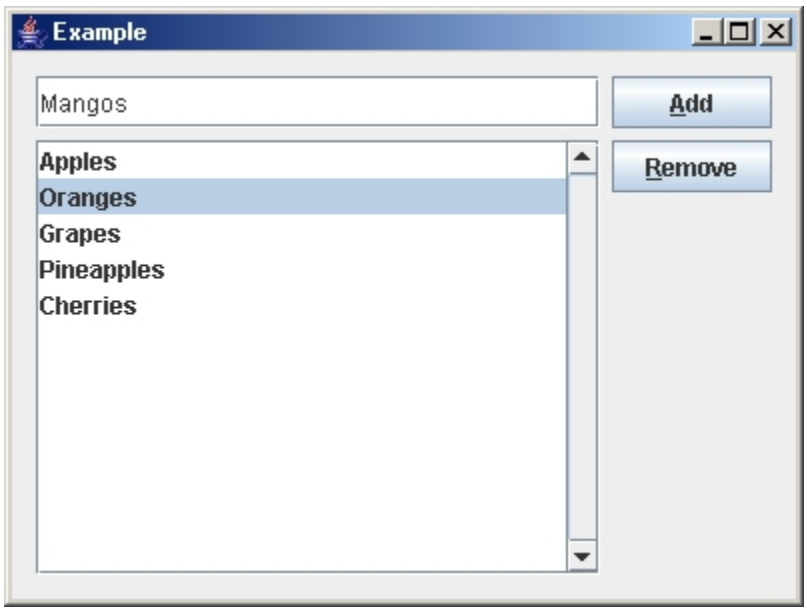
COMPONENT	top	left	bottom	right
categories label	10	10	0	0
recipes label	10	10	0	0
categories list	10	10	0	0
recipes list	10	10	0	10
ingredients label	10	10	0	0
ingredients text area	10	10	0	10
directions label	10	10	0	0
directions text area	10	10	10	10

We obtain this nice look:



Try making the following window ... shown before and after resizing:





For each component, we need to determine how it grows:

- the text field seems to grow only horizontally
- the buttons don't seem to grow
- the buttons and text field have the same height
- the list seems to grow in both directions

- there are margins around the window

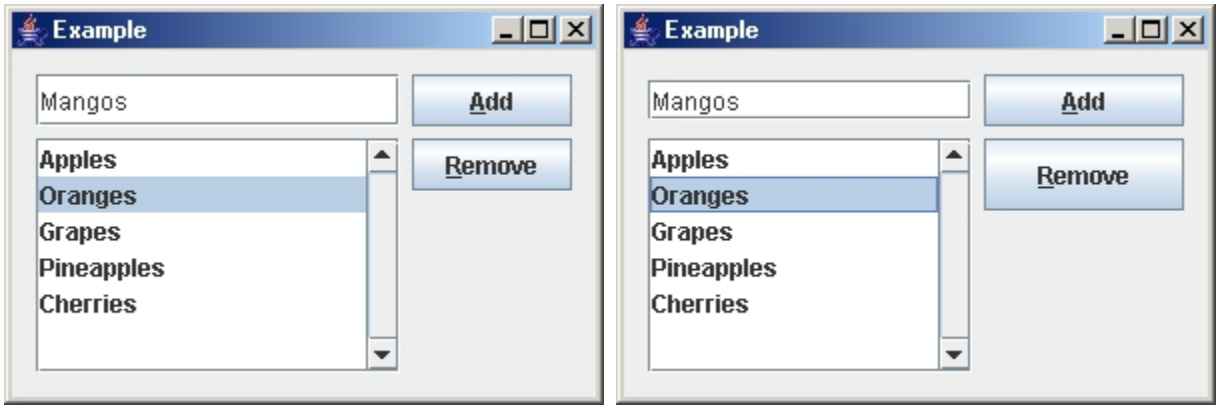
Here is the code:

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

     public GridBagLayoutManagerExample(String name) {
          super(name);

          GridBagLayout layout = new GridBagLayout();
          GridBagConstraints constraints = new GridBagConstraints();
          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);
          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);
          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);
          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);
            add(removeButton);

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

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

It is interesting to see how other components in the same row and column actually affect the other components. For example, assume that the textField had a fill set to HORIZONTAL instead of BOTH. Its height would then be different since it would take up only the height that is needed for the component by default instead of taking up the height of the grid cell that it lies in ... which depends on the height of the add button. Also, for example, the width of the Add button depends on the width of the Remove button since it has a fill HORIZONTAL which takes up the width of the whole cell ... which depends on the width of the Remove button.

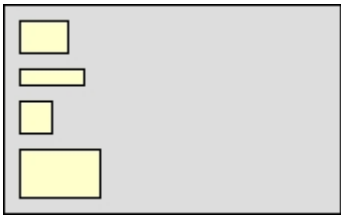
To see this, we will set the fill to HORIZONTAL for the text field and set the internal padding of the Remove button to ipadx = 20; ipady = 10. Here is the result as expected:



Before changes After changes

So you can see ... things can get quite complicated. Make sure to practice a lot with this particular layout manager.

2.3.7 BoxLayout



- components are arranged horizontally from left to right, or vertically from top to bottom.
- much like flow layout, except there is no wraparound when space runs out.
- often used to arrange components in a panel

Here is the one constructor:

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

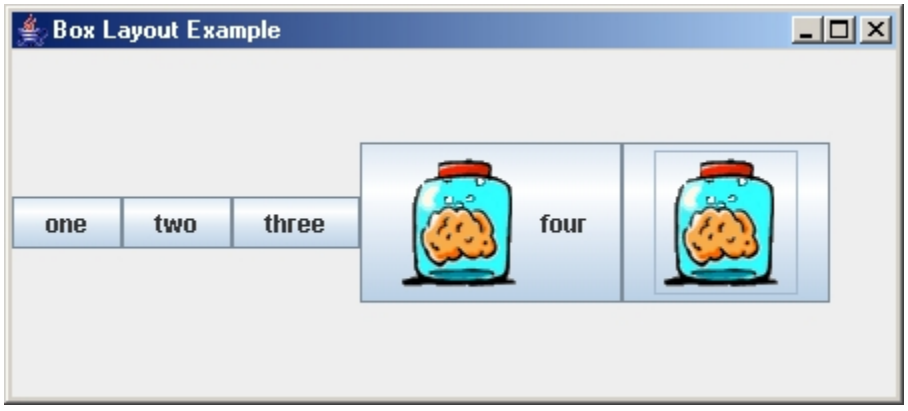
- axis may be `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS` only.

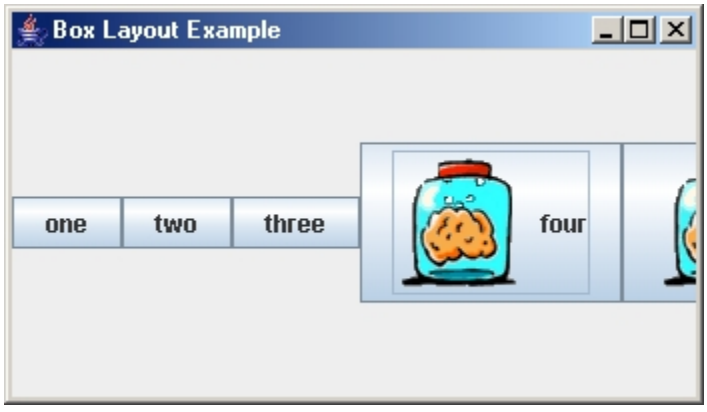
Example

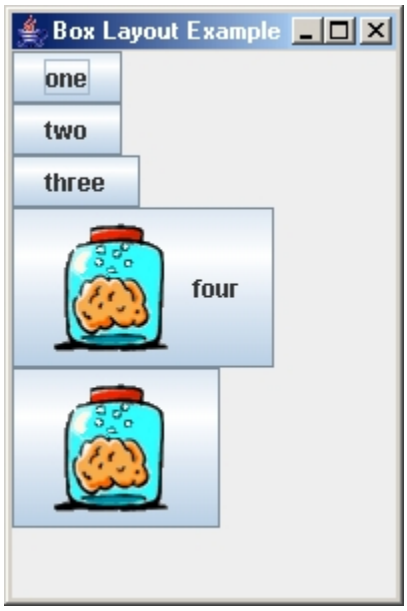
Here is a simple example that adds some components to a panel which uses a **`BoxLayout`**.

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

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

        setLayout(new BoxLayout(this.getContentPane(), BoxLayout.Y_AXIS));

        add(new JButton("one"));
        add(new JButton("two"));
        add(new JButton("three"));
        add(new JButton("four", new ImageIcon("brain.gif")));
        add(new JButton(new ImageIcon("brain.gif")));

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(200, 300);
    }

    public static void main(String args[]) {
        BoxLayoutManagerExample frame =  new BoxLayoutManagerExample("Box
Layout Example");
        frame.setVisible(true);
    }
}
```

Here are three screen snapshots. The first two show the results of the above code which "lines up" the components along the X_AXIS. Notice how the components DO NOT wrap around to the next line when the window is shrunk. The last snapshot shows the arrangement that would be obtained if the Y_AXIS was used.






Box Layout Example
one
two
three
four

3 Events and Listeners

What's in This Set of Notes?

Now that we know how to design the "look" of a window by placing components on it, we need to make the window respond properly to the user interaction. The techniques are based on something called "Event Handling". In JAVA, we handle events by writing "Listeners" (also known as event handlers).

Here are the individual topics found in this set of notes (click on one to go there):

3.1 Events and Event Handlers

In the previous set of notes, we have seen how to create a GUI with various types of components. However, none of the components on the window seem to respond to the user interactions. In order to get the interface to "work" we must make it respond appropriately to all user input such as clicking buttons, typing in text fields, selecting items from list boxes etc... To do this, we must investigate ***events***.

What is an event ?

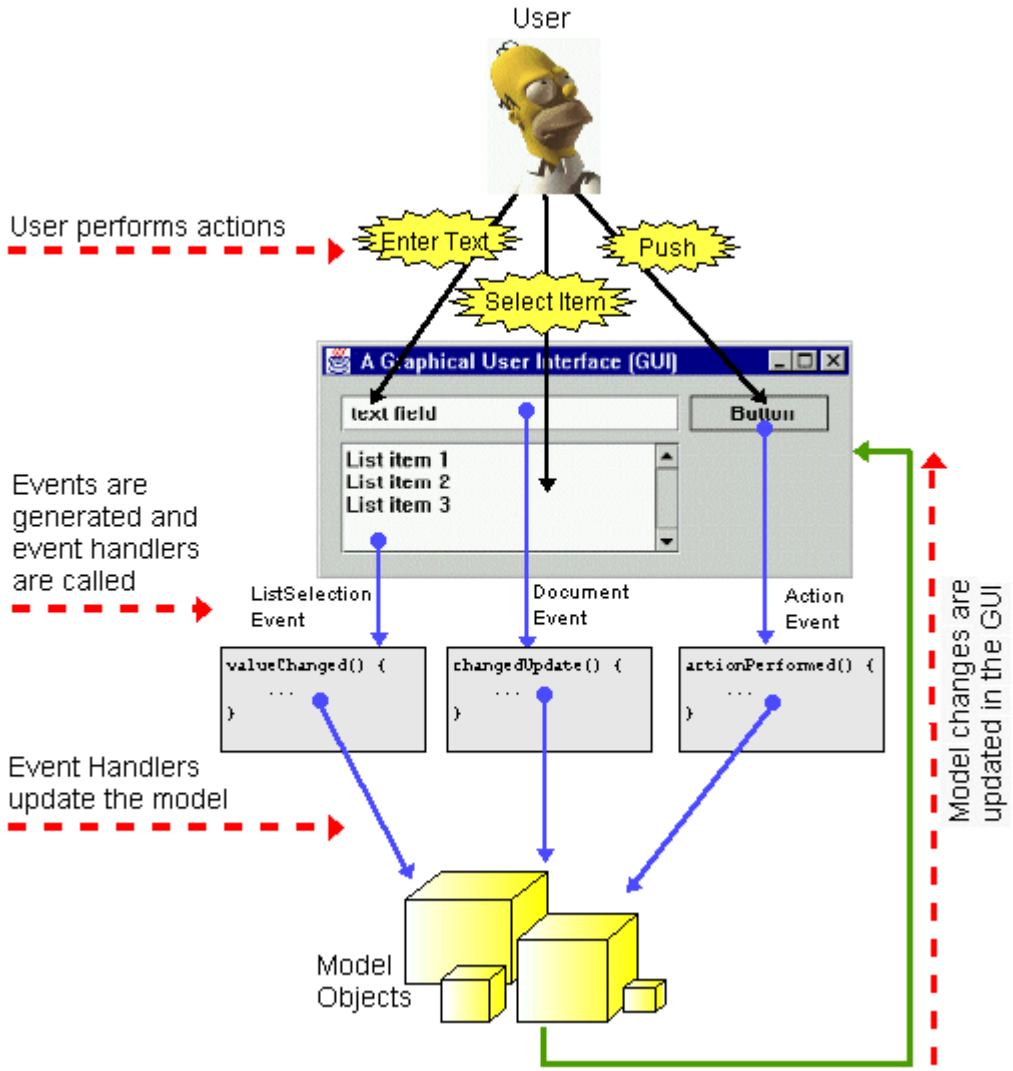
- An ***event*** is something that happens in the program based on some kind of triggering input.
- typically caused (i.e., generated) by user interaction (e.g., mouse press, button press, selecting from a list etc...)
  - the component that caused the event is called the ***source***.
- can also be generated internally by the program

How are Events Used in JAVA ?

- events are objects, so each type of event is represented by a distinct class (similar to the way exceptions are distinct classes)
- low-level events represent window-system occurrences or low-level input such as mouse and key events and component, container, focus, and window events.
- some events may be ignored, some may be ***handled***. We will write ***event handlers*** which are known as ***listeners***.

Nothing happens in your program UNLESS an event occurs. JAVA applications are thus considered to be ***event-driven***.

Here is a picture that describes the process of user interaction with a GUI through events:



Basically...here's how it works:

1. The user causes an event (e.g., click button, enter text, select list item etc...)
2. The JAVA VM invokes (i.e., triggers) the appropriate event handler (if it has been implemented and registered).
    - This invocation really means that a **method is called** to handle the event.
3. The code in the event handling method changes the model in some way.
4. Since the model has changed, the interface will probably also change and so components should be updated.

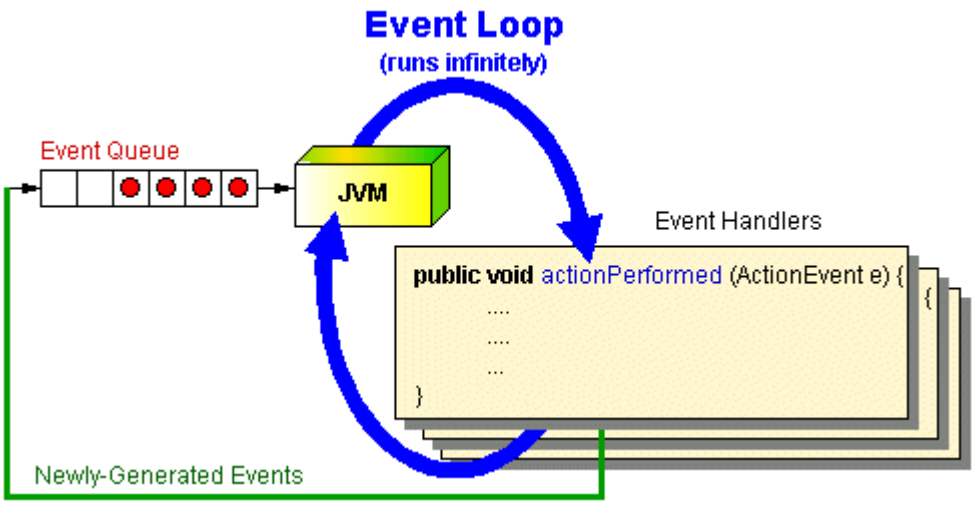
Notice that JAVA itself waits for the user to initiate an action that will generate an event.

- This is similar to the situation of a cashier waiting for customers ... the cashier does nothing unless an event occurs. Here are some events which may occur, along with how they may be handled:
    - a customer arrives - employee wakes up and looks sharp
    - a customer asks a question - employee gives an answer
    - a customer goes to the cash to buy - employee initiates sales procedure
    - time becomes 6:00pm - employee goes home

JAVA acts like this employee who waits for a customer action. JAVA does this by means of something called an ***EventLoop***. An ***Event Loop*** is an endless loop that waits for events to occur:

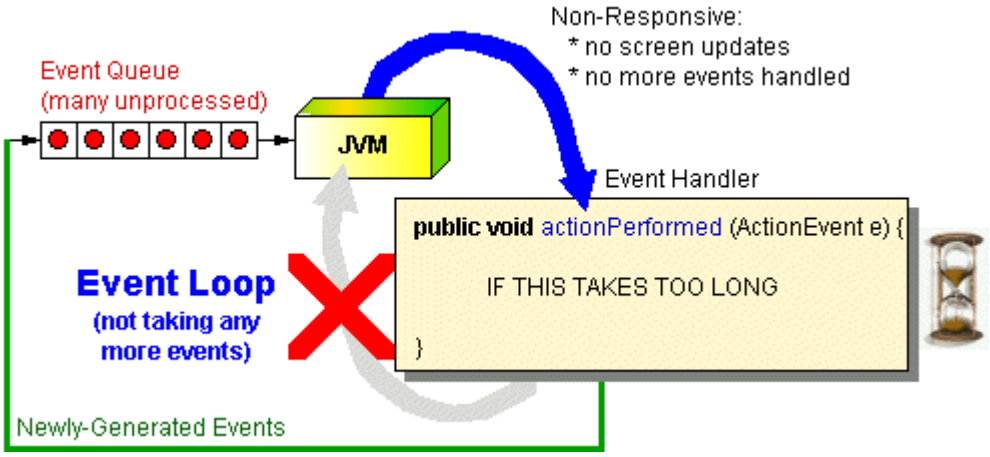
- events are queued (lined up on a first-come-first-served basis) in a buffer
- events are handled one at a time by an ***event handler*** (i.e., code that evaluates when event occurs)
- everything you want done in your application MUST go through this loop

Here is a picture of how the event loop works:



Notice that incoming events (i.e., customers/clients) are stored in the event queue in the order that they arrive. As we will see later, events MUST be handled by your program. The JVM spends all of its time taking an event out of the queue, processing it and then going back to the queue for another.

While each event is being handled, JAVA is unable to process any other events. You MUST be VERY careful to make sure that your event handling code does not take too long. Otherwise the JVM will not take any more events from the queue. This makes your application seem to "hang" so that the screen no longer updates, and all buttons, window components seem to freeze up !!!



In a way, the JVM event loop acts as a *server*. It serves (or handles) the incoming events one at a time on a first-come-first-served basis. So when an event is generated, JAVA needs to go to the appropriate method in your code to handle the event. How does JAVA know which method to call ? We will ***register*** each event-handler so that JAVA can call them when the events are generated. These event-handlers are called ***listeners*** (or ***callbacks***).

A ***listener:***

- acts on (i.e., handle) the event notification.
- must be ***registered*** so that it can be notified about events from a particular source.

- can be an instance of any class (as long as the class implements the appropriate listener interface)

So ... when creating a GUI, we must:

- decide what types of events we want to handle
- inform JAVA which ones we want to handle by *registering* the event handlers (i.e., the listeners)
- write the event handling code for each event

3.2 Listeners and Adapter Classes

You should understand now that when the user interacts with your user interface, some events will be generated automatically by JAVA. There are many types of events that can occur, and we will choose to respond to some of them, while ignoring others. The JAVA VM is what actually generates the events, so we will have to "speak JAVA's language" in order to understand what the event means. In fact, to handle a particular event, we will have to write a particular method with a predefined name (chosen by JAVA).

Here is a list of the commonly used types of events in JAVA:

- **Action Events**: clicking buttons, selecting items from lists etc....
- **Component Events**: changes in the component's size, position, or visibility.
- **Focus Events**: gain or lose the ability to receive keyboard input.
- **Key Events**: key presses; generated only by the component that has the current keyboard focus.
- **Mouse Events**: mouse clicks and the user moving the cursor into or out of the component's drawing area.
- **Mouse Motion Events**: changes in the cursor's position over the component.
- **Container Events**: component has been added to or removed from the container.

Here are a couple of the "less used" types of events in JAVA:

- **Ancestor Events**: containment ancestors is added to or removed from a container, hidden, made visible, or moved.
- **Property Change Events**: part of the component has changed (e.g., color, size,...).

For each event type in JAVA, there are defined interfaces called **Listeners** which we must implement. Each listener interface defines one or more methods that MUST be implemented in order for the event to be handled properly.

There are many types of events that are generated and commonly handled. Here is a table of some of the common events. The table gives a short description of when the events may be generated, gives the interface that must be implemented by you in order for you to handle the events and finally lists the necessary methods that need to be implemented. Note, for a more complete description of these events, listeners and their methods, see the JAVA API specifications.

Event Type	Generated By	Listener Interface	Methods that "YOU" must Write
ActionEvent	a button was pressed, a menu item selected,	ActionListener	actionPerformed(ActionEvent e)

	pressing enter key in a text field or a timer event was generated		
CaretEvent	moving cursor (caret) in a text-related component such as a JTextField	CaretListener	caretUpdate(CaretEvent e)
ChangeEvent	value of a component such as a JSlider has changed	ChangeListener	stateChanged(ChangeEvent e)
DocumentEvent	changes have been made to a text document such as insertion, removal in an editor	DocumentListener	changedUpdate(DocumentEvent e) insertUpdate(DocumentEvent e) removeUpdate(DocumentEvent e)
ItemEvent	caused via a selection or deselection of something from a list, a checkbox or a toggle button	ItemListener	itemStateChanged(ItemEvent e)
ListSelectionEvent	selecting (click or double click) a list item	ListSelectionListener	valueChanged(ListSelectionEvent e)
WindowEvent	open/close, activate/deactivate, iconify/deiconify a window	WindowListener	windowOpened(WindowEvent e) windowClosed(WindowEvent e) windowClosing(WindowEvent e) windowActivated(WindowEvent e) windowDeActivated(WindowEvent e) windowIconified(WindowEvent e) windowDeiconified(WindowEvent e)
FocusEvent	a component has gained or lost focus. Pressing tab key changes focus of components in a window	FocusListener	focusGained(FocusEvent e) focusLost(FocusEvent e)
KeyEvent	pressing and/or releasing a key while within a component	KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
MouseEvent	pressing/releasing /clicking a mouse button, moving a mouse onto or away from a component	MouseListener	mouseClicked(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mousePressed(MouseEvent e) mouseReleased(MouseEvent e)

MouseEvent	moving a mouse within a component while the button is up or down	MouseMotionListener	mouseDragged(MouseEvent e) mouseMoved(MouseEvent e)
ContainerEvent	Adding or removing a component to a container such as a panel	ContainerListener	componentAdded(ContainerEvent e) componentRemoved(ContainerEvent e)

So, 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 write the listener (i.e., event handler) and also register that listener.

To help you understand why we need to do this, think of the Olympic games. There are various events in the Olympics and we may want to participate (i.e., handle) a particular event. Our training and preparation for the event is like writing the event handler code which defines what we do when the event happens. But, we don't get to participate in the Olympic games unless we "sign-up" (or register) for the events ... right ? So registering our event handlers is like joining JAVA's sign-up list so that JAVA informs us when the event happens and then allows our event handler to participate when the event occurs.

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(ActionListener anActionListener);
```

```
aJPanel.addMouseListener(MouseListener aMouseListener);
aJFrame.addWindowListener(WindowListener 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:

```
public class SimpleEventTest extends JFrame implements ActionListener
{

    public SimpleEventTest(String name) {
        super(name);

        JButton aButton = new JButton("Hello");
        add(aButton);

       // Plug-in the button's event
      handler

      aButton.addActionListener(this);


        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(200, 200);
    }

     // 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);
    }
}
```

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

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

Why would you want to disable a listener ? If we don't want to use it, why even make one ? We will see later that we sometimes need to temporarily disable events while other events are being handled so as to avoid overlapping events, which can cause problems.

Adapter Classes:

Assume that we would like to handle a single event ... a **mouseClicked** event whenever someone clicks the mouse inside of our application's window. Recall from COMP1405/1005, that if a class implements an interface it MUST implement ALL of the methods listed in the interface. For example, the **MouseListener** interface is defined as follows:

```
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);
}
```

So, if we simple make our main application implement the **MouseListener** interface, then we will be forced to implement all 5 methods: mouseClicked, mouseEntered, mouseExited, mousePressed and mouseReleased !!!! We can however, merely write empty methods for these other 4 event types but this is a lot of extra code writing that just wastes time and makes the code more confusing:

```
public class MyApplication extends JFrame implements MouseListener {
    ...
    public void mouseClicked(MouseEvent e) { /* Put your code here */
};
    public void mouseEntered(MouseEvent e) {};
    public void mouseExited(MouseEvent e) {};
    public void mousePressed(MouseEvent e) {};
    public void mouseReleased(MouseEvent e {};
    ...
}
```

It does seem a little silly to have to write 4 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. 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 does NOT have an adapter class since it is only one method long.

These adapter classes are **abstract** JAVA classes that implement the interfaces they correspond to. However, even though they implement these interfaces ... their methods remain empty. For example, the **MouseAdapter** class looks like this:

```
public abstract class MouseAdapter implements 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 {};
}
```

They are merely classes that are provided for convenience sake to help us avoid writing empty methods. So, we can simply write subclasses of these adapter classes, then we can take advantage of these blank methods through inheritance.

Well, we don't want to have to make our user interfaces subclass one of these adapter classes ... this would be bad since we would lose the freedom of creating our own arbitrary class hierarchies. Consider handling an event for dealing with a simple mouse click. We could make our own ***internal class*** to handle this event.

```
class MyClickHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
        System.out.println("Do Something fun");
    }
}
```

But this strategy creates an additional .java file. It may make our code more complicated, since we

increase the number of source code files. Well, we can actually write this code within our GUI class itself, provided that we don't put the **public** modifier in front of this class definition. This makes it an internal class. When you compile a .java file that has internal classes, you will notice that you will have additional **.class** files for these additional internal classes (which will have a $ in their name).

To reduce clutter, JAVA allows another way for us to create inner classes which uses VERY strange syntax:

```
new MouseAdapter() {
    public void mouseClicked(WindowEvent event) {
       System.out.println("Do Something fun");
    }
}
```

This syntax actually creates an internal class as a subclass of **MouseAdapter**. 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. We will see below how we can "embed" this code inside other code just like we use any other objects.

Summary of Making Your Own Event Handlers:

Let us now summarize the various ways (i.e., styles) that you can write your event handler code. Here are 4 ways ... you should understand them all:

1. **Make your class implement the specific interfaces needed:**
   - Advantages:
     - Simple
   - Disadvantages:
     - must write methods for ALL events in the interface.
     - can get messy/confusing if your class has many components that trigger the same events or if your class handles many different types of events.

```
public class YourClass extends JFrame implements MouseListener {

    // This line must appear in some method, perhaps the
constructor
    ... {
        aComponent.addMouseListener(this);
    }

    // Some more of your code

    public void mouseClicked(MouseEvent e) { /* Put your code here
*/  };
    public void mouseEntered(MouseEvent e) { /* Put your code here
*/  };
    public void mouseExited(MouseEvent e) { /* Put your code here
*/  };
    public void mousePressed(MouseEvent e) { /* Put your code here
*/  };
    public void mouseReleased(MouseEvent e { /* Put your code here
*/  };

    // Put your other methods here
}
```

2. **Create a separate class that implements the interface:**

- Advantages:
  - nice separation between your code and the event handlers.
  - class can be reused by other classes
- Disadvantages:
  - can end up with a lot of classes and class files
  - can be confusing as to which classes are just event handler classes

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the
constructor
    ... {
        aComponent.addActionListener(new MyButtonListener(this));
    }

    // Some more of your code
}

public class MyButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent theEvent) {
        // Do what needs to be done when the button is clicked
    }
}
```

3. **Create an "inner" class that implements the interface:**
   - Advantages:
     - nice separation between your code and the event handlers.
     - class can be reused in different situations within your class
     - Inner class has access to the "guts" of your class
   - Disadvantages:
     - can still end up with a lot of class names to remember

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the
constructor
    ... {
        aComponent.addActionListener(new MyButonListener());
    }

    // Some more of your code

    class MyButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent theEvent) {
            // Do what needs to be done when the button is clicked
        }
    }
}
```

4. **Create an anonymous subclass of an Adapter class or a Listener interface.**
   - Advantages:
     - nice and compact
     - do not need to come up with class names, reduces complexity
     - only need to handle one event instead of worrying about all events in the interface.
   - Disadvantages:
     - the syntax takes a little "getting use to"
     - requires event handler code to be specified where listener is registered (unless helper methods are used)

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the
constructor
    ... {
        aComponent.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent theEvent) {
                   // Do what needs to be done when the button is
clicked
                }
            }
        );
    }

    // Some more of your code
}
```

3.3 Handling ActionEvents with ActionListeners

In this section, we give various examples showing how to handle one or more **ActionEvents** from different kinds of objects:

Handling two button clicks

We have already seen how to handle a simple button press by writing an **ActionPerformed** method. Here is an application that shows how to handle events for two different buttons. We will make use of the **getActionCommand()** method for the **ActionEvent** class that allows us to determine the label on the button that generated the event. Take notice of the packages that need to be imported.


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Handle2Buttons extends JFrame implements ActionListener {
    public Handle2Buttons(String title)
        super(title);

        JButton aButton1 = new JButton("Press Me");
        JButton aButton2 = new JButton("Don't Press Me");

        setLayout(new FlowLayout());
        add(aButton1);
        add(aButton2);

        // Indicate that this class will handle 2 button clicks
        // and that both buttons will go to the SAME event handler
        aButton1.addActionListener(this);
        aButton2.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,100);
    }

    // This is the event handler for the buttons
```

```
    public void actionPerformed(ActionEvent e) {
        // Ask the event which button was the source that generated it
        if (e.getActionCommand().equals("Press Me"))
            System.out.println("That felt good!");
        else
            System.out.println("Ouch! Stop that!");
    }

    public static void main(String args[]) {
        Handle2Buttons frame =  new Handle2Buttons("Handling 2 Button Presses");
        frame.setVisible(true);
    }
}
```

Notice that the **getActionCommand()** method is sent to the **ActionEvent**. It returns a **String** containing the text that is on the button that generated the event. We then compare this string with the labels that we put on the buttons to determine which button was pressed. One disadvantage of this approach is that our event handler depends on the label associated with the button. Although this is safe in this particular example, there are many occasions when the label associated with a button could change (e.g., international applications). Therefore, we could use the **getSource()** method which returns the component (i.e., an Object) that raised the event instead of **getActionCommand()** to compare the actual button objects instead of the labels. To do this, we need to make two modifications. First, we need to store the buttons we create into instance variables and second, we need to compare the object that generated the event with these buttons using the identity (==) comparison.

```
    // We need to make the buttons instance variables and assign
    // them in the constructor so that we can access these objects
    // from within our event handler code.
    JButton   aButton1, aButton2;

    // Change the event handler to use getSource() to compare the actual objects
    public void actionPerformed(ActionEvent e) {
        // Ask the event which button was the source that generated the event
        if (e.getSource() == aButton1)
            System.out.println("That felt good!");
        else
            System.out.println("Ouch! Stop that!");
    }
```

Another disadvantage of the previous example is that if more buttons (or other components that generate action events) are added, the number of "if-statements" in our handler will increase and become more complex, which may not be desirable. One way to handle this disadvantage (and the previous one as well) is by using anonymous classes. The following code would replace the code in the constructor that registers our frame subclass as a listener. The **actionPerformed** method of our class would no longer be required. Here, each button has its own event handler:

```
    aButton1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("That felt good!");
        }
    });
    aButton2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Ouch! Stop that!");
        }
    });
```

A Simple Slide Show

Now let us make a more interesting example that uses anonymous classes for two buttons. We will create a simple Slide Show application. We will create a window that has a **JPanel** which uses a **CardLayout** to represent the slides (one at a time) and then we will also add two arrow buttons to the window to rewind and forward the slides. Notice the following:

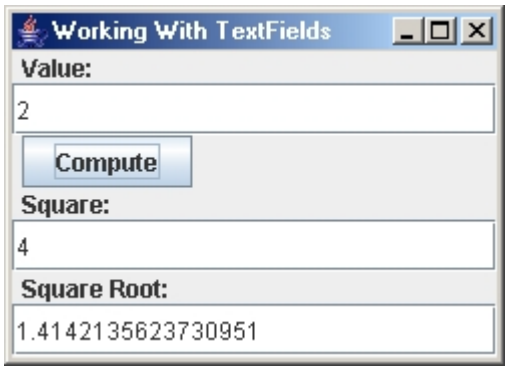
- the buttons we will use are the standard **BasicArrowButton** objects that are available in JAVA in the **`javax.swing.plaf.basic`** package.
- the **JPanel** and **CardLayout** are made into instance variables so that we can access them from our event handlers
- the main window is set to use a **FlowLayout**, the default was **BorderLayout**.
- we used **BorderFactory.createLineBorder()** to make a nice black border around our panel.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.BasicArrowButton;
public class SlideShow extends JFrame {

    JPanel      slides;
    CardLayout  layoutManager;

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

        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("trilobot.jpg", new JLabel(new ImageIcon("trilobot.jpg")));
        slides.add("laptop.jpg", new JLabel(new ImageIcon("laptop.jpg")));
        slides.add("satelite.jpg", new JLabel(new ImageIcon("satelite.jpg")));
        slides.add("torch7.gif", new JLabel(new ImageIcon("torch7.gif")));
        slides.add("SIGNIN.jpg", new JLabel(new ImageIcon("SIGNIN.jpg")));
        add(slides);

        // Now add some slide show buttons for forward and reverse
        JButton rev = new BasicArrowButton(JButton.WEST);
        add(rev);
        JButton fwd = new BasicArrowButton(JButton.EAST);
        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(220,300);
      }

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

Working With JTextFields

Here is a new application that has a button and some text fields. One text field will hold an integer. When the button is pressed, it will compute and display (in two other text fields) the "square" as well as the "square root" of the value within the first text field. Note a few things about the code:



- When creating JTextFields, we can specify the initial content to be displayed (a string) as well as the maximum number of characters allowed to be entered in them (8, 16 and 20 in this example).
- We need to convert to and from Strings when accessing/modifying text field data
- We access/modify a text field's contents using **getText()** and **setText()**
- The code below will generate exceptions if a valid integer is not entered within the value text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HandleTextFieldAndButton extends JFrame {
    JTextField valueField, squareField, rootField;

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

        setLayout(new BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));

        // Add the value text field, along with a label
        add(new JLabel("Value:"));
        valueField = new JTextField("10", 8);
        add(valueField);

        // Add the compute button
        JButton aButton = new JButton("Compute");
        add(aButton);

        // Add the square text field, along with a label
        add(new JLabel("Square:"));
        squareField = new JTextField("0", 16);
        add(squareField);
```

```
        // Add the square root text field, along with a label
        add(new JLabel("Square Root:"));
        rootField = new JTextField("0", 20);
        add(rootField);

        // Handle the button click
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int value = Integer.parseInt(valueField.getText());
                squareField.setText("" + value * value);
                rootField.setText("" + Math.sqrt(value));
            }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,180);
    }

    public static void main(String args[]) {
        HandleTextFieldAndButton frame = new HandleTextFieldAndButton("Working
With TextFields");
        frame.setVisible(true);
    }
}
```

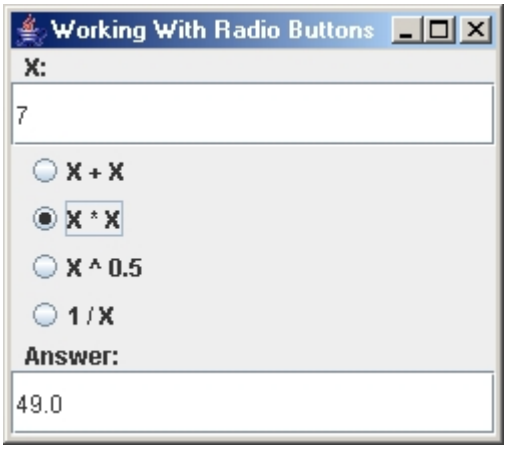
Notes:

Although not done in our example here, we can actually handle an **ActionEvent** for a text field. An action event is generated when the user presses the ENTER key while typing in a text field. As with button clicks, you can handle this ENTER key press in the text field by writing an **actionPerformed()** method for the text field. In such a method, the **getActionCommand()** method will return the text inside the text field. We can also send the g**etSource()** method to the action event to get the text field itself and then get its text as follows:
`((JTextField)e.getSource()).getText()`

As we will discuss later, the style of coding that we are using in our example here is not "clean" since the button accesses the text field directly.

Handling RadioButtons

Let us modify the previous example by using radio buttons that allow us to decide which kind of operation we will do on the value entered in the text field. We will replace the **Compute** button with 4 radio buttons where each radio button, when clicked, will perform a different operation on the value from the text field and then display the result in the answer text field. Here are some interesting points about the code:

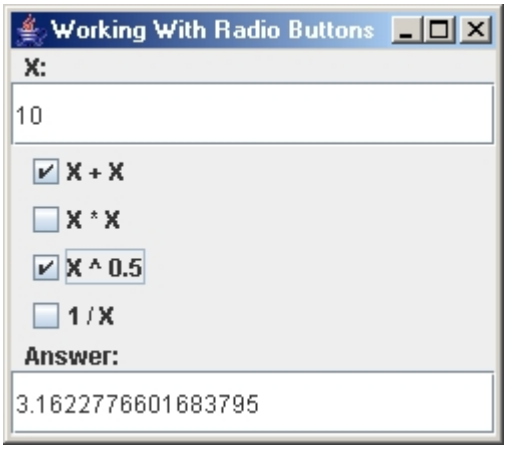


- All **JRadioButtons** go to the same event handler.
- The **JRadioButtons** are stored in an array, which is searched using a FOR loop to determine which one generated the event so that we could perform the desired operation.
- The **JRadioButtons** are added to a **ButtonGroup** as well, which ensures that JAVA allows only one to be "on" at a time. When created, we can specify with a boolean whether a particular button is to be "on" upon window startup.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HandleTextFieldWithRadioButtons extends JFrame implements
ActionListener {

    JTextField            valueField, answerField;
    JRadioButton[]        buttons;

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

        setLayout(new BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));

        // Add the value text field, along with a label
        add(new JLabel("X:"));
        add(valueField = new JTextField("10", 8));

        // Add the operation type radio buttons to the window and
        // also to a ButtonGroup so that one is on at a time
        ButtonGroup operations = new ButtonGroup();
        buttons = new JRadioButton[4];
        buttons[0] = new JRadioButton("X + X", false);
        buttons[1] = new JRadioButton("X * X", false);
        buttons[2] = new JRadioButton("X ^ 0.5", false);
        buttons[3] = new JRadioButton("1 / X", false);
        for (int i=0; i<4; i++) {
            add(buttons[i]);
            operations.add(buttons[i]);
            buttons[i].addActionListener(this);
        }

        // Add the answer text field, along with a label
        add(new JLabel("Answer:"));
        add(answerField = new JTextField("0", 16));

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,220);
    }

    // Handle a radio button click
    public void actionPerformed(ActionEvent e) {
        int value = Integer.parseInt(valueField.getText());
        int buttonNumber = 0;
        for (buttonNumber=0; buttonNumber<4; buttonNumber++) {
            if (buttons[buttonNumber] == e.getSource())
                break;
        }
        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;
        }
        answerField.setText("" + result);
    }

    public static void main(String args[]) {
        JFrame frame = new HandleTextFieldWithRadioButtons("Working With Radio
Buttons");
        frame.setVisible(true);
    }
}
```

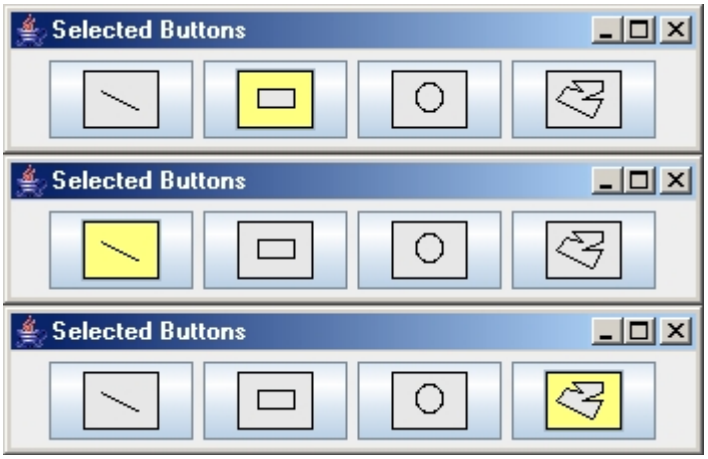
Note that for this example it seems appropriate to have our **JFrame** class act as a listener since the event handling code is the same for all components. Nothing is gained by using another class to handle the events.

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.



Handling JButton Selections

In addition to radio buttons and checkbox buttons, **JButtons** themselves can maintain selected states. For example, we can create an application that allows one of several **JButtons** to be selected. As it turns out, **JButtons** have a **setSelectedIcon()** method that allows us to change the picture on a button when it is selected. Here is an example that makes 4 **JButtons** with icons on them which may be used to select a shape for drawing. When the user selects one of these buttons, the image changes on the button and so the button appears selected. All we have to do is use the **setSelected()** and **getSelected()** methods to change or query the button's state.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SelectedButtonsExample extends JFrame implements ActionListener {
    private JButton[]  buttons;

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

        setLayout(new FlowLayout());

        ButtonGroup group = new ButtonGroup();
        buttons = new JButton[4];
        for (int i=0; i<4; i++)  {
            buttons[i] = new JButton(new ImageIcon("button" + (i+1) + ".gif"));
            buttons[i].setSelectedIcon(new ImageIcon("button" + (i+1) +
```

```
"b.gif"));
            buttons[i].setRolloverEnabled(false);
            buttons[i].addActionListener(this);
            add(buttons[i]);
            group.add(buttons[i]);
        }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(350,75);
    }

    public void actionPerformed(ActionEvent e)  {
        for (int i=0; i<4; i++)
            buttons[i].setSelected(e.getSource() == buttons[i]);
    }

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

Notice that we did not use a **ButtonGroup** to ensure that only one button is selected by itself. That is because **JButtons** do not work with **ButtonGroups**. Instead, we had to do everything manually. If we wanted to allow multiple buttons on at a time, we would merely change the action performed method to be:

```
public void actionPerformed(ActionEvent e) {
    JButton src = (JButton)e.getSource();
    src.setSelected(!src.isSelected());
}
```

This would allow the buttons to be toggled on and off individually. Notice one good thing, we don't have to worry about changing the icon. It is done automatically for us when we set the button to be selected or not.

Notice that we did a **setRolloverEnabled(false)** for our buttons. This is because JAVA, by default, has a default rollover enabled value of **true** for the buttons, which redraws our buttons whenever the mouse passes over them. In this case, a completely separate icon may be used. So, we can modify our code to have our image show when the mouse rolls over the button instead of when we click it by setting this icon and enabling the rollover effect.

We would need to use these methods to accomplish this: **setRolloverIcon()** and **setRolloverSelectedIcon()**.

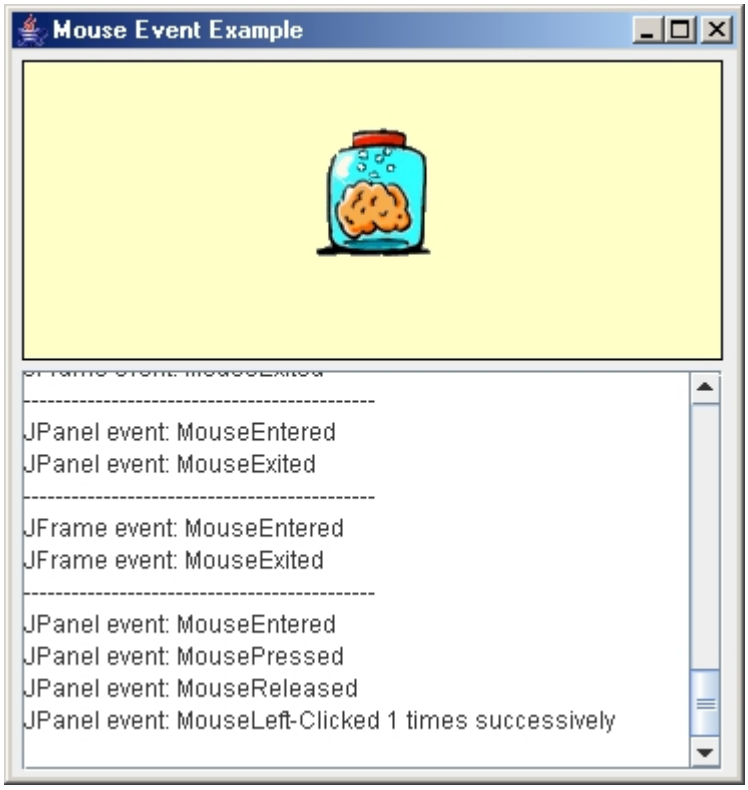
3.4 Handling MouseEvents with MouseListeners

In this section, we talk about mouse events. Mouse events are typically used in graphics applications (as we'll see later in a graph editing application). There are many kinds of mouse events as we have shown in an earlier table:

- **`mouseClicked`** - one of the mouse buttons has been both pressed and released within the same component.
- **`mouseEntered`** - the mouse cursor has entered the component's area.
- **`mouseExited`** - the mouse cursor has left the component's area.
- **`mousePressed`** - a mouse button has been pressed.
- **`mouseReleased`** - a mouse button has been released.

Example using Mouse Listeners

In this example we create two components: a yellow **JPanel** and a white **JTextArea** within a **JScrollPane**. Both the **JFrame** itself as well as the **JPanel** will respond to all **MouseEvents** and display an appropriate message within the text area. We will also add an **ImageIcon** to the JPanel as a **JLabel** and we will move it around depending on where the user presses the mouse. We make use of the following **MouseEvent** methods to get information about mouse presses:



- **`getClickCount()`** - returns the number of successive mouse clicks
- **`getButton()`** - returns the button that was pressed (1 = left, 2 = middle, 3 = right)
- **`getX()`** - returns the x coordinate of the mouse location w.r.t. top left corner of component.
- **`getY()`** - returns the x coordinate of the mouse location w.r.t. top left corner of component.
- **`getPoint()`** - returns the (x,y) point of the mouse location w.r.t. top left corner of component.

Here is the code for the application:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class WorkingWithMouseEvents extends JFrame implements
MouseListener {
     JPanel     blankArea;
     JTextArea  textArea;
     JLabel     movableImage;
     Class      latestComponent;

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

        setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));

        // Create a yellow JPanel
        blankArea = new JPanel();
        blankArea.setLayout(null);
        blankArea.setBackground(new Color(255,255,200));
```

```
        blankArea.setOpaque(true);

blankArea.setBorder(BorderFactory.createLineBorder(Color.black));
        blankArea.setPreferredSize(new Dimension(350, 150));
        add(blankArea);

        // Add an image to the JPanel
        blankArea.add(movableImage = new JLabel(new
ImageIcon("brain.gif")));
        movableImage.setSize(80,80);
        movableImage.setLocation(100,100);

        // Create a text area to display event information
        textArea = new JTextArea();
        textArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(textArea);
        scrollPane.setPreferredSize(new Dimension(350, 200));
        add(scrollPane);

        //Register for mouse events on the JPanel AND the JFrame
        blankArea.addMouseListener(this);
        addMouseListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(370,390);
    }

    // Handle the mouse events (pressed, released, entered, exited &
clicked)
    public void mousePressed(MouseEvent event) {
        addToTextArea("MousePressed", event);
        if (event.getComponent().getClass() != this.getClass())
            movableImage.setLocation(event.getX()-40,
event.getY()-40);
    }

    public void mouseReleased(MouseEvent event) {
        addToTextArea("MouseReleased", event);
    }

    public void mouseEntered(MouseEvent event) {
        addToTextArea("MouseEntered", event);
    }

    public void mouseExited(MouseEvent event) {
        addToTextArea("MouseExited", event);
    }

    public void mouseClicked(MouseEvent event) {
        String s;
        if (event.getButton() == 3)
            s = "Right";
        else s = "Left"; // Ignores the middle button case
        addToTextArea("Mouse" + s + "-Clicked " +
event.getClickCount() +
                          " times successively ", event);
    }

    // Append the specified event-specific text to the text area
    private void addToTextArea(String eventDescription, MouseEvent
event) {
        if (latestComponent != event.getComponent().getClass())

textArea.append("-----------------------------------------\n");
        latestComponent = event.getComponent().getClass();
        if (latestComponent == this.getClass())
            textArea.append("JFrame event: ");
```

```
            else
                textArea.append("JPanel event: ");
            textArea.append(eventDescription + "\n");
        }

        public static void main(String args[]) {
            JFrame frame = new WorkingWithMouseEvents("Mouse Event
 Example");
            frame.setVisible(true);
        }
 }
```

How can we modify the code to only handle clicked events if it was a double-click ?

```
public void mouseClicked(MouseEvent event) {
    if (event.getClickCount() == 2)
       // do something
}
```

3.5 Key Press Events

Every Component in JAVA can listen for ***KeyEvents***. **KeyEvents** are generated when the user presses, releases or types a key while in a component. In order for the event to be generated, the component MUST have the focus. The ***focus*** represents the current component that is selected (e.g., we all understand how the TAB key moves the focus from one component to another in many windows applications). Thus, if a particular component is listening for a key press, but that component does not have the focus, then no events are generated.

We can have a component listen for **KeyEvents** by adding a **KeyListener** with the **addKeyListener()** method. Inside these listeners, we can determine which key was pressed by examining the **KeyEvent** object itself. The **KeyEvent** class has a bunch of static constants that represent all the keys on the keyboard. These constants all begin with **VK_** and you can look in the JAVA API to get the exact names. Here are a few:

- **VK_A**, **VK_B**, **VK_C**, ..., **VK_Z**
- **VK_SHIFT**, **VK_ALT**, **VK_CONTROL**, **VK_ENTER**
- **VK_DOWN**, **VK_UP**, **VK_LEFT**, **VK_RIGHT**
- etc...

We send the **getKeyCode()** message to the **KeyEvent** to get back the code representing the key that was pressed. We then compare the code to one of these constants. Since every key press generates an event, if we want to detect multiple keys pressed at the same time, me must make use of both **keyPressed()** and **keyReleased()** listeners and keep track by ourselves as to which key has been pressed. There is also a **keyTyped()** event which can detect the entering of a Unicode character. ~~Often a **keyTyped()** event is synonymous with a key press/release sequence (kinda like a **mouseClicked** event).~~

The **keyTyped()** event is generated along with a **keyPressed()** event, whenever letter/number /symbol keys are pressed. However, the **keyTyped()** event is not generated when the control-related keys are pressed (e.g., shift, alt, ctrl, caps-lock, insert, home, end, pageup/down, break, arrow keys, function keys etc...) In this case, just the **keyPressed()** and **keyReleased()** events are generated. Odly enough, some control-related keys do generate **keyTyped()** events (e.g., esc, del), some keys generate only a **keyReleased()** event (e.g., print screen) and some keys do not generate events at all (e.g., tab, function key)!!!!! Check to make sure that the key you want to handle

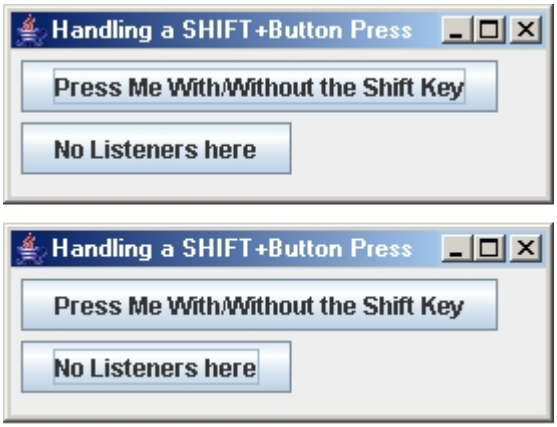
behaves the way you want it to.

Also, we can combine other listeners with key presses ... for example, if we want to detect a SHIFT-CLICK operation.

Here is an example with code that detects three things:

- Pressing the 'A' key by itself
- Pressing the 'SHIFT' and 'A' keys together
- Pressing the 'SHIFT' key and pressing a button together

Note that there are two buttons. The bottom one is not hooked up to the listeners so when it has the focus, no key events are generated. The image on the top (below) shows the first button having the focus (notice the thin gray line around the text of the button). The bottom image shows the non-listener button with the focus.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ShiftButtonTest extends JFrame implements ActionListener,
KeyListener {
    private boolean    shiftPressed;

    public ShiftButtonTest(String title) {
        super(title);
        setLayout(new FlowLayout(5));

        JButton aButton = new JButton("Press Me With/Without the Shift Key");
        JButton bButton = new JButton("No Listeners here");
        add(aButton);
        add(bButton);

        shiftPressed = false;

        //Indicate that this class will handle the button click
        aButton.addActionListener(this);
        aButton.addKeyListener(this); // Change aButton to this if you want to
ignore focus

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,100);
    }

    //This is the event handler for the button
    public void actionPerformed(ActionEvent e) {
        if (shiftPressed)
            System.out.println("You SHIFTED Me!!");
        else
            System.out.println("You did not SHIFT Me.");
```

```
    }

    public void keyPressed(KeyEvent e)  {
        if (e.getKeyCode() == KeyEvent.VK_SHIFT)
            shiftPressed = true;
        else if (e.getKeyCode() == KeyEvent.VK_A)   {
                if (shiftPressed)
                        System.out.println("You pressed the [SHIFT]+[A] keys");
                else
                        System.out.println("You pressed the [A] key");
        }
    }
    public void keyReleased(KeyEvent e)  {
        if (e.getKeyCode() == KeyEvent.VK_SHIFT)
            shiftPressed = false;
    }
    public void keyTyped(KeyEvent e)  {
        // Get and display the character for each key typed
       System.out.println("Key Typed: " + e.getKeyChar());
    }

    public static void main(String args[]) {
        ShiftButtonTest frame = new ShiftButtonTest("Example: Handling a
SHIFT+Button Press");
        frame.setVisible(true);
    }
}
```

If you do not want this "focus-oriented" behavior (e.g., perhaps you want to listen for a particular key press regardless of which component has been selected) you can have the **JFrame** listen for the key press. In this case, you must "disable" the focus ability for all the components on the window (but not the JFrame itself). In our example, we would replace the line:

```
aButton.addKeyListener(this);
```

with the following lines that disallow the buttons to have the focus:

```
this.addKeyListener(this);
aButton.setFocusable(false);
bButton.setFocusable(false);
```

Be aware however, that this disables the "normal" behavior for the window and will prevent standard use of the TAB key to traverse between components in the window.

3.6 Proper Coding Style for Component Interaction

Recall that before designing an application, we must distinguish between the **model** and the **interface**.

Recall as well that the ***model*** is:

- the underlying "meat" of the application (represents "business/problem domain" logic)
- corresponds to all classes and objects that do not deal with the user interface appearance or operation.

The ***GUI (Graphical User Interface)*** is:

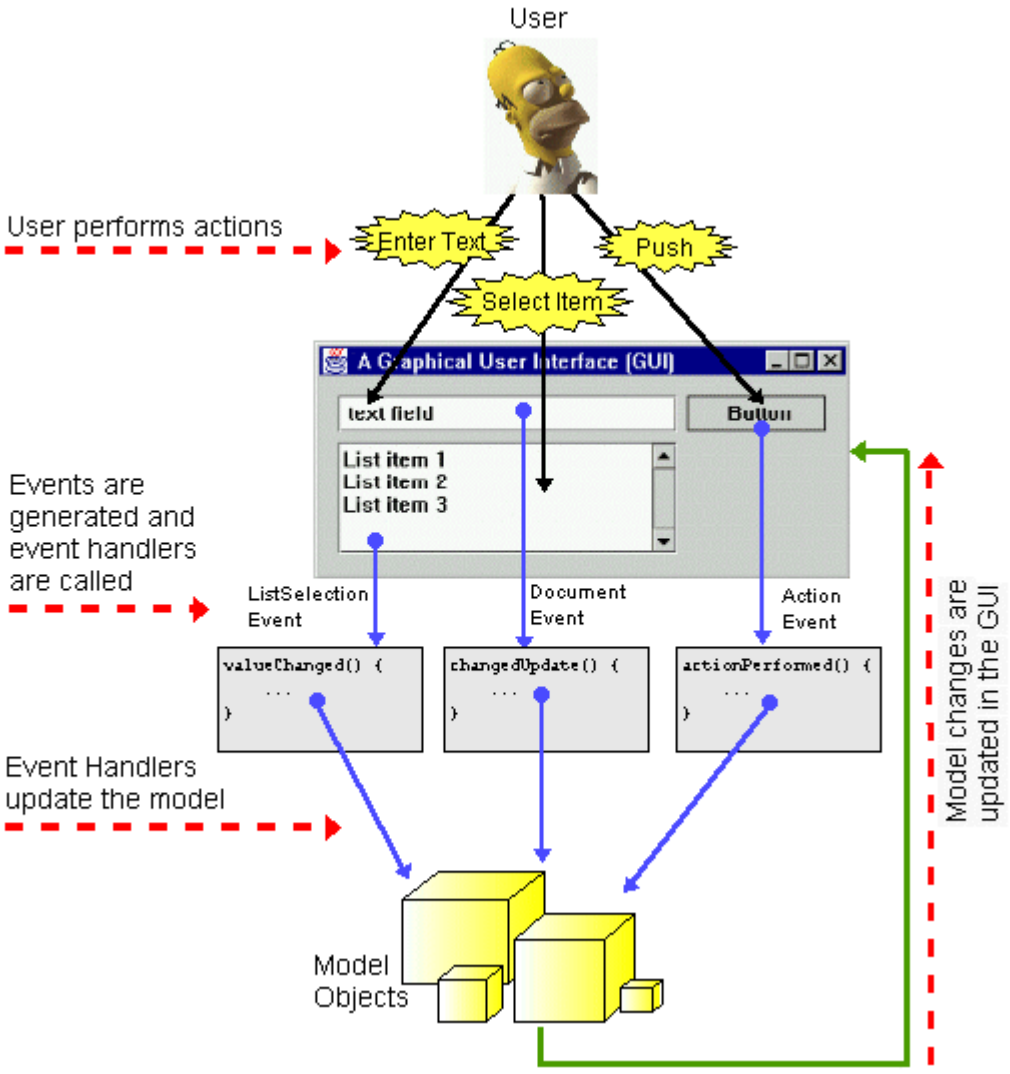
- the portion of your code that deals with the appearance of the application and the interaction

between the interface components.

It is IMPORTANT to keep the model separate from the interface. Also, with respect to the GUI, we need to have a "good" understanding of how the components of the interface will work and interact with each other. Let us see if we can explain how the components interact. We need to determine ALL of the following:

1. What events do we need to handle for each component ?
2. What should happen when each event is triggered ?
3. How do the events affect the model ?
4. How do we make changes to the model ?
5. How do these model changes affect the appearance of the interface ?

Recall that a user interface works as follows (based on what is called the *Observer Pattern*).



There are two questions regarding the updating stage of the components:

- When do we do this updating ? (i.e., where in our code)
- Which components need updating ?

Well, as a rule of thumb, we should do an update whenever we make any change to the model. When we go to do the update, if we know our code very well, we can determine which components will need updating based on the particular change to the model. However, in case we have a complex application, we may not know which ones need updating, so we can take the simple brainless approach and just update everything. We often simply write an **update()** method, where

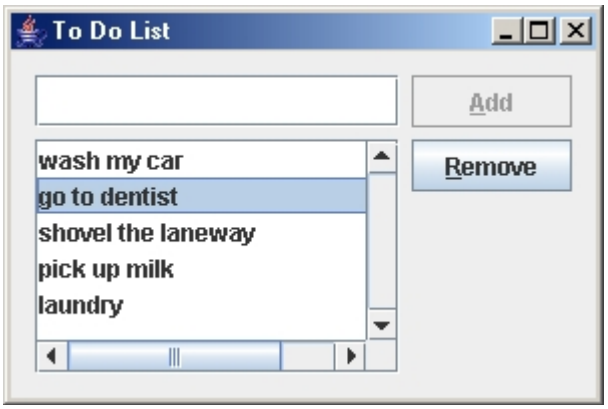
we update all the components (i.e., a kind of ***global*** update is performed). We call this method whenever the model changes.

So, REMEMBER the two VERY important things that you should normally do in every event handler:

1. **Change the model**
2. **Call** update()

Example:

Let us now build the following application which represents a list of "things to do":



The application will work as follows:

- The model consists of a collection (stored in a list) of items. In our example, the items will be Strings.
- The user can add new items to the list by typing the new item in the text field and clicking the **Add** Button.
- Items are removed from the list by selecting an item from the list and clicking the **Remove** Button.

In this "to do list" application, the **model** is simply the collection (e.g., a Vector) of things to do (i.e. Strings). So we don't need to make a model class in this simple example. What about the GUI ? First, we determine how the interface should ***react*** to user input. We must determine which events are necessary to be handled. The events and their consequences are as follows:

AddButton
`actionPerformed()` - Should take text from text field and add it to the list.

RemoveButton
`actionPerformed()` - Should determine selected item from the list and then remove it.

TextField
Nothing for now. We will add some behavior here later.

List
Nothing for now. We will add some behavior here later.

Now what about the model updating ? How do these events change the model ? How does the model then change the window again ?

- **Adding** an item should cause a new entry to be added to the list (i.e., model). Then we must

show these changes in the list.

- **Removing** an item also changes the model and we should show the changes right away in the list as well.

Refreshing the user interface to show the changes in the model is called ***updating.***

Let us now look at a basic "working" application. We will handle the Adding and Removing of items from the list. The highlighted code below indicates the code required for handling the events from the buttons and updating the interface:

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class  TodoListFrame extends JFrame {
    private JTextField newItemField;
    private JList      itemsList;
    private JButton    addButton;
    private JButton    removeButton;

    private Vector<String>  items;  // The model

    public TodoListFrame() {
        this(new Vector<String>());
    }
    public TodoListFrame(Vector<String> todoEntries) {
        super("To Do List");
        items = todoEntries;

        // ...
        // The code for building the window has been omitted
        // ...

       // Add listeners for the buttons and then enable them
       addButton.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
               addButtonEventHandler();
           }
       });
       removeButton.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
               removeButtonEventHandler();
           }
       });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);

       update();

    }

    // Event Handler for the Add button
    private void addButtonEventHandler() {
        items.add(newItemField.getText());
        update();
    }


    // Event Handler for the Remove button
    private void removeButtonEventHandler() {
        items.remove((String)itemsList.getSelectedValue());
        update();
```

```
    }
    // Update all the components
    private void update() {
        itemsList.setListData(items);
    }

     public static void main(String[] args) {
         // Set up the items to be put into the list
         Vector<String> todoItems = new Vector<String>();
         todoItems.add("wash my car");
         todoItems.add("go to dentist");
         todoItems.add("shovel the laneway");
         todoItems.add("pick up milk");
         todoItems.add("laundry");

         TodoListFrame frame = new TodoListFrame(todoItems);
         frame.setVisible(true);
     }
}
```

Notice:

- we made a single **`update()`** method that is called from the Add and Remove button event handlers as well as when the window is first opened.
- the update really just updated the data in the **JList** to reflect the changes in the model.

So, REMEMBER the two VERY important things that you should normally do in your **update()** method:

1. **Read the model's information**
2. **Change the "look" of the interface components**

There are two problems with the application:

1. When no text is in the text field and **Add** is pressed, a blank item is added.
2. When no item is selected from the list and a **Remove** is done, our code tries to remove a **null** item from the model. Since the model is a **Vector**, and the remove method for vectors handles this attempt with grace (i.e., no exception), then it is not really a problem. However, what if someone changes the underlying model to be something other than a vector ? We should fix this.

How can we fix these ? First check if there is any text before doing the **Add**. If there is none, don't add. The change occurs in the event handler for the **Add** button. Here is the changed code:

```
private void addButtonEventHandler() {
    if
 (newItemField.getText().length()
 > 0) {
         items.add(newItemField.getText());
         update();
    }
}
```

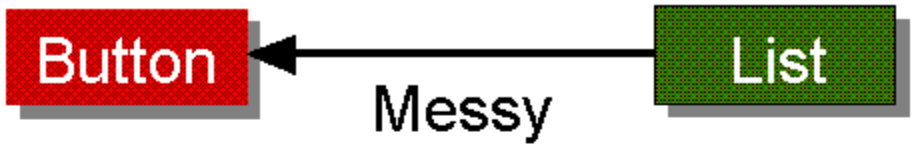
For the remove problem, we would like to have a way of determining whether or not anything was selected from the list. To do this, we merely ask if the selected list value is **null**:

```
private void removeButtonEventHandler() {
    if
 (itemsList.getSelectedValue()
 != null) {
         items.remove((String)itemsList.getSelectedValue());
         update();
    }
}
```

We have fixed the problems ... but now we have a messy situation. It seems that the **JButtons** MUST know about the **JList** component. The **JButtons** are somehow "tied" with the **JList** so that if the **JList** is removed and perhaps replaced by something else, we must go into the **JButton** event handler and make changes. This is "messy".

- code is not easily maintained when many components rely on other components
- components need to know exactly how they affect other components

A better way to solve these problems is to make the list selection a part of the model. We would like to use the model as a kind of "middle man" between all component interaction so that this "dependence" between components is severed.



Messy

Clean

So ... we will keep track of the item that was selected and this will be part of our model.
Of course this means that we will have to handle the *selection* event for the **JList** component.

Here is what we need to add/change:

1. Add an instance variable to store the selected item:

```
private String  selectedItem; // Part of the model
```

2. Modify the constructor to initially select a list item if there is one available, and also add a list selection listener for whenever someone makes a selection from the list:

```
public TodoListFrame(Vector<String> todoEntries) {
    super("To Do List");
    items = todoEntries;

   if (items.size() > 0)
       selectedItem = (String)items.firstElement();
   else
       selectedItem = null;

    // ... Some code has been omitted ...

    // Add listeners for the buttons and then enable them
```

```
        addButton.addActionListener(...);
        removeButton.addActionListener(...);

       itemsList.addListSelectionListener(new
    ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                listSelectionEventHandler();
            }
       });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);
        update();
    }
```

3. Write the **listSelectionEventHandler()** so that the new instance variable is updated to reflect the latest selection made:

```
private void listSelectionEventHandler() {
    selectedItem = (String)itemsList.getSelectedValue();
    update();
}
```

4. Modify the **Remove** button event handler to use the new **selectedItem** variable now. We must make sure to set the **selectedItem** to **null** after an item is removed, since the list will not have anything selected in it anymore:

```
private void removeButtonEventHandler() {
   if (selectedItem != null)
 {

  items.remove(selectedItem);
        selectedItem == null;
         update();
    }
}
```

5. Modify the **update()** method to make sure that the **selectedItem** variable always matches the item selected from the list:

```
private void update() {
    itemsList.setListData(items);

   itemsList.setSelectedValue(selectedItem, true);


}
```

At this point, we have a strange bug in our application. It seems that we are unable to actually select anything from our **JList** now ! The problem is that our **update()** method calls **setSelectedValue()** which changes the contents of the list. This generates an internal **valueChanged()** event ... which is the event that we are handling. Hence, when we do a list selection, our event handler is called, which itself calls **update()**. Then update generates another **valueChanged()** event which again calls our handler and **update()** again. Really, this is an endless loop. JAVA is able to deal with this problem without generating exceptions, but it does not give us desirable results in that we cannot really select anything from the list ;).

The simplest and most logical solution is to disable the list selection listener while updating and then re-enable it afterwards. To do this, we will need the actual listener object and de-register it at the beginning of the **update()** method, then re-register it afterwards. Here are the steps:

1. Declare the following instance variable:

```
private ListSelectionListener   itemsListListener;
```

2. Store the ListSelectionListener that was created in our constructor:

```
itemsList.addListSelectionListener(itemsListListener =
    new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
            listSelectionEventHandler();
        }
    }
);
```

3. Disable and then re-enable the listener in our **update()** method:

```
private void update() {

  itemsList.removeListSelectionListener(itemsListListener);
     itemsList.setListData(items);
     itemsList.setSelectedValue(selectedItem, true);
    itemsList.addListSelectionListener(itemsListListener);
}
```

The application now works and has nice clean code.

We have prevented the **Add** button from doing anything when there is no text in the text field and the **Remove** button from doing anything when there is nothing selected. It is best to let the user know that these buttons will not work under these circumstances. The proper way of doing this is to disable the buttons at these times. Let us make these changes now. We will make use of the **`setEnabled()`** method for buttons which enables or disables the button according to a given boolean.

Where do we write the code for disabling these buttons ? Well, does it have to do with functionality or with appearance ?
After some thought, you realize that this is a "cosmetic" issue and that it has to do with the "look" of the buttons. Hence, we should make these changes in the **update()** method.

Disabling the **Remove** button is easy. Just add the following line to the **update()** method:

```
removeButton.setEnabled(selectedItem != null);
```

For the **Add** button, we can add a similar line:

```
addButton.setEnabled(newItemField.getText().length() > 0);
```

There is a small problem. When the interface starts up, the text field is empty and so the **Add** button is disabled. That's good. But when the user starts typing in the text field, there is then text in the text field but the **Add** button remains enabled. The problem is that **update()** is not being called unless an event occurs. So we need to have some kind of event for when the user types text in the text field. So we will need to make some changes. In addition, this approach to enabling the **Add** button results in a dependency on there being a text field. We should create another instance variable to indicate whether or not there is any text in the text field. We can just use a boolean, but we may as well keep the whole item that is in there instead of just a flag. First we need to make the following additions:

```
// Add this as an instance variable
private String  newItem;
```

```
// Add this to the constructor
newItem = "";
```

Now, we could use handle **ActionEvents** for the **TextField**, but these events only occur when the user presses **Enter** within that field. Instead, we will make use of something called a ***DocumentListener***. Every **JTextField** has a ***document*** object associated with it that can be obtained with **`getDocument()`**. We can then add the listener to this object. This way, we can handle events that occur whenever the text changes (character by character) even if no **Enter** key is pressed. We need to make the following changes to our code:

```
// Declare this instance variable at the top
private DocumentListener   newItemFieldListener;

// Add this to the constructor
newItemField.getDocument().addDocumentListener(newItemFieldListener =
    new DocumentListener() {
        public void changedUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
        public void insertUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
        public void removeUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
    }
);

// Add this event handler for the text field
private void handleTextFieldEntry() {
    newItem = newItemField.getText();
    update();
}
```

Notice that there are three events that may be generated by the **DocumentListener**. These correspond to typing in text, inserting and removing (i.e., paste/cut). Notice that despite the particular edit change in the text field, all three events call our helper method which simply sets the **newItem** variable to match the contents of the text field.

Of course, we will want to now modify the event handler for the **Add** button to make use of the **newItem** field. We will also select the item that was just added. This is not necessary, but it is a nice form of "feedback" for the user:

```
private void addButtonEventHandler() {
    if (newItem.length() > 0) {
        items.add(newItem);
        selectedItem = newItem; // select the newly added item
        update();
    }
}
```

We should also modify our **update()** method so that the **Add** button uses the **newItem** variable now:

```
addButton.setEnabled(newItem.length() > 0);
```

Now our code should work fine.

One last feature that we will add is to clear out the text field AFTER we add a new item. Well, to do this, we will have to set the **newItem** to "" in the **Add** button handler as follows:

```
private void addButtonEventHandler() {
    if (newItem.length() > 0) {
```

```
            items.add(newItem);
            selectedItem = newItem;
            newItem = "";
            update();
        }
    }
```

Notice however, that at this point, the **newItem** variable will be "", but there will still be some contents in the text field ... so they are not in agreement. To fix this, we will have to actually clear out the text field contents. This, of course, has to do with the appearance of the text field, so we could try placing the following code within the **update()** method:

```
newItemField.setText(newItem);
```

But, we have a small problem. If we were to run our code right now, we would notice a bug when we tried to type into the text field. Our code would generate the following exception:

```
java.lang.IllegalStateException: Attempt to mutate in notification
```

JAVA version 1.4 and onward, however, will not allow us to set or modify the contents of the **JTextField** while we are handling one of its document events. So, we will need to make the following changes:

1. avoid setting the text field's text within the **update()** whenever we call **update()** from a **DocumentListener**
2. remove/add the document listener at the start/end of the **update()** method.

To do this, we will make a special **update()** method. In fact, we will split up our **update()** method as follows:

```
// Update called by Document Listeners directly
private void update(boolean calledFromTextField) {
    itemsList.removeListSelectionListener(itemsListListener);

newItemField.getDocument().removeDocumentListener(newItemFieldListener);

    itemsList.setListData(items);
    itemsList.setSelectedValue(selectedItem, true);
    removeButton.setEnabled(selectedItem != null);
    addButton.setEnabled(newItem.length() > 0);
    if (!calledFromTextField)  newItemField.setText(newItem);

    itemsList.addListSelectionListener(itemsListListener);

newItemField.getDocument().addDocumentListener(newItemFieldListener);
}

// Update used by all the event handlers, except Document event
handlers
private void update() {
    update(false);
}
```

The make the following change in the **handleTextFieldEntry()** method:

```
private void handleTextFieldEntry() {
    newItem = newItemField.getText();
    update(true); // since we called from this document listener
}
```

Note that all event handlers will call the usual **update()** method (which will set the text in the text field), but the DocumentListeners will call **update(true)** so as to avoid setting the text illegally.

Thus, when we press the **Add** button and set the **newItem** variable to "", the call to **update()** at the end of the event handler will ensure that the text field's contents are set to "". Meanwhile, if we make changes to the text field directly, we will not be updating the text field appearance, as it does not need updating since it will always have the same contents as the **newItem** variable anyway.

Now the last improvement is within the **update()** method. When we make changes to our code by adding or removing components, we must go to the update method and make changes. It is difficult to determine what code pertains to which components. We can extract this update code so that we make separate methods such as **updateTextField()**, **updateList()**, and **updateButtons()**. These will do the corresponding updates for the individual components. This way, when we modify or remove a component, it is clear as to what code should be modified/removed. This alternative method also allows us to update only those components that have changed (not all). This is good if the interface becomes slow in drawing the components. We can speed everything up by only updating necessary components. We can also extract the code for disabling/enabling the listeners into separate methods as well.

The final completed code is shown below (the class name has been changed to **ToDoListFrame2**):

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

public class ToDoListFrame2 extends JFrame {
    private JTextField   newItemField;
    private JList        itemsList;
    private JButton      addButton;
    private JButton      removeButton;

    // Listeners that need to be disabled/enabled during update()
    private ListSelectionListener    itemsListListener;
    private DocumentListener         newItemFieldListener;

    private Vector<String>  items;        // The model
    private String          selectedItem; // item selected in the list
    private String          newItem;      // String contained in text field

    public ToDoListFrame2() {
        this(new Vector<String>());
    }

    public ToDoListFrame2(Vector<String> todoEntries) {
        super("To Do List");
        items = todoEntries;

        if (items.size() > 0)
            selectedItem = (String)items.firstElement();
        else
            selectedItem = null;

        newItem = ""; // nothing in the text field yet

        initializeComponents();

        // Add listeners for the buttons, list and text field
        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                addButtonEventHandler();
            }
        });
        removeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

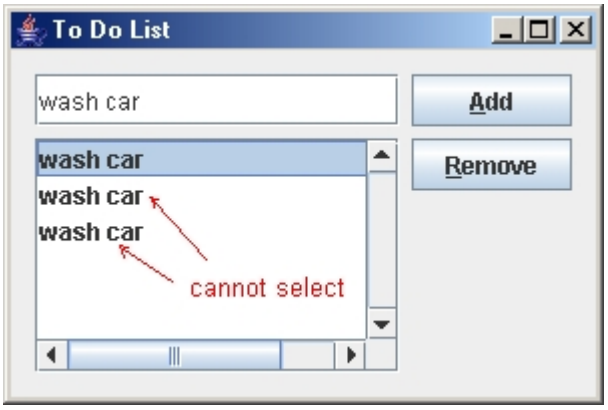
```
                removeButtonEventHandler();
            }
        });

        itemsList.addListSelectionListener(itemsListListener =
            new ListSelectionListener() {
                  public void valueChanged(ListSelectionEvent e) {
                      listSelectionEventHandler();
                  }
            }
        );
        newItemField.getDocument().addDocumentListener(newItemFieldListener =
            new DocumentListener() {
                public void changedUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
                public void insertUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
                public void removeUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
            }
        );

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);

        update();
    }

    // Build the frame by adding all the components
    private void initializeComponents() {
        GridBagLayout layout = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        setLayout(layout);

        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 = 1;
        constraints.weighty = 0;
        layout.setConstraints(newItemField, constraints);
        add(newItemField);

        addButton = new JButton("Add");
        addButton.setMnemonic('A');
        constraints.gridx = 1;
        constraints.gridy = 0;
        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);
        add(addButton);

        itemsList = new JList();
        itemsList.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxxxxxx");
        JScrollPane scrollPane = new JScrollPane(itemsList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        constraints.gridx = 0;
        constraints.gridy = 1;
```

```
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(3, 12, 12, 3);
        constraints.weightx = 1;
        constraints.weighty = 1;
        layout.setConstraints(scrollPane, constraints);
        add(scrollPane);

        removeButton = new JButton("Remove");
        removeButton.setMnemonic('R');
        constraints.gridx = 1;
        constraints.gridy = 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);
        add(removeButton);
    }

    // Event Handler for the Add button
    private void addButtonEventHandler() {
        if (newItem.length() > 0) {
            items.add(newItem);
            selectedItem = newItem; // select the newly added item
            newItem = "";           // clear the text
            update();
        }
    }

    // Event Handler for the Remove button
    private void removeButtonEventHandler() {
        if (selectedItem != null) {
            items.remove(selectedItem);
            selectedItem = null;
            update();
        }
    }

    // Event Handler for List Selection
    private void listSelectionEventHandler() {
        selectedItem = (String)itemsList.getSelectedValue();
        update();
    }

    // Handler for entering text in the text field
    private void handleTextFieldEntry() {
        newItem = newItemField.getText();
        update(true);
    }

    // Update all the components
    private void update(boolean calledFromTextField) {
        disableListeners();

        updateList();
        updateButtons();
        if (!calledFromTextField)
            updateTextField();

        enableListeners();
    }

    private void update() {
        update(false);
    }

    private void disableListeners() {
```

```
        itemsList.removeListSelectionListener(itemsListListener);
        newItemField.getDocument().removeDocumentListener(newItemFieldListener);
    }
    private void enableListeners() {
        newItemField.getDocument().addDocumentListener(newItemFieldListener);
        itemsList.addListSelectionListener(itemsListListener);
    }
    private void updateList() {
        itemsList.setListData(items);
        itemsList.setSelectedValue(selectedItem, true);
    }
    private void updateButtons() {
        removeButton.setEnabled(selectedItem != null);
        addButton.setEnabled(newItem.length() > 0);
    }
    private void updateTextField() {
        newItemField.setText(newItem);
    }

    public static void main(String[] args) {
        // Set up the items to be put into the list
        Vector<String> todoItems = new Vector<String>();
        todoItems.add("wash my car");
        todoItems.add("go to dentist");
        todoItems.add("shovel the laneway");
        todoItems.add("pick up milk");
        todoItems.add("laundry");

        ToDoListFrame2 frame = new ToDoListFrame2(todoItems);
        frame.setVisible(true);
    }
}
```

Exercise:

But wait a minute! There is still a problem with our code. If we try adding two or more items with the same name, JAVA will not allow us to select any of these items except the topmost one!



Looking at the **updateList()** method and the **listSelectionEventHandler()** method, can you determine what the problem is ? As a practice exercise, try to solve this problem by making some appropriate changes to the code. You may want to look at the **JList** class in the JAVA documentation.

---

4 A Traffic Light Application

What's in This Set of Notes?

It is a good idea now to look at an application with a more interesting model component. We will look at the example of a TrafficLight model object with a user interface attached to it. We will also look at how we can use a **Timer** object to have automatic updating of the TrafficLight.

Here are the individual topics found in this set of notes (click on one to go there):

4.1 Application Description

The purpose of this application is to give you more experience in understanding how to create an application by using the following steps:

1. Understand what you want the interface to do.
2. Understand what the model should be and then develop the model.
3. Create the user interface layout.
4. Connect the interface to the model.

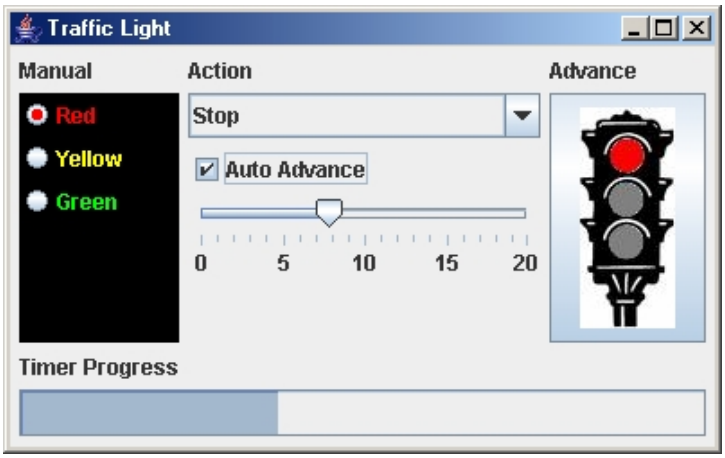
You should follow these steps whenever creating an interface in this course.

Understanding the interface:

First let us describe the application. The application is a window with the following components:

- A set of 3 **JRadioButtons** to represent the traffic light state (Red, Yellow or Green).
- A **JComboBox** to indicate actions (i.e., Stop, Yield and Go) which correspond to the traffic light state.
- a JButton with a traffic light icon which reflects (and allows advancing of) the state of the traffic light.
- a **JCheckBox** which will allow the light to advance automatically, based on a timer.
- a **JSlider** that will allow the user to adjust the speed of the automatic time advance feature.
- a **JProgressBar** that will indicate how much longer the traffic light will remain in its current state.
- various JLabels to make the window more self-explanatory.

The interface will look something like this:



Here is how the behaviour of the interface is described:

- The radio buttons and the combo box should be linked together such that if the **Red** light is selected, then the **Stop** item is selected automatically in the combo box as well. The same goes for the **Yellow**/**Yield** and **Green**/**Go** options as well. Also, if a selection is made in the combo box, then this selection is also made in the radio button group as mentioned.

- The status pane should show the following messages depending on the state of the lights:
  - Red: "Red Traffic Light Means: 'Stop Please'"
  - Yellow: "Yellow Traffic Light Means: 'Yield to Others'"
  - Green: "Green Traffic Light Means: 'Go Really Fast'"

- The **Advance** button should cause the light to advance to the next state in a cyclic fashion in the order red, green, yellow, red, green, ...

- When the **Auto** button is pressed, the lights will advance automatically such that the red light stays on for 6 seconds, the green for 8 seconds and the yellow for 3 seconds. Pressing the **Advance** button while in **Auto** mode will also advance the state of the light. The progress bar should indicate the number of seconds that the traffic light has been in its current state.

4.2 Developing the Model

How do we make the model ? What is the model ? It is a traffic light.
Two simple ways of representing a traffic light are as:

- an integer from 1 to 3 (where 1 = Red, 2 = Yellow, 3 = Green) or some other numbering scheme
- one of the following Strings: "Red", "Yellow" or "Green"

This would work, but then we don't get to define any behaviour such as **getState()** or **advanceState()**. We should make our own class.
Within this class, we will simply use an integer to store the state.

Note:

- It is always a good idea to make a "stand-alone" model with behaviour such that any kind of user interface can be plugged into it.

Here is the model code:

```
public class TrafficLight {
    int  currentState;

    // Constructor that makes a red traffic light
    public TrafficLight() {
        currentState = 1;
    }

    // Advance the traffic light to the next state
```

```
    public int advanceState() {
        currentState = ++currentState % 3 + 1;
        return currentState;
    }

    // Return the state of the traffic light (as a number from 1 to 3)
    public int getState() {
        return currentState;
    }

    // Set the state of the traffic light (as a number from 1 to 3)
    // If the integer is out of range, do nothing
    public void setState(int newState) {
        if ((newState > 0) && (newState <4))
            currentState = newState;
    }

    // Return a string representation of the traffic light
    public String toString() {
        String[] colours = {"Red", "Yellow", "Green"};
        return colours[currentState] + " Traffic Light";
    }
}
```

Making a traffic light is easy:

```
    new TrafficLight();
```

We can ask for the state or change the state. A state of 1 is a Red light, 2 is a Yellow light and 3 is a Green light. The advance method will cause the state to cycle as follows: 1, 3, 2, 1, 3, 2, 1, 3, 2, 1, ...

Notice that there are no **System.out.println()** messages here, nor is there any keyboard input. That is because, those are actually I/O operations and they depend heavily on the type of user interface that will be used. We leave that kinda "stuff" out of the model let the user interface worry about those issues. (Sometimes we may have println statements for debugging/testing purposes, but ultimately these should be removed from the model class code). That way, we can "plug-in" the model into any user interface and the model becomes modular, clean, shared code. So remember,

Model classes should NOT:

- print to the console, nor
- get input from the keyboard

4.3 Designing the User Interface Layout

Now ... the user interface. We will use a GridBagLayout manager. The basic code for building the window is shown below, although we will add more to it. There is nothing tricky about the code.

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

public class TrafficLightFrame extends JFrame {

    // These are the window's components
    private JRadioButton[]   buttons = new JRadioButton[3];
    private JButton          advButton;
    private JProgressBar     progressBar;
    private JSlider          slider;
    private JComboBox        actionList;
    private JCheckBox        autoButton;
```

```
public TrafficLightFrame(String title) {
    super(title);
    buildWindow();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(400, 250);
}

// Add all components to the frame's panel
private void buildWindow() {
    GridBagLayout         layout = new GridBagLayout();
    GridBagConstraints    constraints = new GridBagConstraints();
    setLayout(layout);

    // Add all the labels
    JLabel label = new JLabel("Manual");
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.gridwidth = 1;
    constraints.gridheight = 1;
    constraints.weightx = 0;
    constraints.weighty = 0;
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.NORTHWEST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(label, constraints);
    add(label);

    label = new JLabel("Action");
    constraints.gridx = 1;
    layout.setConstraints(label, constraints);
    add(label);

    label = new JLabel("Advance");
    constraints.gridx = 2;
    layout.setConstraints(label, constraints);
    add(label);

    label = new JLabel("Timer Progress");
    constraints.gridx = 0;
    constraints.gridy = 4;
    layout.setConstraints(label, constraints);
    add(label);


    // Add the Radio Buttons
    ButtonGroup lights = new ButtonGroup();
    JPanel aPanel = new JPanel();
    aPanel.setLayout(new BoxLayout(aPanel, BoxLayout.Y_AXIS));
    aPanel.setBackground(Color.black);
    for (int i=0; i<3; i++) {
        buttons[i] = new JRadioButton("", false);
        buttons[i].setBackground(Color.black);
        lights.add(buttons[i]);
        aPanel.add(buttons[i]);
    }
    buttons[0].setText("Red");
    buttons[1].setText("Yellow");
    buttons[2].setText("Green");
    buttons[0].setForeground(Color.red);
    buttons[1].setForeground(Color.yellow);
    buttons[2].setForeground(Color.green);
    constraints.gridx = 0;
    constraints.gridy = 1;
    constraints.gridheight = 3;
    constraints.fill = GridBagConstraints.BOTH;
    layout.setConstraints(aPanel, constraints);
    add(aPanel);

    // Make the Actions List
    String[] actions = {"Stop", "Yield", "Go"};
    actionList = new JComboBox(actions);
    constraints.gridx = 1;
    constraints.gridy = 1;
    constraints.gridheight = 1;
    constraints.weightx = 1;
```

```
        constraints.fill = GridBagConstraints.HORIZONTAL;
        layout.setConstraints(actionList, constraints);
        add(actionList);

        // Make the Slider
        slider = new JSlider(JSlider.HORIZONTAL, 0, 20, 1);
        slider.setMajorTickSpacing(5);
        slider.setMinorTickSpacing(1);
        slider.setPaintTicks(true);
        slider.setPaintLabels(true);
        constraints.gridx = 1;
        constraints.gridy = 3;
        layout.setConstraints(slider, constraints);
        add(slider);

        // Add the auto checkbox button
        autoButton = new JCheckBox("Auto Advance");
        constraints.gridx = 1;
        constraints.gridy = 2;
        constraints.fill = GridBagConstraints.BOTH;
        layout.setConstraints(autoButton, constraints);
        add(autoButton);

        // Add the Advance Picture button
        advButton = new JButton(new ImageIcon("RedLight.jpg"));
        constraints.gridx = 2;
        constraints.gridy = 1;
        constraints.gridheight = 3;
        constraints.weightx = 0;
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 0, 5);
        layout.setConstraints(advButton, constraints);
        add(advButton);

        // Add the progress bar
        progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0, 8);
        constraints.gridx = 0;
        constraints.gridy = 5;
        constraints.gridwidth = 3;
        constraints.gridheight = 1;
        constraints.weightx = 1;
        constraints.weighty = 2;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 5, 5);
        layout.setConstraints(progressBar, constraints);
        add(progressBar);
    }

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

When we run the application at this point, the components all appear on the window, although the interface does not really do anything useful yet. Some interesting things to note are:

- We made an array of **JRadioButtons** so that we can access the buttons by index. This is useful since we need to change the state of a button based on the traffic light's state (which will be an integer).
- The **JRadioButtons** were added to a **JPanel** so that it is easier to keep them grouped together when the window resizes.
- The **JProgressBar** uses a range from 0 to 8. We will see more about this later.
- The **JButton** initially has a picture called **"RedLight.jpg"** representing a red traffic light. This file must be in the same directory as this code as well as two others which will be used later, called **"GreenLight.jpg"** and **"YellowLight.jpg"**.

4.4 Connecting it all Together

Now we must connect the model and the interface together to make it all work.

We begin by adding an instance variable representing the model:

```
// This is the model
private TrafficLight aTrafficLight = new TrafficLight();
```

This model MUST ALWAYS be synchronized with the user interface. That is, the user interface should always reflect perfectly the state of the traffic light. That means, upon startup, the application should show the default traffic light state in the radio buttons, the combo box and the picture on the advance button. To do this, we will make sure that our **update()** method always updates the components properly. In fact, it is best to first write the update() method BEFORE writing any event handlers. That way, the interface always reflects the model, making debugging the event handlers easier. Also, it ensures that you put your code in the correct place.

Always follow these steps:

1. **Create the model**
2. **Create your user interface "look" (i.e., frame with its components)**
3. **Write the update() method to refresh ALL components that may change their look**
4. **Write and test your event handlers one by one**

So now let us write our update method. It is often the case that we write helper methods (one for each component or group of components) to help keep our code neat and tidy. We will start by updating the radio buttons, combo box and advance button only, since they deal directly with the traffic light state:

```
// Update all relevant components according to the traffic light state
public void update() {
    updateRadioButtons();
    updateComboBox();
    updateAdvanceButton();
}
```

Notice the three helper methods. These methods should put the appropriate data into the components based on the current state of the model. These methods will also be used later on when we make changes to the model and need to reflect these changes in the window.

```
private void updateRadioButtons() {
    for (int i=0; i<3; i++)
        buttons[i].setSelected(aTrafficLight.getState() == (i+1));
}
private void updateComboBox() {
    actionList.setSelectedIndex(aTrafficLight.getState() - 1);
}
private void updateAdvanceButton() {
    String[] iconNames = {"RedLight.jpg","YellowLight.jpg","GreenLight.jpg"};
    advButton.setIcon(new ImageIcon(iconNames[aTrafficLight.getState()-1]));        }
}
```

Notice a couple of things:

- Each helper update method corresponds to a single component (or group, as with radio buttons).
- Each helper update method relies on the model only, not on any other components (this is clean).

In our constructor, we will call the **update()** method (after we add the components) so that when the frame is created, the components will all be updated to reflect the initial state of the model.

```
public TrafficLightFrame(String title) {
    super(title);
    buildWindow();

   update();

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(400, 250);
```

```
    }
```

OK. When we run our window now, it should represent our default traffic light state (which is red). So, the top radio button should be selected, the combo box should indicate "Stop" and the red light image should be on the advance button.

It is now time to write the event handlers.

How can we get the radio buttons to change the state of the model ? We need to add an action listener for each button. We need to write the following in the constructor in order to register the listener for each Radio Button:

```
    // Register the JRadioButton Listeners
    for (int i=0; i<3; i++)
        buttons[i].addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleRadioButtonPress((JRadioButton)e.getSource());
            }
        });
```

Then, we can write the **handleRadioButtonPress()** helper method. What should it do ? Do you remember ... its simple ... change the model ... then call **update()**. But how does it change the model ? Well, the model's state should match the index of the button, shouldn't it ?

```
    // This is the radio button event handler
    private void handleRadioButtonPress(JRadioButton source) {
        for (int i=0; i<3; i++) {
            if (source == buttons[i])
                aTrafficLight.setState(i+1);
        }
        update();
    }
```

Notice that the event handler determines which button was pressed by using **`getSource()`** and comparing this to the actual buttons by using the identity operator. Notice also, that we simply ask the model to change its state and then call **`update()`**. The **`update()`** method will take care of making sure that all other components will reflect the recent model changes. If we were to test things now, we would see that by clicking the radio buttons, the ComboBox and icon on the Advance button would be updated properly to reflect the model changes as desired.

To get this to happen for selecting combo box items as well, we merely add an **ActionListener** to the ComboBox (again in out constructor):

```
    // Register the JComboBox Listener
    actionList.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            handleComboBoxSelection((JComboBox)e.getSource());
        }
    });
```

Now here is the helper method which simply determines the index of the selected item and updates the model accordingly

```
    // The ComboBox selection event handler
    private void handleComboBoxSelection(JComboBox source) {
        aTrafficLight.setState(source.getSelectedIndex() + 1);
        update();
    }
```

Once again, we can use **`getSource()`** to get the component (i.e. the combo box). Alternatively, we could have just accessed the **actionList** instance variable (simpler code, but it would be dependent on the variable name ... which is not too bad). After making these changes, both the radio buttons and combo box are "synchronized" in that selection from one causes selection in the other. Neat isn't it ?

There is a slight problem. The **setSelectedIndex()** call in our **updateComboBox()** method will generate an **ActionEvent** again, leading to another call to **update()** and we are led into an endless loop as with our "todo List" example. So we need to disable the **ActionListener** for the combobox while we are updating. We can store the **ActionListener** into an instance variable when we create it and then remove/add it in the **update()**

method:

```
// Add this new instance variable
private ActionListener    comboBoxListener;

public TrafficLightFrame(String title) {
    ...
    actionList.addActionListener(comboBoxListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            handleComboBoxSelection((JComboBox)e.getSource());
        }
    });
    ...
}

public void update() {
    actionList.removeActionListener(comboBoxListener);
    updateRadioButtons();
    updateComboBox();
    updateAdvanceButton();
    updateProgressBar();
    actionList.addActionListener(comboBoxListener);
}
```

So what about the Advance button ? Well, it too should advance the state of the model and then update all components. We add this code to the constructor:

```
    advButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            handleAdvanceButtonPress();
        }
    });
```

And then add this very simple helper method:

```
// This is the Advance button event handler
private void handleAdvanceButtonPress() {
    aTrafficLight.advanceState();
    update();
}
```

Wow! Isn't this getting really easy now. We just ask the model to do the advancing of state and then call **update()**. Now everything is just peachy.

4.5 Hooking up the Timer

Our last step is to get the **Timer** working properly. This is actually quite easy. We will add an instance variable to hold the **Timer** so that we can grab it whenever we want to Start or Stop it:

```
private Timer  aTimer;
```

Now we must make the timer. To make one, we simply call a constructor which specifies the number of milliseconds that we would like between timer events as well as the listener (i.e., event handler). As it turns out, the listener is simply an ActionListener ... just as with JButtons. We can write the following code in our constructor to generate a timer tick twice per second (i.e., 1secons / 500ms = 2 seconds):

```
// Add a timer for automode.  Set it to go off every 500 milliseconds
aTimer = new Timer(500, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleTimerTick();
    }
});
```

After doing this, the timer has NOT yet started and so no events are actually generated. We have to explicitly start and/or stop the timer with separate methods. When do we want the timer to start anyway ? Well, if the "Auto Advance" checkbox is turned on, we should start the timer. If it is turned off, we should stop the timer. We need to add an event handler for the check box. We can do this by adding the following code to the

constructor:

```
autoButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleAutoButtonPress((JCheckBox)e.getSource());
    }
});
```

And here is the helper method:

```
// This is the Auto button event handler
private void handleAutoButtonPress(JCheckBox source) {
    if (source.isSelected())
        aTimer.start();
    else
        aTimer.stop();
    update();
}
```

As can be seen, when the checkbox is turned on, the timer is started. When turned off, it is stopped. Let us now look at what we must do on each timer event. We can write the following code as our **TimerEventHandler**:

```
// This is the Timer event handler
private void handleTimerTick() {
    aTrafficLight.advanceState();
    update();
}
```

Wow! It is the same code as the advance button. In fact, we could have used the exact same event handler ! If you were to test this, you would see the traffic light change state every 0.5 seconds as long as the check box remains selected. Once turned off, the advancing stops. Notice that the advance button will still cause the light to change state as well.

We forgot one of our criteria ... we must have the lights remain in a certain state for different amounts of time. Remember, red for 6 seconds, green for 8 seconds, then yellow for 3 seconds. We will have to keep a counter of some kind to keep track of how long the light has been in the current state. Once it has been on long enough, we advance.

But doesn't this have something to do with the model ? Shouldn't the traffic light itself know how long to remain in each state ? It looks like we may want to adjust our model. We will need to add some kind of counter to our model that counts how long the traffic light remains in a certain state. We can add the following instance variable to our **TrafficLight** model class:

```
private int     stateCount;    // amount of time in this state
```

We can also choose a maximum amount of time that the traffic light remains in any particular state. This can be used by the progress bar later. We will define the following static constant:

```
public static final int  MAX_TIME_COUNT = 8;
```

We should set the **stateCount** to zero in our constructor and reset this counter to zero whenever we make state changes.

```
public TrafficLight() {
    currentState = 1;
    stateCount = 0;
}
public int advanceState() {
    currentState = ++currentState % 3 + 1;
    stateCount = 0;
    return currentState;
}
public void setState(int newState) {
    if ((newState > 0) && (newState <4)) {
        currentState = newState;
        stateCount = 0;
    }
```

```
}
```

We will likely want a get method for this counter too:

```
public int getStateCount() {
    return stateCount;
}
```

Now how (and when) do we go about making the traffic light advance automatically over time ? Well, we will let the user interface decide on how fast to make the timer go, but we will want a method in our model that "simulates" the passing of time. We can write a method called advanceTime() that will be called once per time unit (e.g., once per second) to advance the time (i.e., our stateCount counter). So then in this method we just have to check and see if the counter reached its limit according to what state it is in:

```
// Simulate a single time unit of time passing by
public void advanceTime() {
    // advance the time spent in the current state
    stateCount++;

    // The number of seconds (i.e., time units) that
    // the traffic light remains in each state
    int[] stateTimes = {6, 3, 8};

    // Check if we have reached time limit for current state
    if (stateCount > stateTimes[currentState-1])
        advanceState();
}
```

So now we can get back to our user interface. We need to change the call from **advanceState()** to **advanceTime()** in our timer tick event handler:

```
private void handleTimerTick() {
    aTrafficLight.advanceTime();
    update();
}
```

So what about the progress bar ? How can we have it reflect the current count ? We first need to go back to our code that built our window and now use the static constant that we defined in our model representing the maximum limit for the progress bar:

```
 progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0,
 TrafficLight.MAX_TIME_COUNT);
```

We will need to update the progress bar every time that the traffic light increments its stateCount. We will write an update method for this:

```
// Update all relevant components according to the traffic light state
public void update() {
    updateRadioButtons();
    updateComboBox();
    updateAdvanceButton();

   updateProgressBar();

}

// Update the progress bar
private void updateProgressBar() {
    progressBar.setValue(aTrafficLight.getStateCount());
}
```

That's it. The progress bar will now show the amount of time that the traffic light remains in its current state.

The last remaining task for us to complete is to get the slider working. We need to register for a **stateChanged** event. We can add this to our constructor:

```
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        handleSlider((JSlider)e.getSource());
```

```
        }
    });
```

Now of course, we need to write the event handler. We only want to make a change to the timer delay when the user lets go of the slider. So while the user is adjusting the value, we do not want to handle the event. We can check for this with a **getValueIsAdjusting()** method call to our slider. Then, we can get the value of the slider with **getValue()**. It will return an integer within the range that we specified when creating the slider (from 0 to 20 in our case). Lastly, we just need to call **setDelay(int)** for our timer, passing it in the new delay value. Of course, if the delay is 0, we probably want to pick a HUGE delay such as Integer.MAX_VALUE. Also, we will need to restart the timer after making the change (as long as it has already been started by the check box):

```
// This is the Slider event handler
private void handleSlider(JSlider source) {
    if (!source.getValueIsAdjusting()) {
        int delay = source.getValue();
        if (delay > 0) {
            aTimer.setDelay(1000/delay);
            if (aTimer.isRunning())
                aTimer.restart();
        }
        else
            aTimer.setDelay(Integer.MAX_VALUE);
        update();
    }
}
```

Well ... that is it! We are done.

4.6 Splitting up the Model, View and Controller

Taking a look at our traffic light application, we notice a couple of things:

- The **TrafficLightFrame** class mixes together code related to
  - how the interface looks and
  - how the interface behaves.
- There is a problem if some other "entity" changes the **TrafficLight** model.....then the **TrafficLightFrame** GUI will not be updated.

We will now make a distinction between a view and a controller.

Recall that a ***view*** is:

- the part of the application that specifies how the model is shown visually.
- the part of the code that deals with the appearance of the interface.

and a ***controller*** is:

- the part of the application that specifies how the model interacts with the view.
- the part of the code that deals with the behaviour of the interface.
- code that serves as a "mediator" between the model and the view.

Also, recall that it is ALWAYS a good idea to separate the model, view and controller (MVC):

- code is cleaner and easier to follow when the view and controller are separated
- we may want to have multiple views and controllers on the same model.

Consider this second point for a moment. If we have multiple controllers and views on the same model (i.e., two windows for the same traffic light), then we have two different windows affecting the model.

How can we cleanly allow one view to change the model and have the other view updated automatically ?

We could have each view/controller keep track of all other views and controllers. This could get messy, especially if we get more controllers and views later on down the road. A nice solution is to have the model inform all interested parties whenever it has changed. Things brings up the notion of a commonly used Object-

Oriented Design pattern called the "**subject/observer**" pattern.

If we make our model inform all interested applications when it has changed, then it must somehow keep track of all applications that need to be updated when a change occurs. The model becomes a kind of ***subject*** which the applications ***observe***. To make a clean connection between the two, we will make our model class implement a standardized **Subject** interface that allows observers to register or un-register with it. We will write the following interface:

```
public interface Subject {
    public void registerObserver(Observer observer);
    public void unregisterObserver(Observer observer);
}
```

The model should therefore keep track of the applications that have registered with it so that it can later update them. The applications should make sure that they have an **update()** method so that this will all work. To do this, we will have the applications implement an **Observer** interface.

```
public interface Observer {
    public void update();
}
```

As a result, we will have to change the model code so that it:

- keeps a list of all Observers that have registered
- informs all Observers whenever it changes (i.e., whenever its instance variables change)

We will also have to go through our interface code and:

- split up the view and controller into two separate classes
- remove all the calls to **update()** from the controller (except maybe one for initializing)

The Model:

Let us take a look at how the model now looks. Notice the changes highlighted:

```
// Making the TrafficLight model implement the Subject interface allows
// it to inform all of the observers whenever there has been a change
import java.util.ArrayList;
public class TrafficLight implements Subject {

    public static final int  MAX_TIME_COUNT = 8;

    private int  currentState;  // 1=red, 2=yellow, 3=green
    private int  stateCount;    // amount of time in this state

   ArrayList<Observer>     observers = new ArrayList<Observer>();

    // Constructor that makes a red traffic light
    public TrafficLight() {
        currentState = 1;
        stateCount = 0;
    }

    // Advance the traffic light to the next state
    public int advanceState() {
        currentState = ++currentState % 3 + 1;
        stateCount = 0;

       updateObservers(); // Tell the observer applications about this change

        return currentState;
    }

    // Simulate a single time unit of time passing by
    public void advanceTime() {
        // The number of seconds (i.e., time units) that
```

```
        // the traffic light remains in each state
        int[] stateTimes = {6, 3, 8};

        // advance the time spent in the current state
        stateCount++;

       updateObservers(); // Tell the observer applications about this change

        // Check if we have reached time limit for current state
        if (stateCount > stateTimes[currentState-1])
            advanceState();
    }

    // Return the amount of time spent in the current state
    public int getStateCount() {
        return stateCount;
    }

    // Return the state of the traffic light (as a number from 1 to 3)
    public int getState() {
        return currentState;
    }

    // Set the state of the traffic light (as a number from 1 to 3)
    // If the integer is out of range, do nothing
    public void setState(int newState) {
        if ((newState > 0) && (newState <4)) {
            currentState = newState;
            stateCount = 0;

           updateObservers(); // Tell the observer applications about this change

        }
    }

    // Return a string representation of the traffic light
    public String toString() {
        String[] colours = {"Red", "Yellow", "Green"};
        return colours[currentState] + " Traffic Light";
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void unregisterObserver(Observer observer) {
        observers.remove(observer);
    }

    // This method is called whenever there is a change to the model.
    // It informs all registered observer applications of the change.
    private void updateObservers() {
        for (Observer anObserver: observers)
            anObserver.update();
    }


}
```

The View:

What portion of code represents the view ? All of the stuff related to adding Frames/Panels/Components etc.. We can take the **TrafficLightFrame** class and "strip away" all of the behaviour and model related stuff (i.e., remove the Listeners etc..) If we do this, then the controller MUST add the behaviour-related stuff (i.e., event handlers, updates, listeners etc..).

One problem is that the controller must be able to access the view's components in order to add listeners, get their contents, change them etc.. One solution to this problem is to make instance variables for all the components and then supply **public** "**get**" methods for each.

We will make our views as separate **JPanels** that hold the entire contents of the window:

```
import java.awt.*;
import javax.swing.*;
public class TrafficLightPanel extends JPanel {

    // These are the components
    private JRadioButton[]   buttons = new JRadioButton[3];
    private JButton          advButton;
    private JProgressBar     progressBar;
    private JSlider          slider;
    private JComboBox        actionList;
    private JCheckBox        autoButton;

    // Make some get methods so that the controller can access this view
    public JRadioButton  getButton(int i)   { return buttons[i]; }
    public JButton       getAdvanceButton() { return advButton; }
    public JProgressBar  getProgressBar()   { return progressBar; }
    public JSlider       getSlider()        { return slider; }
    public JComboBox     getActionList()    { return actionList; }
    public JCheckBox     getAutoButton()    { return autoButton; }

    public TrafficLightPanel() {
        GridBagLayout         layout = new GridBagLayout();
        GridBagConstraints    constraints = new GridBagConstraints();
        setLayout(layout);

        // Add all the labels
        JLabel label = new JLabel("Manual");
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.weightx = 0;
        constraints.weighty = 0;
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.NORTHWEST;
        constraints.insets = new Insets(5, 5, 0, 0);
        layout.setConstraints(label, constraints);
        add(label);

        label = new JLabel("Action");
        constraints.gridx = 1;
        layout.setConstraints(label, constraints);
        add(label);

        label = new JLabel("Advance");
        constraints.gridx = 2;
        layout.setConstraints(label, constraints);
        add(label);

        label = new JLabel("Timer Progress");
        constraints.gridx = 0;
        constraints.gridy = 4;
        layout.setConstraints(label, constraints);
        add(label);


        // Add the Radio Buttons
        ButtonGroup lights = new ButtonGroup();
        JPanel aPanel = new JPanel();
        aPanel.setLayout(new BoxLayout(aPanel, BoxLayout.Y_AXIS));
        aPanel.setBackground(Color.black);
        for (int i=0; i<3; i++) {
            buttons[i] = new JRadioButton("", false);
            buttons[i].setBackground(Color.black);
            lights.add(buttons[i]);
            aPanel.add(buttons[i]);
        }
        buttons[0].setText("Red");
        buttons[1].setText("Yellow");
        buttons[2].setText("Green");
        buttons[0].setForeground(Color.red);
        buttons[1].setForeground(Color.yellow);
        buttons[2].setForeground(Color.green);
```

```
            constraints.gridx = 0;
            constraints.gridy = 1;
            constraints.gridheight = 3;
            constraints.fill = GridBagConstraints.BOTH;
            layout.setConstraints(aPanel, constraints);
            add(aPanel);

            // Make the Actions List
            String[] actions = {"Stop", "Yield", "Go"};
            actionList = new JComboBox(actions);
            constraints.gridx = 1;
            constraints.gridy = 1;
            constraints.gridheight = 1;
            constraints.weightx = 1;
            constraints.fill = GridBagConstraints.HORIZONTAL;
            layout.setConstraints(actionList, constraints);
            add(actionList);

            // Make the Slider
            slider = new JSlider(JSlider.HORIZONTAL, 0, 20, 1);
            slider.setMajorTickSpacing(5);
            slider.setMinorTickSpacing(1);
            slider.setPaintTicks(true);
            slider.setPaintLabels(true);
            constraints.gridx = 1;
            constraints.gridy = 3;
            layout.setConstraints(slider, constraints);
            add(slider);

            // Add the auto checkbox button
            autoButton = new JCheckBox("Auto Advance");
            constraints.gridx = 1;
            constraints.gridy = 2;
            constraints.fill = GridBagConstraints.BOTH;
            layout.setConstraints(autoButton, constraints);
            add(autoButton);

            // Add the Advance Picture button
            advButton = new JButton(new ImageIcon("RedLight.jpg"));
            constraints.gridx = 2;
            constraints.gridy = 1;
            constraints.gridheight = 3;
            constraints.weightx = 0;
            constraints.weighty = 1;
            constraints.fill = GridBagConstraints.BOTH;
            constraints.insets = new Insets(5, 5, 0, 5);
            layout.setConstraints(advButton, constraints);
            add(advButton);

            // Add the progress bar
            progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0,
    TrafficLight.MAX_TIME_COUNT);
            constraints.gridx = 0;
            constraints.gridy = 5;
            constraints.gridwidth = 3;
            constraints.gridheight = 1;
            constraints.weightx = 1;
            constraints.weighty = 2;
            constraints.fill = GridBagConstraints.BOTH;
            constraints.insets = new Insets(5, 5, 5, 5);
            layout.setConstraints(progressBar, constraints);
            add(progressBar);
        }
    }
```

Notice the following:

- This "view" code has no listener stuff, nor event handlers nor update methods
- We do not need to import the event packages anymore
- All of the instance variables related to the **Timer** have been removed (since this is behaviour related)
- There is no main method

The Controller:

What portion of code represents the controller ? All of the stuff related to adding listeners, event handlers, update methods etc.. as well as any behaviour-related code such as the **Timer** code. All of the stuff that we removed from the **View** must be added here. We will put it all in the **TrafficLightFrame** class that will serve as a *Mediator* between the model and the view. It will also contain the "main" method which will be responsible for coordinating the startup of the application. The controller will keep hold of the model and the view (as instance variables) as part of this coordination.

Here is the code:

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

public class TrafficLightFrame extends JFrame implements Observer {

    private TrafficLight        aTrafficLight;
    private Timer               aTimer;

   // This is the view
   private TrafficLightPanel   aView;


    private ActionListener    comboBoxListener;

    public TrafficLightFrame(String title, TrafficLight model, TrafficLightPanel
view) {
        super(title);

        aTrafficLight = model;

       aView = view;
       setContentPane(aView); //Replace old
     panel with ours

        // Add the Listeners
        for (int i=0; i<3; i++)
            aView.getButton(i).addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    handleRadioButtonPress((JRadioButton)e.getSource());
                }
            });

        aView.getActionList().addActionListener(comboBoxListener = new
ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleComboBoxSelection((JComboBox)e.getSource());
            }
        });
        aView.getSlider().addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                handleSlider((JSlider)e.getSource());
            }
        });
        aView.getAutoButton().addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleAutoButtonPress((JCheckBox)e.getSource());
            }
        });
        aView.getAdvanceButton().addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleAdvanceButtonPress();
            }
        });

        // Add a timer for automode.  Set it to go off every 500 milliseconds
        aTimer = new Timer(500, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleTimerTick();
            }
        });

       // !!! IMPORTANT !!!
```

```
        // Register with the model so that when it changes, we get
       informed
         aTrafficLight.registerObserver(this);

         setDefaultCloseOperation(EXIT_ON_CLOSE);
         setSize(400, 250);
     }


     // This is the Timer event handler
     private void handleTimerTick() {
         aTrafficLight.advanceTime();
         // update() has been removed now
     }

     // This is the Advance button event handler
     private void handleAdvanceButtonPress() {
         aTrafficLight.advanceState();
         // update() has been removed now
     }

     // This is the Auto button event handler
     private void handleAutoButtonPress(JCheckBox source) {
         if (source.isSelected())
             aTimer.start();
         else
             aTimer.stop();
         // update() has been removed now
     }

     // This is the radio button event handler
     private void handleRadioButtonPress(JRadioButton source) {
         for (int i=0; i<3; i++) {
             if (source == aView.getButton(i))
                 aTrafficLight.setState(i+1);
         }
         // update() has been removed now
     }

     // The ComboBox Selection event handler
     private void handleComboBoxSelection(JComboBox source) {
         aTrafficLight.setState(source.getSelectedIndex() + 1);
         // update() has been removed now
     }

     // This is the Slider event handler
     private void handleSlider(JSlider source) {
         if (!source.getValueIsAdjusting()) {
             int delay = source.getValue();
             if (delay > 0) {
                 aTimer.setDelay(1000/delay);
                 if (aTimer.isRunning())
                     aTimer.restart();
             }
             else {
                 aTimer.setDelay(Integer.MAX_VALUE);
             }
             // update() has been removed now
         }
     }

     // Update the radio buttons according to the traffic light state
     private void updateRadioButtons() {
         for (int i=0; i<3; i++) {
             aView.getButton(i).setSelected(aTrafficLight.getState() == (i+1));
         }
     }

     // Update the status pane according to the traffic light state
     private void updateAdvanceButton() {
         String[] iconNames = {"RedLight.jpg","YellowLight.jpg","GreenLight.jpg"};
         aView.getAdvanceButton().setIcon(new
ImageIcon(iconNames[aTrafficLight.getState()-1]));
     }
```

```
        // Update the combo box according to the traffic light state
        private void updateComboBox() {
            aView.getActionList().setSelectedIndex(aTrafficLight.getState() - 1);
        }


        // Update the progress bar
        private void updateProgressBar() {
            aView.getProgressBar().setValue(aTrafficLight.getStateCount());
        }

        // Update all relevant components according to the traffic light state
        public void update() {
            aView.getActionList().removeActionListener(comboBoxListener);
            updateRadioButtons();
            updateComboBox();
            updateAdvanceButton();
            updateProgressBar();
            aView.getActionList().addActionListener(comboBoxListener);
        }

        public static void main(String args[]) {
            TrafficLight aModel = new TrafficLight();

            // Instantiate three controllers each with their own views
            // Two controllers (A and B) will share the same model while
            // the 3rd will be alone with its own model.
            new TrafficLightFrame("Traffic Light A (tied with B)", aModel,
                                  new TrafficLightPanel()).setVisible(true);
            new TrafficLightFrame("Traffic Light B (tied with A)", aModel,
                                  new TrafficLightPanel()).setVisible(true);
            new TrafficLightFrame("Traffic Light C (alone)", new TrafficLight(),
                                  new TrafficLightPanel()).setVisible(true);
        }
    }
```

5 Recursion

What's in This Set of Notes?

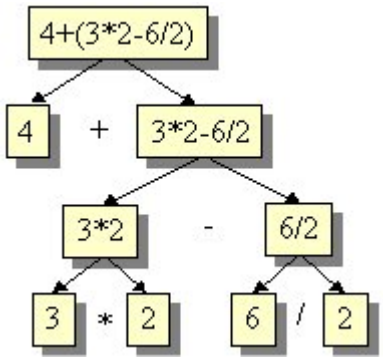
We will now take a short break from GUI design and look at an important programming style/technique known as **Recursion**. In nature (and in mathematics), many problems are more easily represented (or described) using the notion of recursion. It is often the case that we program recursively so as to provide a simpler and more understandable solution to the problem being implemented. In fact, many data structures used in computer science are inherently recursive, making recursive programming natural and often efficient.

Here are the individual topics found in this set of notes (click on one to go there):

5.1 What is Recursion ?

One of the most common techniques used in computer science is known as "divide and conquer". This technique represents a strategy of dividing a complex problem into smaller, easier-to-solve sub problems. There are many real-world examples of how we break problems into smaller ones to solve them:

1. Jigsaw puzzles are solved in "steps": border, interesting portion, grass, sky, etc..
2. Math problems are broken down into smaller/simpler problems
3. Even climbing stairs eventually breaks down to climbing one step at a time.



So then what does this have to do with recursion ? Well, recursion applies the divide and conquer strategy. The word ***Recursion*** actually comes from a Latin word meaning "a running back". This makes sense because recursion is the process of actually "going off" and breaking down a problem into small pieces and then bringing the solutions to those smaller pieces back together to form the complete solution. Here are some points to remember:

- recursion **breaks down** a complex problem into **smaller sub-problems**
- subproblems are **smaller** instances of the **same type** of problem.

It sounds a little bit abstract doesn't it ? Why would we want to do this anyway ?

- some problems are **naturally** recursive
  - e.g., especially math problems/functions such as **factorial**
- simpler, **more elegant solutions** are often obtained
- **easier to understand** completed solutions
- can be **only way to approach** a seemingly **overwhelming problem**

Do we really need recursion ? Well, any problem that is solved recursively can also be done without recursion, but usually the solution is more complex and it is difficult to consider all special cases.

So, recursion is all about:

1. figuring out how to break the problem down into smaller sub-problems
2. handling the smaller sub-problems
3. figuring out how to merge the results of the smaller sub-problems to answer the original problem

In fact, its actually easier than this sounds. Most of the time, we simply take the original problem and break/bite off a small piece that we can work with. We simply keep biting off the small pieces of the problem, solve them, and then merge the results.

It is important to remember the following very important facts about the sub-problems:

- must be an instance of the **same kind** of problem
- must be **smaller** than the original problem

So how many **times** do we bite off the small pieces ? When do we know when to stop ?

Its simple. We stop when there are no more pieces ... or when the remaining piece is so simple, that it is easily solved without needing to break it down any further. At this "lowest level", we call this simplest problem the "***base case***" or "***basis case***".

In fact, when writing our code, we will usually start with the base cases since they are the ones that

we know how to handle. For example, if we think of our "real world" examples mentioned earlier, here are the base cases:

1. For the jigsaw puzzle, we divide up the pieces until we have just a few (maybe 5 to 10) pieces that form an interesting part of our picture. We stop dividing at that time and simply solve (by putting together) the simple base case problem of this small sub picture. So the base case is the problem in which there are only a few pieces all matching together.

2. For the math problem, we simply keep breaking down the problem until either:

   a) there is a simple expression remaining (e.g., 2 + 3) or
   b) we have a single number with no operations (e.g., 7).

   These are simple base cases in which there is either one operation to do or none.

3. For the stair climbing problem, our base case is our simplest case ... when there is only one stair. We simply climb that stair.

Tips for Designing Recursive Methods:

1. Decide on a name for the method and what its ***parameters*** should be; you have to do this for non-recursive methods just as well.

2. You must ***believe that the method will work*** before you even begin to implement it; this is important. Without it, you will not have the faith to use the method to solve a simple problem especially if the method isn't finished.

3. You must decide on the ***simple cases*** that can be implemented trivially (the basis cases) and then write the code.

4. Determine a technique for breaking up the more complicated case into simpler parts, some of which can be done by using the original method with simpler parameters. This is usually the hard part. ***Don't think about the recursion***. Just think about how to express the original problem in terms of the smaller one's solution.

It would be a good idea to read the above tips again after you have tried writing a couple of recursive methods.

5.2 Recursion With Primitives

We now need to start looking at some examples of using recursion. We will first consider examples in which the recursion occurs without needing to manipulate any objects. That is, we will look at a couple of examples in which recursion is used to compute some values. The simplest example is that of using the factorial function. In fact, the factorial example is the "hello world" example of recursion since the factorial function is a very natural and simple operation that is inherently recursive. By now, most of you know what the factorial operation does:

5! = 5*4*3*2*1
4! = 4*3*2*1
3! = 3*2*1
2! = 2*1
1! = 1
0! = 1

The operation is defined non-recursively as follows:

```
1                                          if N = 0
N! = N x (N-1) x (N-2) x ... 3 x 2 x 1     if N >= 1
```

We can easily write a **factorial()** method that takes an integer and computes the factorial using loops.

```
// A non-recursive method for computing the factorial
public static int factorial(int num) {
    int  result = 1;
    for (int i=2; i<=num; i++)
        result *= i;
    return result;
}
```

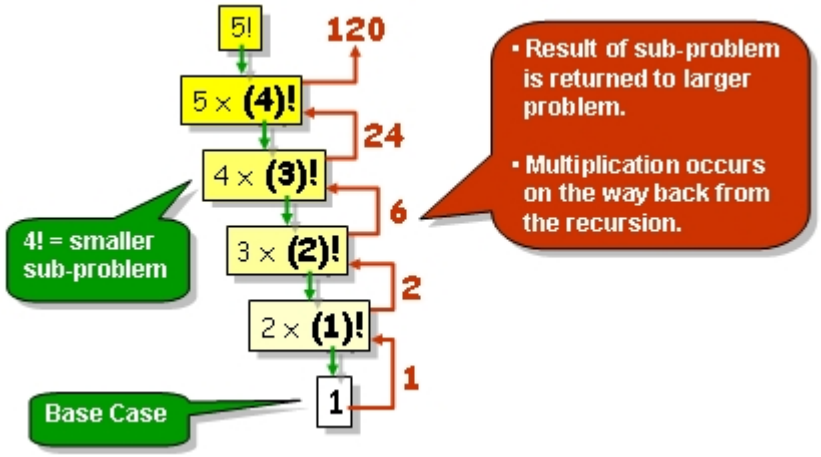
How could we write this code recursively ? First, we must figure out how to express the problem recursively. In this case it is easy since there is a well-known recursive definition:

```
1                 if N = 0
N! = N x (N-1)!   if N >= 1
```

Notice here that **N!** is defined in terms of a smaller factorial problem ... that of **(N-1)!**. So we in fact reduce our initial number N by 1. Then how do we solve the (N-1)! problem ? Well, just apply the same formula:

```
(N-1)! = (N-1) x ((N-1)-1)! = (N-1) x (N-2)!
```

So then we need to break down (N-2)! ... which is done in the same way. Eventually, as we keep reducing **N** by 1 each time, we end up with **N=0 or 1** and for that simple problem we know the answer is 1. So breaking it all down we see the solution of **5!** as follows:



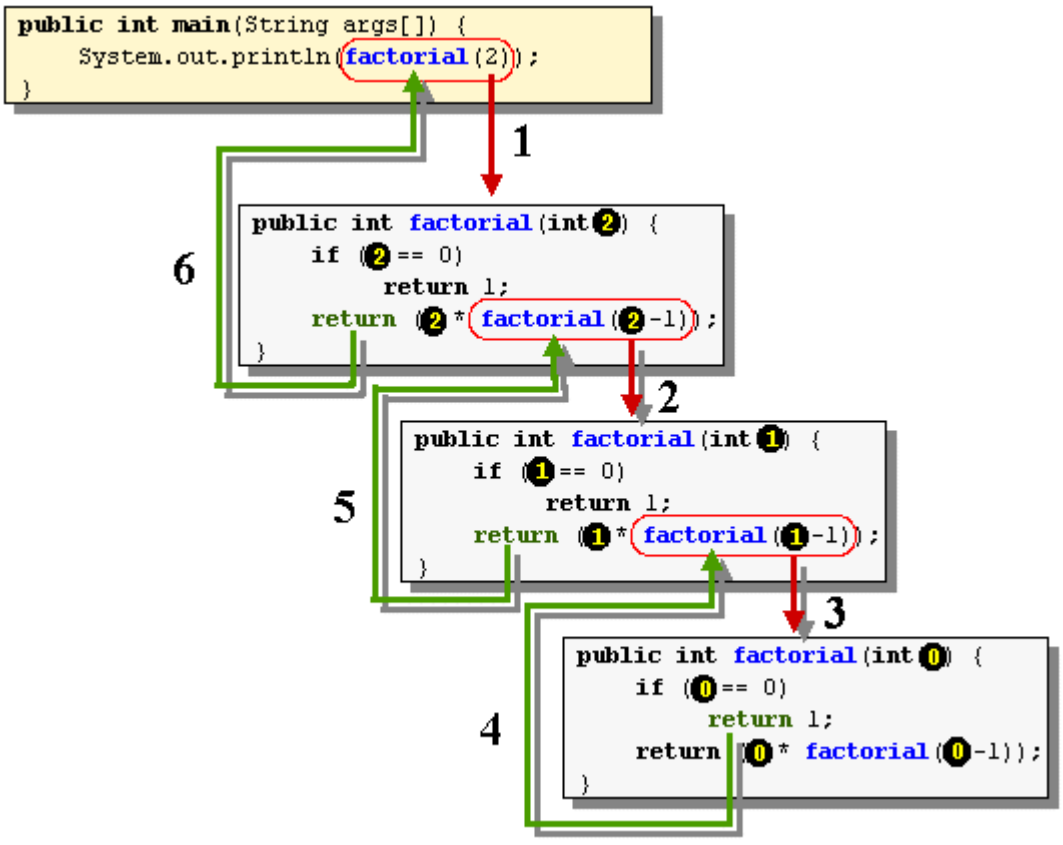
So how do we write the code ? Its easy. You start with the base case. Then use the formula to break down the problem:

```
public static int factorial(int n) {
    if (n == 0)  // BASE CASE
        return 1;
    return n * factorial(n - 1);  // RECURSIVE STEP
}
```

Wow! That's a simpler solution than the non-recursive version. Did you notice how the method actually calls itself ?

Recursive methods actually work just like your "normal everyday" methods. There is nothing tricky. In essence, each time a message is sent, a new "copy" of the method runs on a new copy of the parameters. The local variables of one method invocation are not visible to another invocation.

Consider tracing the recursive execution of "2 factorial":

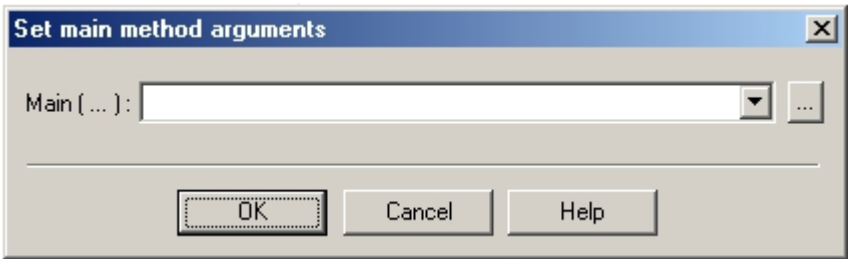


Below is the code as a test application. In it, we will make use of something new called ***command-line arguments***.

Command-line arguments:

- are specified when running a JAVA program
- provide some additional information similar to the notion of parameters.
- are passed as String parameters to the **main** method, stored in **`String args[]`**.
- we can ask **`args`** for its length to see if there are any command line arguments.
- you may not use spaces in your String parameters since a space character is used to separate the parameters.

In JCreator, we can set the command line arguments by going to the **Configure**/**Options** menu and selecting **JDK Tools**. Than **select the tool type** to be **Run Application**. Select **<Default>**, then press **Edit**. Click on the **Parameters** tab and then select the option "**Prompt for main method arguments**". Press OK twice and then run your application. Now when you run, you will be asked to enter some text for your application:



We will run our code the first time with the value of 5 as the command line argument. So type 5 into the text field. You should get a result of 120. Here is the code:

```
public class FactorialTest {

    public static int factorial(int n) {
        if (n == 0)                     // BASE CASE
            return 1;
        return n * factorial(n - 1);    // RECURSIVE STEP
    }

    public static void main(String[] args) {
        //First check to see that there is at least one command line
argument
        if (args.length == 0) {
            System.out.println("Usage: FactorialTest <anInteger>");
            System.exit(-1);
        }
        int theInteger = Integer.parseInt(args[0]);

        if (theInteger < 0) {
            System.out.println("Factorial of a negative int is not
defined");
            System.exit(-1);
        }
        else
            System.out.println("Factorial of " + theInteger + " is " +
factorial(theInteger));
    }
}
```

Did you notice how we can check the length of the **args** array to ensure that there is at least one argument ? We then accessed that argument by accessing the String array **args** at its first position (which is the first parameter, which was the number 5 we entered). Be aware though that the value

comes in as a String, so we have to parse it into the desired type.

What important test cases are not covered by the code above ? Try entering 16, 17, 100. Are there any problems ? Do you know why ?

Example (Mortgage Payment Calculator):

Consider a second recursion example that computes how much money per month a person would have to pay to pay back a mortgage on his/her home. Consider the following notation:

- **p** = the principal cost/amount of the home (e.g., $130,000)
- **i** = the annual interest rate as a percentage (e.g., 3.5%)
- **t** = the term (in months) that we wish the mortgage to be for (e.g., 300 months for a 25 year mortgage)
- **m(p, i, t)** = the monthly payments we need to make based on the parameters just mentioned (e.g., $647.57)

We would like to determine the value for **m**. Let us assume that **t > 0**, otherwise the point of calculating a payment is silly anyway.
Certainly if the **t = 1** then we must pay one month interest and so the value of **m** should be **p*(1+i)** ... that is ... one month of interest. Otherwise, we will consider the following recursive formula to calculate the amount to be paid each month:

$$m(p, i, t) = \frac{p}{\left(\frac{p}{m(p, i, t-1)} + \frac{1}{(1+i)^t}\right)}$$

Now there are more accurate and efficient wasy to compute mortgage payments, but we will use the above formula so that we can practice our recursion.

So how do we write the code to do this ? Certainly the function we need to write requires 3 parameters:

```
public static double calculatePayment(int p, double i, int t) {
  ...
}
```

Now let us determine the base case. It actually follows right from the definition.

```
if (t == 1)
    return p*(1+i);
```

That wasn't so bad. Now what about the recursive part ? It also follows from the formula:

```
return p / ((p / calculatePayment(p,i,t-1)) + (1/Math.pow(1+i, t)));
```

So here is the whole thing:

```
public static double calculatePayment(int p, double i, int t) {
    if (t == 1)
```

```
            return p*(1+i);

        return p / ((p / calculatePayment(p,i,t-1)) + (1/Math.pow(1+i, t)));
    }
```

Does it work ? We should write a test method. Here is a class with our method, along with a main method that reads the command line arguments:

```
public class MortgagePaymentCalculator {
    public static double calculatePayment(int p, double i, int t) {
        if (t == 1)
            return p*(1+i);

        return p / ((p / calculatePayment(p,i,t-1)) + (1/Math.pow(1+i,
t)));
    }

    public static void main(String[] args) {
        //First check to see that there is at least three command line
arguments
        if (args.length < 3) {
            System.out.println("Usage: MortgagePaymentCalculator
<principal> <interest rate> <term>");
            System.exit(-1);
        }
        int principal = Integer.parseInt(args[0]);
        double intRate = Double.parseDouble(args[1]);
        int term = Integer.parseInt(args[2]);

        System.out.printf("The monthly mortgage payment for a %d month
mortgage of $%,d " +
                          "home at %1.3f percent annual interest is $%4.2f
per month.",
                          term, principal, intRate,
calculatePayment(principal, intRate/100.0/12.0, term));
    }
}
```

Notice that the user enters the interest rate as an annual rate (e.g., 3.5) and the term is in months. We then need to adjust the interest rate to be a percentage (hence divide by 100) and also to make it monthly to match the term units (hence divide by 12).

So from our two examples, you can see that recursion is quite simple once you have a recursive formula.

5.3 Recursion With Objects (Non-Destructive)

Now that we have seen some simple examples of recursion that dealt only with some primitive calculations, we need to look at how recursion works when objects are involved. Let us look at a simple example of reversing a string.

Example (Reversing a String):

As you may know, strings in JAVA are not mutatable (that is, you cannot actually modify a string once it is made). We will look at a method that takes a string and then returns a new string which has the same characters as the original, but in reverse order:


```
public static String reverse(String s) {
    ...
}
```

We start by considering the base case. What strings are the easiest ones to solve this problem for ? Well, a string with 1 or 0 characters is easy, since there is no reversing to do. So there you have it. Those are the base cases:

```
if (s.length() == 1)
    return s;

if (s.length() == 0)
    return s;
```

We can simplify this to one line:

```
if (s.length() <= 1) return s;
```

Now, how do we express the problem recursively ? Remember ... think of a smaller problem of the same type. A smaller problem would be a smaller string. So what if we take a piece of the string and then solve for the smaller string ? That is the way we should approach it. What piece should we take off ? Perhaps the first character. We can use:

```
s.substring(1, s.length())
```

to get the smaller string which is the original without the first character.

OK. Now what do we do ? Remember, we need to express the solution to original problem in terms of the smaller problem. So we should be thinking the following question: "If I have the reverse of the smaller string, how can I use that to determine the reverse of the whole string ?". Lets look at an example:

"STRING" reversed is "GNIRTS". If we use consider the shorter string "TRING" and reverse it to "GNIRT", then how can we use this reversed shorter string "GNIRT" to obtain the solution "GNIRTS" to the original problem ? You can see that all we have to do is append the "S" to the smaller string solution "GNIRT" to get the complete solution "GNIRTS". So,

```
reverse("STRING") = reverse("TRING") + "S"
```

Now we should know how to write the code:

```
public static String reverse(String s) {
    if (s.length() <= 1) return s;

    return reverse(s.substring(1, s.length())) + s.charAt(0);
}
```

As you can see the solution is quite simple ... **after** we see the solution of course ;).

Here is the test code:

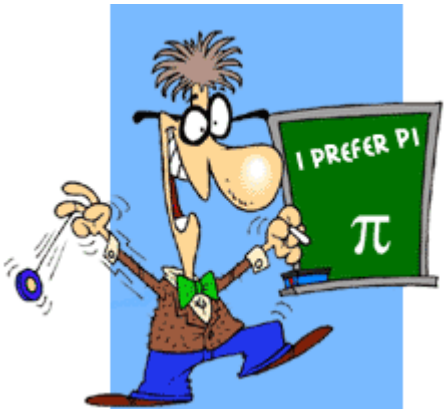
```
public class ReverseTest {
    public static String reverse(String s) {
        if (s.length() <= 1) return s;
```

```
        return reverse(s.substring(1, s.length())) + s.charAt(0);
    }

    public static void main(String[] args) {
        //First check to see that there is at least one command line
argument
        if (args.length == 0) {
            System.out.println("Usage: ReverseTest <aString>");
            System.exit(-1);
        }
        System.out.println(args[0] + " reversed is " + reverse(args[0]));
    }
}
```

Example (Palindromes):

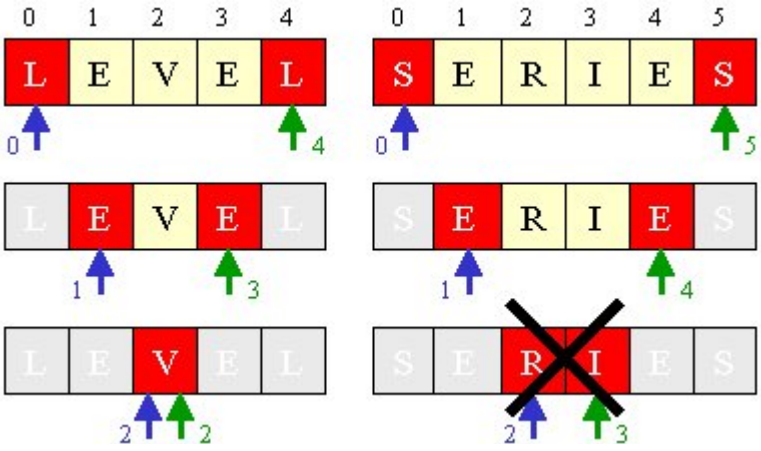
Consider the simple problem of determining whether or not a String is a palindrome. A ***palindrome*** is a String which reads the same forwards as backwards:

- level
- noon
- mom
- madam



How do we write a method using a **for** loop to detect whether or not a String is a palindrome ? Well ... how do we do it without a computer ?

We probably compare the first and last characters and then work our way inward toward the center of the String:



Here is how we may write code to do this:

```
public static boolean isPalindrome(String s) {
    for (int i=0; i<=(s.length()-1)/2; i++) {
        if (s.charAt(i) != s.charAt(s.length() - i - 1))
            return false;
    }
    return true;
}
```

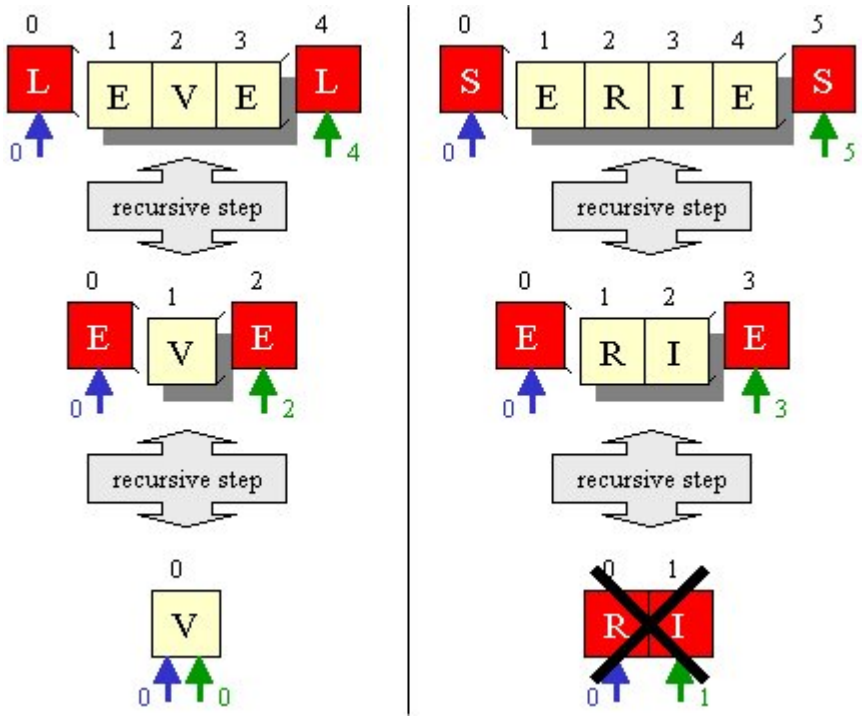
Now what about solving this recursively ?

We must be able to express the palindrome problem in terms of smaller palindrome problems.

Here is the recursive formulation of the problem:

- A String is a palindrome if its first and last characters are identical AND the substring in between is a palindrome.

Here is how the recursion can be done:



For example, in our palindrome problem, when only 1 character remains in the String, or in the case where the String is empty, we don't need to divide any further and so these are the base cases for our problem. Often, the base cases also correspond similarly with error-checking (e.g., like checking for an empty string first). So ... the ***base case*** is used to "**stop**" the recursive process.

Here are some palindrome examples that show when the base case is reached.

```
isPalindrome("level")  ---> isPalindrome("eve")  --->  isPalindrome("v")  --->
true
isPalindrome("poop")  ---> isPalindrome("oo")  --->  isPalindrome("")  ---> true
isPalindrome("abcdba")  ---> isPalindrome("bcdb")  --->  isPalindrome("cd")
---> false
```

So how do we write the code for this problem recursively ? Think of the stopping conditions and write pseudo code:

- if the string is empty return true (i.e., an empty string is considered a palindrome)
- if the string has only 1 character in it, return true
- otherwise ... if the first and last characters do not match, return false
- otherwise return the result of the recursive sub-problem on a substring which excludes the first and last characters

Start the code with the base cases, then do the recursive part:

```
public class StringUtilities {

    public static boolean isPalindrome(String s) {
        //BASE CASE (1 or 0 character cases combined together here)
        if (s.length() <= 1)
            return true;
```

```
            //BASE CASE (first and last characters do not match)
            if (s.charAt(0) != s.charAt(s.length() - 1) )
              return false;

            //RECURSIVE STEP (check if middle portion is a palindrome)
            return isPalindrome(s.substring(1, (s.length() - 1)));
        }
    }
```

Here is another way to write it:

```
    public static boolean isPalindrome(String s) {
        return ((s.length() <= 1) ||
                ((s.charAt(0) == s.charAt(s.length() - 1)) &&
                 (isPalindrome(s.substring(1, (s.length() - 1))))));
    }
```

Now we must write a test application. Here is some testing code written in a different class:

```
public class PalindromeTest {
    public static void main(String[] args) {
        //First check to see that there is at least one command line argument
        if (args.length == 0) {
            System.out.println("Usage: PalindromeTest <aString>");
            System.exit(-1);
        }
        String input = args[0]; // Access the first argument from the argument
list

        if (StringUtilities.isPalindrome(input))
            System.out.println(input + " is a palindrome ");
        else
            System.out.println(input + " is NOT a palindrome " );
    }
}
```

5.4 Recursion With Objects (Destructive)

In the previous section, we showed how we could break pieces off of Strings each time we called the method recursively. In fact, we did not really alter the string, since the **substring()** method actually returns a new string. As a result, we did not really destroy the original string. Sometimes, however we might want to use a destructive modification of an object. For example we might want the actual elements of an object to be altered, instead of making a new object that represents the changes to the original.

- ***Destructive methods*** - alter the receiver or parameters in some way to obtain result.
- ***Non-destructive methods*** - usually creates new objects to contain the "answer" to the problem, leaving the receiver and/or parameters intact.

Often, it is easier to destroy an object when performing an operation. For example, if someone asked you to count the jelly beans in a big jar you would probably not leave the jar intact. Instead, you'd alter it by removing the jelly beans one at a time (or in small amounts) and do the counting while placing the counted jelly beans in a new and initially empty jar (or eating them !). To do this completely non-destructively, you'd have to count the beans without taking them out of the jar....a difficult task indeed.

In the case where the object is temporarily destroyed during the computation and then restored at the end, this is considered to be ***Non-Destructive***. For instance, we may temporarily remove the jelly beans from the jar to count them and then put them back in when done. From the outsiders point of view, the jar arrangement has not been destroyed. We do realize however that the order of the jelly beans has been altered. If the order was important, then this process is considered destructive, as we are destroying the ordering.

Another approach that is common is to take the original object, **make a copy** and then write a destructive method on the copy. This is kind of like "cheating" but it does solve the problem, of course with the added running time overhead of copying the original object beforehand. This often leads to the need to write more than one method (i.e., use helper methods) to solve the problem. We will see later that this brings up the notion of ***indirect recursion***.

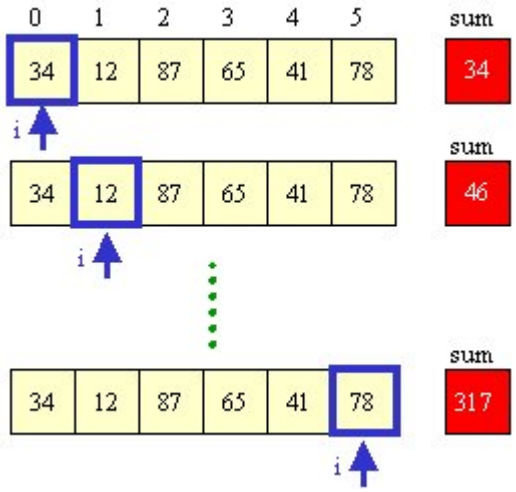
The notion of writing destructive or non-destructive methods is not something specific to recursion. In fact, you have already written some destructive AND non-destructive methods in COMP1405/1005.

Example (Summing Elements in an ArrayList)

Assume that we have an ArrayList of Integer objects. Consider writing a function that sums the integers in the ArrayList:

```
public static int sum(ArrayList<Integer>
anArrayList) {
    int sum=0;
    for(Integer x: anArrayList)
        sum += x;
    return sum;
}
```

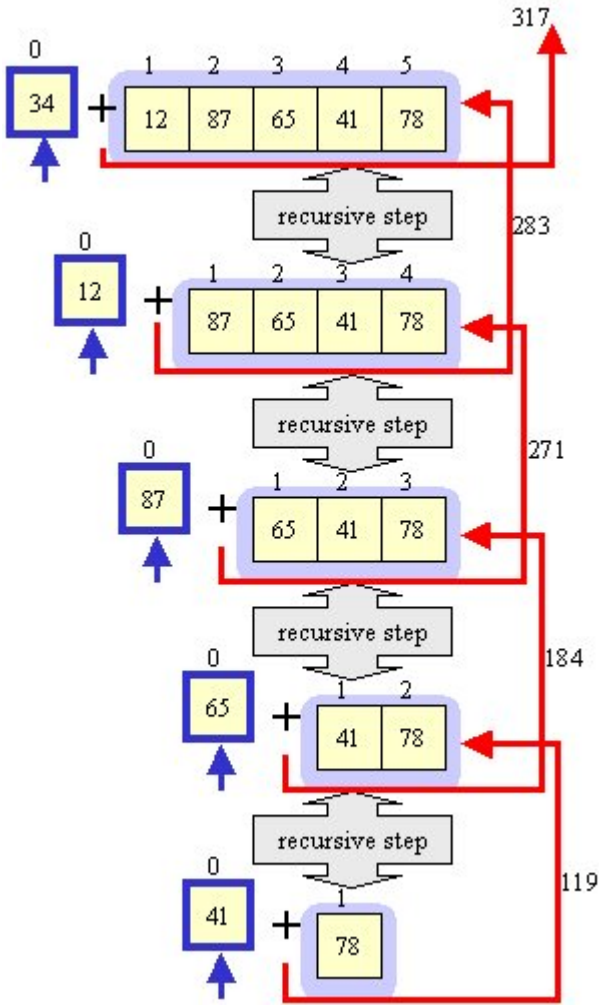
As you can see, we simply go through the elements one by one and compute the total:



What about doing it recursively ? How do we define the problem recursively ?

We can do this as follows:

- grab the first element and add it to the sum
- remove it from the ArrayList and recursively sum the remaining Integers.



Note this strategy is ***destructive*** in that it modifies the ArrayList by removing all of its components. Nevertheless, let us examine how this is done.

First, we consider the recursion by finding the simple base cases:

- if the ArrayList is empty, then the sum is 0
- if the ArrayList has only one element, then the sum is that element
- otherwise we must do some recursion

Here is the code:

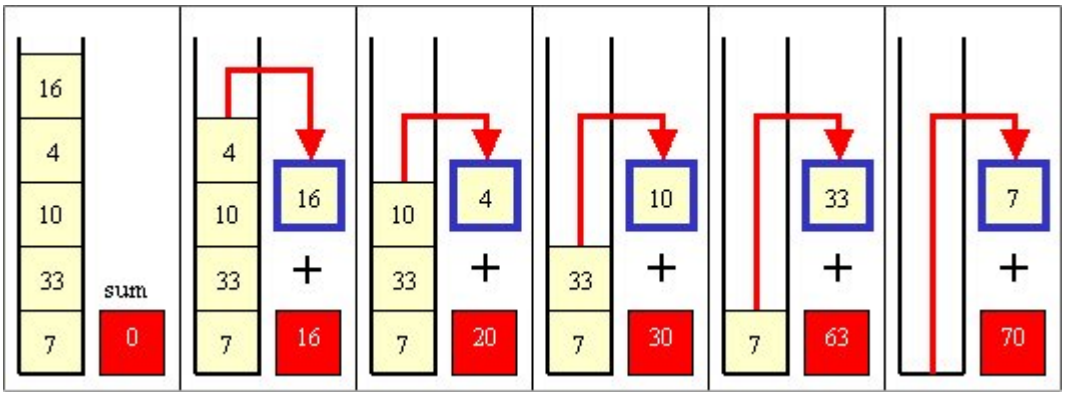
```
    public static int
sum(ArrayList<Integer> arrayList) {
        //BASE CASES
        if (arrayList.isEmpty())
           return 0;
        if (arrayList.size() == 1)
           return arrayList.get(0);

        //RECURSIVE STEP
        Integer element =
arrayList.get(0);
        arrayList.remove(element);
        return element.intValue() +
sum(arrayList);
    }
```

But wait. Think about what happens as we empty elements from the ArrayList and head towards the base case. Do we really need that second base case ? The answer is NO since one more step of the recursive case will bring us to the first base case. Hence, we can simplify the code by removing the second base case:

```
    public static int sum(ArrayList<Integer> arrayList) {
        if (arrayList.isEmpty()) return 0;

        Integer element = arrayList.get(0);
        arrayList.remove(element);
        return element.intValue() + sum(arrayList);
    }
```

This hardly seems more efficient than the non-recursive version!! As mentioned before, some problems are not meant to be done recursively. We are simply examining them recursively for practice in order to help us understand recursion.

In some cases, the recursive solution is simpler. Consider summing **Integer** objects that are in a **Stack**:

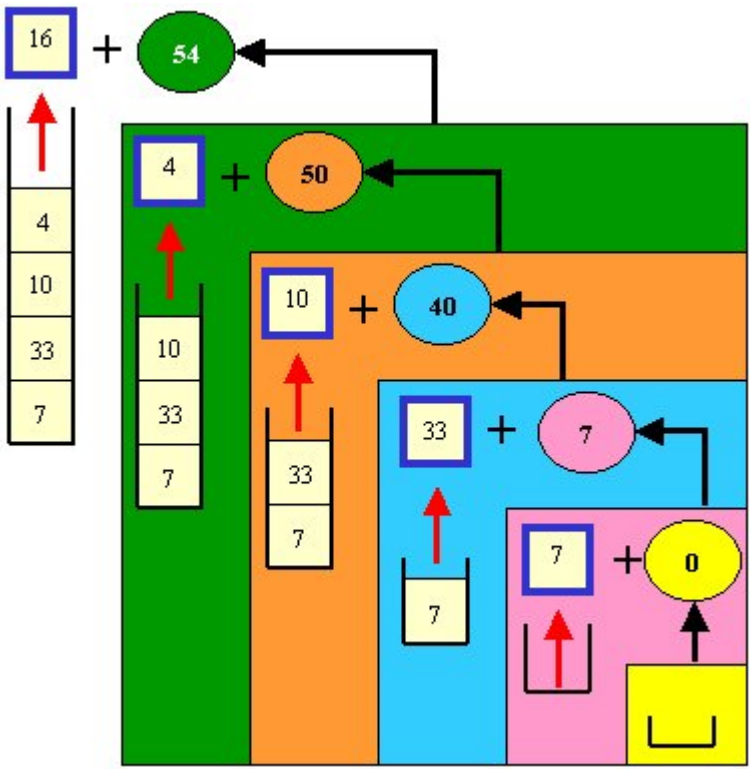



```
public static int sum(Stack<Integer> s) {
    int sum=0;
    while(!s.isEmpty())
        sum += s.pop();
    return sum;
}
```

Is this destructive ? Yes!

What does the recursive version look like ?

- The **sum**(s) = firstElementOf(s) + **sum**(s without the first element)



Here is the code:

```
public static int sum(Stack<Integer> s) {
    if (s.isEmpty()) return 0;
    return s.pop() + sum(s);
}
```

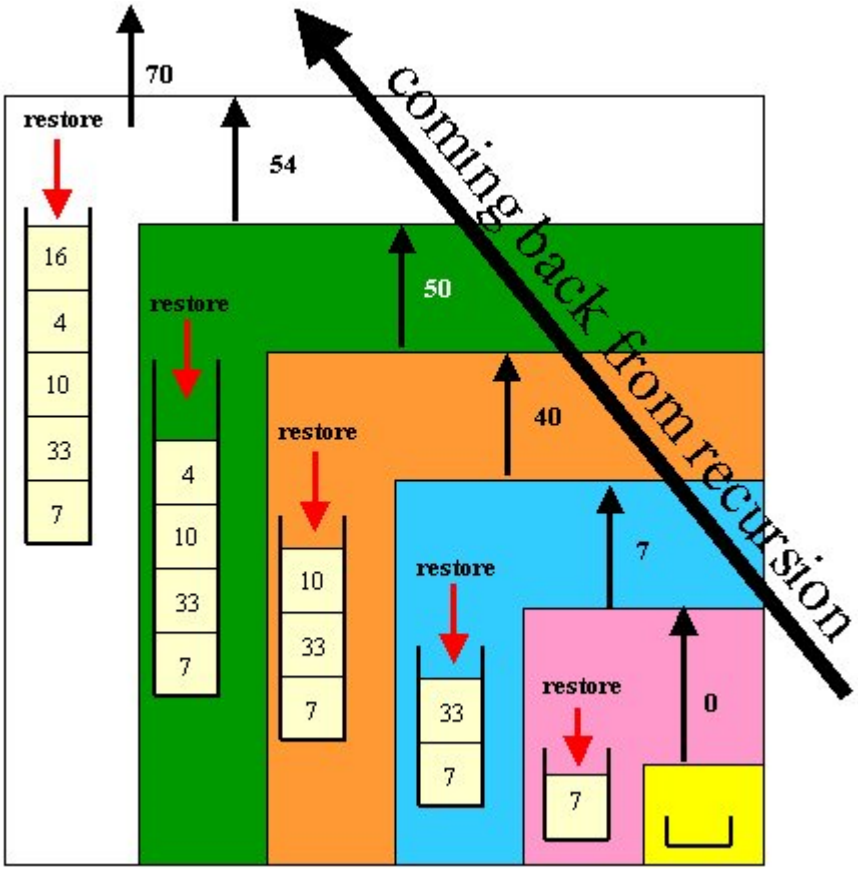
It is roughly the same amount of code. But now, what if we wanted to write these methods non-destructively ?
We need to put the items back onto the **Stack** so that the **Stack** is restored.
Here is the non-recursive, non-destructive version:

```
public static int sum(Stack<Integer> s) {
    int sum=0;
    Stack<Integer> tempStack = new Stack<Integer>();
    while(!s.isEmpty()) {
        Integer item = s.pop();
        sum += item;
        tempStack.push(item); // Keep backup of popped items
    }
    //Restore the original Stack
    while(!tempStack.isEmpty()) {
        s.push(tempStack.pop());
    }
    return sum;
}
```

Notice that we had to write code to keep track of the items that we removed so that we could put them back later. We used another **Stack** to keep this information, but we could have used any kind of collection.

Now what about the non-destructive recursive version ?

```
public static int sum(Stack<Integer> s) {
    if (s.isEmpty()) return 0;

    Integer element = s.pop();      // temporarily destroy stack
    int answer = element + sum(s);  // get recursive sum
    s.push(element);                // restore after the recursion
    return answer;
}
```



Hey! It is much simpler. We don't need extra variables. We simply restore the popped item on the way back from the recursion. This is an example of temporarily destroying the object in order to get an answer and then "undo"ing the destroying on the way back from the recursion. It is considered a non-destructive method. So we can see that the recursive solution is more elegant.

Example (Counting):

Consider counting the elements in some kind of collection. Perhaps we want to only count elements that satisfy some condition. In this example, we count the odd integers in an **ArrayList**. We will do it first destructively, and then alter our code to make it non-destructive.

Destructively, we can simply take off one element from the ArrayList each time through the recursion. We simply look at the number that we take out and add one to the total each time it is found to be odd.

Here is the destructive version inside a text class:

```
import java.io.*;
import java.util.*;

public class CountTest {

    public static int countOdd(ArrayList<Integer> nums) {
        //BASE CASE
        if (nums.isEmpty()) return 0;

        //RECURSIVE STEP
        Integer element = nums.get(0);
        nums.remove(element);

        if (element%2 == 0)
            return countOdd(nums);
        else
            return 1 + countOdd(nums);
    }

    public static void main(String[] args){
        int input = 0;
        ArrayList<Integer> nums = new ArrayList<Integer>();
        do {
            System.out.println("Enter integers one at a time (0 to end):");
            if ((input = new Scanner(System.in).nextInt()) != 0)
                nums.add(input);
        } while (input != 0);
        System.out.println("There were " + countOdd(nums) + " odd integers
entered.");
    }
}
```

How can we do it non-destructively ? We need to ensure that anything we remove is put back in on the way back from the recursion. Since we begin by removing the first element from the ArrayList and then doing the recursion, we must add that removed element back to the front of the ArrayList AFTER the recursion is done. If we do this at every step, the ArrayList should be back to normal. Notice the changes that we need to make:

```
public static int countOdd(ArrayList<Integer> nums) {
    //BASE CASE
    if (nums.isEmpty()) return 0;

    //RECURSIVE STEP
    Integer element = nums.get(0);
    nums.remove(element);

     int result = 0;
```

```
        if (element%2 == 0)
            result = countOdd(nums);
        else
            result = 1 + countOdd(nums);

        // Add the element back to the collection before
     returning
        nums.add(0, element);
        return result;
    }
```

In fact, we can simplify this code further, noticing that both conditions in the **IF** statement call recursively:

```
    public static int countOdd(ArrayList<Integer> nums) {
        //BASE CASE
        if (nums.isEmpty()) return 0;

        //RECURSIVE STEP
        Integer element = nums.get(0);
        nums.remove(element);

        int result = countOdd(nums);
        if (element%2 != 0) result++;

        // Add the element back to the collection before returning
        nums.add(0, element);
        return result;
    }
```

5.5 Direct vs. Indirect Recursion

A method that calls a recursive method is considered ***recursive*** itself. If the method calls itself, it is considered to be ***directly recursive***. In general, it is possible for two or more methods to call one another repeatedly. This is commonly termed "***mutual***" recursion. For example, method **A** calls method **B**, which calls method **A** again etc...

We use the term "**indirect**" recursion to describe a method which itself is not recursive, but which calls a directly recursive method to compute its solution. The **isPalindrome()** method that we wrote is considered **directly** recursive as it calls itself recursively.

Example (Summing Integers in an Array)

Let us look again at the example of summing integers, but this time using an array of ints. We cannot remove from the array as we did with the **ArrayLists** and **Stacks** in the previous section of the notes. We must take a different, non-destructive, approach. How would we do this non-recursively ?

```
    public static int sum(int[] theArray) {
        int result = 0;
        for (int i=0; i<theArray.length; i++)
            result += theArray[i];
        return result;
```

```
    }
```

Now this is ***non-destructive*** because it merely accesses the array elements without removing or changing them.

How can we do this recursively ? At each step of the recursion, we will need to know which number we are adding, so we will also need an index. Where do we put this index ? It MUST be a parameter to the method so that this position needs to be carried through the recursive iterations:

```
    private static int sum(int[] theArray, int n) {
        //BASE CASE: sum(theArray,0) = a[0]
        if (n == 0) return theArray[0];

        //RECURSIVE STEP: sum(theArray,n) = theArray[n] + sum(theArray,n-1)
        return (theArray[n] + sum(theArray,n-1));
    }
```

Notice that this method adds the elements in reverse order. That is different from the non-recursive method. It is more natural this way since we continually reduce the value of **n** until we reach the base case of **n == 0**. Note that we did not do any error checking. What if n is greater than the array length ?

We do have one annoying problem now ... we have to supply an additional parameter.
To avoid this problem, we will make another method that will be called by the user in place of this one. This new method will provide the parameter for us:

```
    public static int sum(int[] theArray) {
        return (sum(theArray, theArray.length - 1);
    }
```

Can we have these two methods with the same name ? Yes ... since they have a different signature (i.e., parameter list). Recall that this is known as ***overloading***.

Now to test it we merely call this new method. In fact, we should make the original method **private**. This technique of making a kind of "wrapper" for the recursive method is known as ***indirect recursion***. The first method is ***directly recursive*** but this second one is called ***indirectly recursive*** since it does not call itself, but does call a recursive method. This second method is commonly called a ***helper*** method.

It should be a simple task for you to take the ArrayList method that we wrote earlier and make it non-destructive. Make sure you can do this. Here is the code altogether:

```
//Example of summing elements of Arrays and Arr using recursive and overloaded
sum methods.
import java.io.*;
import java.util.*;
public class SumTest {
    //This destructive method returns the sum of a vector of Integers
    public static int sum(ArrayList<Integer> arrayList) {
        //BASE CASE
        if (arrayList.isEmpty()) return 0;

        //RECURSIVE STEP
        Integer element = arrayList.get(0);
        arrayList.remove(element);
        return (element.intValue() + sum(arrayList));
    }

    //This is a helper method
    private static int sum(int[] theArray, int n) {
        if (n == 0) return theArray[0];
```

```
        return (theArray[n] + sum(theArray,n-1));
    }

    //This method returns the sum of an array of Integers.
    public static int sum(int[] anArray) {
        return (sum(anArray, anArray.length-1));
    }

    public static void main(String[] args) throws IOException {
        int MAX = 100, i=0, count=0, temp=0;
        int[] a = new int[MAX];
        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        System.out.println("Enter the first integer (0 to end, Max=100):");

        //Add first number to array and vector
        temp = new Scanner(System.in).nextInt();
        a[i] = temp;
        arrayList.add(new Integer(temp));

        //read numbers into array and vector
        while ((a[i] != 0) && (i < MAX)) {
            i++;
            count++;
            System.out.println("Enter another integer (0 to end):");
            temp = new Scanner(System.in).nextInt();
            a[i] = temp;
            arrayList.add(new Integer(temp));
        }
        System.out.println("Here are the results:");
        System.out.println("Array Sum = " + sum(a));
        System.out.println("ArrayList Sum = " + sum(arrayList));
    }
}
```

Note that this method adds the final zero to the arrayList and the array. Can **you** fix this problem ? I hope so. Does it matter ?

Example (Reversing a StringBuffer):

Consider our earlier example in which we reversed the characters of a **String**. We could replace the **String** with a **StringBuffer** and then change our method so that the actual characters of the original **StringBuffer** would be reversed. This would be considered a destructive method since it modifies the original object parameters. Take notice of the method overloading and the use of a **private** helper method.

Our strategy will be to swap the first and last characters of the **StringBuffer** and then move inwards on the **StringBuffer** the same way we iterated through the **String** in our palindrome example. To do this, we will not shrink the **StringBuffer** each time. Instead, we will keep track of where we are by using indices as we traverse recursively. We will write the following method which will reverse the characters of the given **StringBuffer** within the indices specified by i and j:

```
private static void reverseBetween(StringBuffer s, int leftIndex, int
rightIndex) { ... }
```

We will call this method recursively, increasing **leftIndex** each time while decreasing **rightIndex**. Thus the base case will be the stopping condition when **leftIndex** is greater than or equal to the **rightIndex**. Here is the code:

```
    private static void reverseBetween(StringBuffer s, int leftIndex, int
rightIndex) {
        //BASE CASE
        if (leftIndex >= rightIndex )return;

        //RECURSIVE STEP
        char temp = s.charAt(leftIndex);
        s.setCharAt(leftIndex, s.charAt(rightIndex));
        s.setCharAt(rightIndex, temp);
        reverseBetween(s, leftIndex+1, rightIndex-1);
    }
```

Notice that the method does not return anything. It is simply a procedure that modifies the argument passed to it (i.e., modifies the **StringBuffer**).

Of course, this is not the method that we wanted. We want the following method signature:

```
     public static void reverse(StringBuffer s) { ... }
```

But we can make use of the **reverseBetween()** method, just ensuring that we pass in "good" **leftIndex** and **rightIndex** parameters:

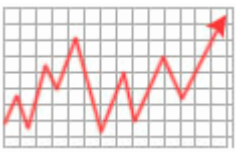
```
    //Reverse the contents of StringBuffer s
    public static void reverse(StringBuffer s) {
        if (s.length() > 1)
           reverseBetween(s, 0, s.length()-1);
    }
```

Notice that this method is "indirectly" recursive since it calls a recursive method, but it does not call itself. Here is the test code:

```
public class DReverseTest {

    //Reverse the contents of StringBuffer s
    public static void reverse(StringBuffer s) { ... }

    private static void reverseBetween(StringBuffer s, int leftIndex, int
rightIndex) { ... }

    public static void main(String[] args) {
        //First check to see that there is at least one command line argument
        if (args.length == 0) {
            System.out.println("Usage: DReverseTest <aString>");
            System.exit(-1);
        }
        StringBuffer input = new StringBuffer(args[0]);
        System.out.println("string buffer's initial contents: " + input);

        reverse(input);  //Notice that we call the public method

        System.out.println("string buffer's final contents:   " + input);
    }
}
```

5.6 Some More Examples

Example (Averaging):



Consider finding the average of a set of Integers that are stored in an **ArrayList**. One approach is to use the **sum** method just mentioned and merely

divide by the number of elements:

```
public static double avg(ArrayList<Integer> v) {
    return (sum(v) / (double)v.size());
}
```

This approach is fine and will work as desired. However, a more interesting challenge is to write a recursive averaging method, without using the **sum()** method. Let us try to see if we can express the averaging function recursively. We will say that avg(A, n) is the function for finding the average of **n** numbers stored in an ArrayList **A**. We will denote the elements of the ArrayList A to be A_1, A_2, A_3, ..., A_n.

What about the base cases ?

$avg(A, 0) = 0$
$avg(A, 1) = A_1$

Now the inductive case:

$$
\begin{aligned}
avg(A, n) &= (A_1 + A_2 + A_3 + \ldots + A_n) / n \\
&= [(A_1 + A_2 + A_3 + \ldots + A_{n-1}) / n] + [A_n / n] \\
&= [((n-1) / n) * (A_1 + A_2 + A_3 + \ldots + A_{n-1}) /(n-1)] + [A_n / n] \\
&= [((n-1) / n) * avg(A, n-1)] + [A_n / n] \\
&= [((n-1) * avg(A, n-1) + A_n] / n
\end{aligned}
$$

So the recursive definition is:

$avg(A, 0) = 0$
$avg(A, 1) = A_1$
$avg(A, n) = [((n-1) * avg(A, n-1) + A_n] / n$

Now we can easily write the code:

```
import java.io.*;
import java.util.*;
public class AvgTest {
    //This method returns the average of a n Integers in the given vector.
    private static double avg(ArrayList<Integer> a, int n) {
        //BASE CASES:
        if (n > a.size()) return -1;
        if (n == 0) return 0;

        Integer last = a.get(a.size() - 1);
        if (n == 1) return last;

        //RECURSIVE STEP
        a.remove(last);
        return (((n-1) * avg(a, n-1) + last)/n);
    }

    //This method returns the average of an ArrayList of Integers
    public static double avg(ArrayList<Integer> a) {
        return (avg(a, a.size()));
    }

    public static void main(String[] args) {
        int input = 0;
        ArrayList<Integer> nums = new ArrayList<Integer>();
        do {
            System.out.println("Enter integers one at a time (0 to end):");
            if ((input = new Scanner(System.in).nextInt()) != 0)
                nums.add(input);
```

```
        } while (input != 0);
        System.out.println("The average is " + avg(nums));
    }
}
```

Do we really need this extra index ? Can we re-write this method using direct recursion ? Well the index of **n** represents the size of the array, which changes as the recursion progresses. So we have to be careful:

```
private static double avg(ArrayList<Integer> a) {
    if (a.isEmpty()) return 0;

    Integer last = a.get(a.size() - 1);
    if (a.size() == 1)
        return last;

    a.remove(last);
    int size = a.size();
    return (size * avg(a) + last)/(size+1);
}
```

Notice the use of the **size** variable to that when the **ArrayList** size changes due to the recursion, this local variable stays constant. We can re-arrange the order of the calculations in the expression if we want to eliminate this extra **size** variable. We just have to make sure that the **size** is used BEFORE the recursive call:

```
private static double avg(ArrayList<Integer> a) {
    if (vec.isEmpty()) return 0;

    Integer last = a.get(a.size() - 1);
    if (a.size() == 1)
        return last;

    vec.remove(last);
    return (1/(a.size()+1.0))*(a.size() * avg(a) + last);
}
```

Of course, we can play games like this all day long :). We can actually eliminate the 2nd base case, since it is handled by the recursive call:

```
private static double avg(ArrayList<Integer> a) {
    if (a.isEmpty()) return 0;

    Integer last = a.get(a.size() - 1);
    a.remove(last);
    return (1/(a.size()+1.0))*(a.size() * avg(a) + last);
}
```

Also, we can get rid of the **last** variable by re-arranging the expression again:

```
private static double avg(ArrayList<Integer> a) {
    if (a.isEmpty()) return 0;
    return (1.0/(a.size()))*(a.remove(0) + a.size() * avg(a));
}
```

But WHY would you write such a complicated looking expression ??? It is better to leave the **last** variable as it was for ease of reading the code.

Example (Selecting):

Now, what if we wanted to not just count the odd integers in an **ArrayList**, but to gather and return them (i.e., select them) ? That is, make a new **ArrayList** that will contain all of the odd integers from the original ArrayList. Can we do this with direct recursion ? Perhaps we should make a helper method.

We can take the approach that is common to everyday life. Get a blank list ready, and write down all the odd integers in it. In terms of coding, we can prepare the blank "list" as an initially empty ArrayList passed in as a parameter. We can then add to this list as we go through the recursion. Hence we will need to pass this list along when we do the recursive calls.

```
import java.io.*;
import java.util.*;

public class SelectTest {

    //This destructive method returns the odd Integers in an ArrayList
    public static ArrayList<Integer> selectOdd(ArrayList<Integer> a) {
       return (selectOdd(a, new ArrayList<Integer>()));
    }

    //This is the helper method that does all of the work
    public static ArrayList<Integer> selectOdd(ArrayList<Integer> a, ArrayList<Integer> result)
{
        //BASE CASE:
        if (a.isEmpty()) return result;

        //RECURSIVE STEP
        Integer element = a.get(0);
        a.remove(element);

        if (element%2 != 0)
            result.add(element);

        return selectOdd(a, result);
    }

    public static void main(String[] args) {

        int input = 0;
        ArrayList<Integer> nums = new ArrayList<Integer>();
        do {
            System.out.println("Enter integers one at a time (0 to end):");
            if ((input = new Scanner(System.in).nextInt()) != 0)
                nums.add(input);
        } while (input != 0);
        ArrayList<Integer> result = selectOdd(nums);
        System.out.println("Here are the odd integers:");
        for (int i=0; i<result.size(); i++) {
            System.out.println(result.get(i));
        }
    }
}
```

Can you write this method using direct recursion (i.e., no additional parameters) ?

```
    public static ArrayList<Integer> selectOdd(ArrayList<Integer> a) {
        if (a.isEmpty())
     return new ArrayList<Integer>();

        Integer element = a.get(0);
```

```
        a.remove(element);

        result = selectOdd(a);
        if (element%2 != 0)
            result.add(element);

        return result;
    }
```

Do the odd numbers come back in the same order as before ? Think about it.

What about making it non-destructive now:

```
    public static ArrayList<Integer> selectOdd(ArrayList<Integer> a) {
        if (a.isEmpty()) return new ArrayList<Integer>();

        Integer element = a.get(0); //remove from end now

        a.remove(element);
        result = selectOdd(a);
        a.add(element); //restore after recursive call by adding to end

        if (element%2 != 0)
            result.add(element);

        return result;
    }
```

Example (Choosing k items from N):

Denote the number of ways of choosing **k** items out of **N** by **C(N,k)**. We can determine what this number is by doing an experiment and counting.
The approach we'll take is to look at any one particular item, say **X**, and decide if it is either going to be chosen in the answer or if it is not. All possibilities have to be considered.

- If **X** is chosen, there are **N-1** items left from which we must still choose **k-1** (one has already been chosen).
- If **X** is not chosen, there are **N-1** items left from which we still must choose **k** (since we haven't chosen any yet).

  That is, **C(N,k) = C(N-1,k-1) + C(N-1,k)**

We can define the degenerate cases (the basis) as

1. **C(N,N) = 1**
2. **C(N,0) = 1**
3. **C(N,k) = 0** if **k > N** (We assume **k <= N** always)

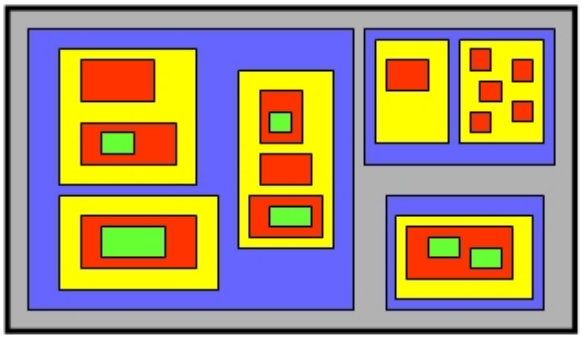
Here is the code:

```
//Computing C(n,k) example.  (use: java ChooseTest n k)
public class ChooseTest {
    //returns C(n,k)
    public static int choose(int n, int k) {
        //BASE CASES:
        if (n == k) return 1; //C(n,n) = 1
        if (k == 0) return 1; //C(n,0) = 1
```

```
        if (k > n) return 0;  //C(n,k) = 0

        //RECURSIVE CASES: C(N,k) = C(N-1,k-1) + C(N-1,k)
        return (choose(n-1,k-1) + choose(n-1,k));
    }

   public static void main(String[] args) {
        //First check to see that there is at least one command line argument
        if (args.length < 2) {
            System.out.println("Usage: ChooseTest <n> <k>");
            System.exit(-1);
        }
        int n = Integer.parseInt(args[0]);
        int k = Integer.parseInt(args[1]);

        System.out.println("C(" + n + "," + k + ")= " + choose(n,k));
    }
}
```

Example (Counting boxes within boxes):

Consider the following scenario. You wrap up your friends graduation gift in a box...but to be funny, you decide to wrap that box in a box and that one in yet another box. Also, to fool him/her you throw additional wrapped boxes inside the main box. The boxes-within-boxes scenario is recursive. Consider this problem now. We have boxes that are completely contained within other boxes and we would like to count how many boxes are completely contained within any given box. Here is an example where the outer box has 29 internal boxes:



Assume that there are some classes that implement the following **Box** interface:

```
public interface Box {
    public ArrayList<Box>   internalBoxes();      //Returns the boxes within the
receiver box
    public boolean          isEmpty();            //Return whether or not there
are any boxes within the receiver box
    public void             addBox(Box aBox);     //Add the given box to the
receiver
    public void             removeBox(Box aBox);  //Remove the given box from
the receiver
}
```

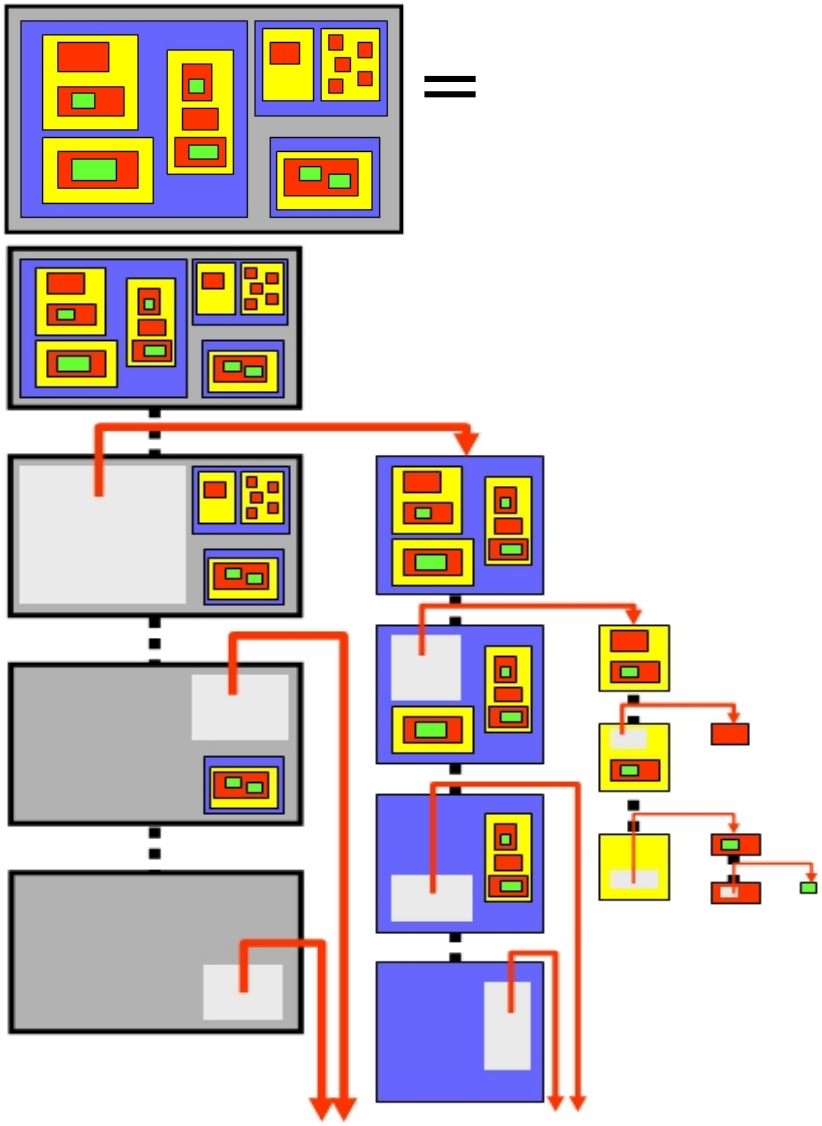
Now we have no idea how the boxes are stored or maintained. All we know is how to use the interface.
How can we write a recursive method to find the total number of boxes with a given box ?

- Base case: if the given box is empty, the answer is 0.
- Inductive case: the number of boxes directly within the receiver + the total of the number of internal boxes within each box that is within the receiver.

Here is the code:

```
public int countBoxes(Box aBox) {
    if (aBox.isEmpty())
        return 0;

    int count = 0;
    ArrayList<Box>   innerBoxes = aBox.internalBoxes();

    // Go through each internal box and get count
    // their internal boxes recursively
    for (Box b: innerBoxes) {
        count += countBoxes(b);
    }
    // Return the count of all internal boxes' boxes plus the number
    // of internal boxes at this level of the recursion
    return (count + innerBoxes.size());
}
```

Notice that we need the **`for`** loop here to go through all internal boxes. What if we were allowed to destroy the boxes ? Can we do this without a for loop ?



Here is the code:

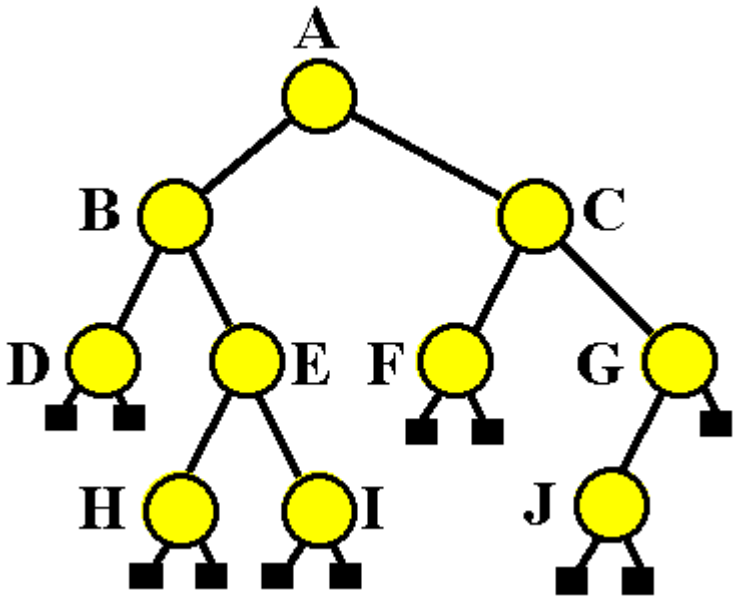
```
public int countBoxes(Box aBox) {
    if (aBox.isEmpty())
        return 0;

    Box   anInnerBox = aBox.internalBoxes().get(0);
    aBox.removeBox(anInnerBox);
    return (countBoxes(aBox) + countBoxes(anInnerBox) + 1);
}
```

Will this work ? Think about it. Are the recursive sub problems always smaller ? Which of the two pieces of code do you prefer ?
Try writing a class that implements the box interface and test this method out.

Example (Height of a Tree):

Consider a class called **BinaryTree** that keeps a collection of ***nodes***. The topmost node is the ***root*** of the tree. Each **node** stores an item of some kind and keeps pointers to a left and right ***child***. The children are themselves trees and are considered ***subtrees*** of the original tree. If a node has no left or right child, then **null** is stored there. A node with no children at all is considered a ***leaf***.

Here is an example of a binary tree:



A basic implementation of a binary tree may look like this:

```
public class TreeNode {
    private TreeNode  rightChild, leftChild;
    private Object    item;   // set to whatever the node represents

    public TreeNode(Object anObject) {
        this(anObject, null, null);
    }

    public TreeNode(Object anObject, TreeNode aLeftChild, TreeNode aRightChild)
{
        item = anObject;
```

```
        rightChild = aRightChild;
        leftChild = aLeftChild;
    }

    public final TreeNode rightChild() {
        return rightChild;
    }

    public final TreeNode leftChild() {
        return leftChild;
    }

    public final void setRightChild(TreeNode child) {
        rightChild = child;
    }

    public final void setLeftChild(TreeNode child) {
        leftChild = child;
    }
}
```

Notice that we called the class **TreeNode**. In fact, every node in the tree is a tree itself. It is actually a recursive data structure.

Here is an example of how to build the tree above:

```
public static void main(String args[]) {
    TreeNode root;
    root = new TreeNode("A",
                new TreeNode("B",
                            new TreeNode("D"),
                            new TreeNode("E",
                                        new TreeNode("H"),
                                        new TreeNode("I"))),
                new TreeNode("C",
                            new TreeNode("F"),
                            new TreeNode("G",
                                        new TreeNode("J"),
                                        null)));
}
```

The ***height*** of a tree is the number of nodes encountered along the longest path from (but not including) the root to a leaf of the tree.
The tree above has a height of 3. Can you write a recursive method that determines the height of a binary tree ?

```
    public int height() {
        if (leftChild == null)
            if (rightChild == null)
               return 0;
            else
               return (1 + rightChild.height());
        else
            if (rightChild == null)
               return (1 + leftChild.height());
            else
               return (1 + Math.max(leftChild.height(), rightChild.height()));
    }
```

5.7 Efficiency with Recursion

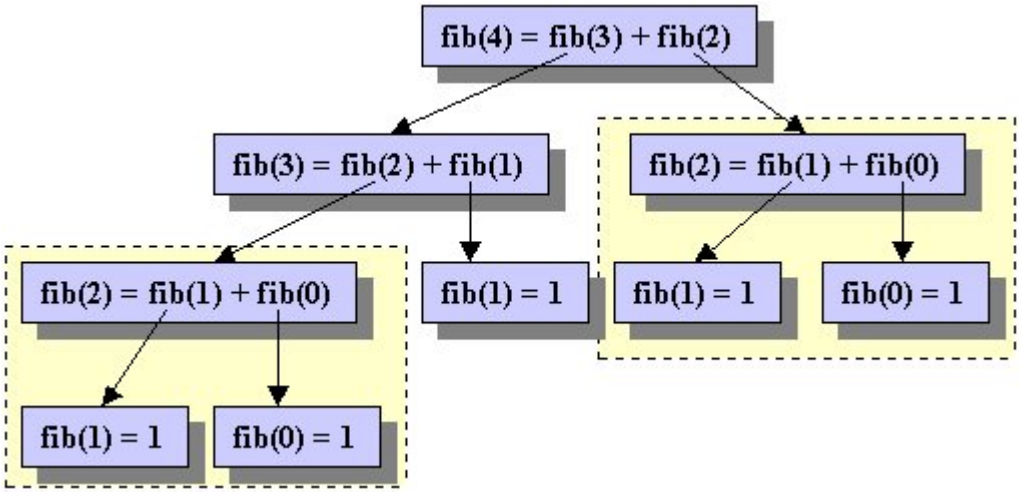
Although recursion is a powerful problem solving tool, it has some drawbacks. A non-recursive (or

iterative) method may be **more** efficient than a recursive one for two reasons:

- there is an overhead associated with large number of method invocations
- some algorithms are inherently inefficient.

Example: Computing the **n**th Fibonacci number can be written as:

1	if **n** = **0**
fib(n) = 1	if **n** = **1**
fib(n-1) + fib(n-2)	if **n** > **1**



Notice in the above computation some problems (e.g. **fibonacci(2)**) are being solved more than once, even though we presumably know the answer after doing it the first time. The following iterative solution avoids this. (It can also be avoided with a properly formulated recursive solution)

```
public int finonacci(int n) {
    if (n < 0)
       return -1;

    int first = 1;
    int second = 1;
    int third = 1;
    for (i=2; i<=n; i++) {
        third = first + second; //compute new value
        first = second; second = third; //shift the others to the right
    }
    return third;
}
```

Try designing an efficient recursive Fibonacci method. You may use indirect recursion.

Here are some points to remember about efficiency:

- Anything that can be done with a loop using variables can be done recursively with the variables replaced by parameters.
- In some languages like **Lisp** and **Smalltalk**, recursion is normal. Consequently, recursive

messages are implemented very efficiently.
- In Lisp, **for** loops and **while** loops are implemented recursively. Hence loops are no more (also no less) expensive than recursive invocations.

5.8 Practice Questions

Try writing recursive methods for the following problems:

Raising To A Power

Consider the evaluation of $\mathbf{x^n}$,where **n** is a non-negative integer.

- 1st Approach: (Could be done with linear recursion)
  $\mathbf{x^n = x * x * x*.....* x}$
- 2nd Approach (Recursive Formulation)

$\mathbf{x^{n/2} * x^{n/2}}$	if **n** is even
$\mathbf{x^n = x * x^{n-1}}$	if **n** is odd
$\mathbf{x^0 = 1}$	if **n** is **0** (Basis Case)

Which approach would require fewer multiplications ?

Binary Search:

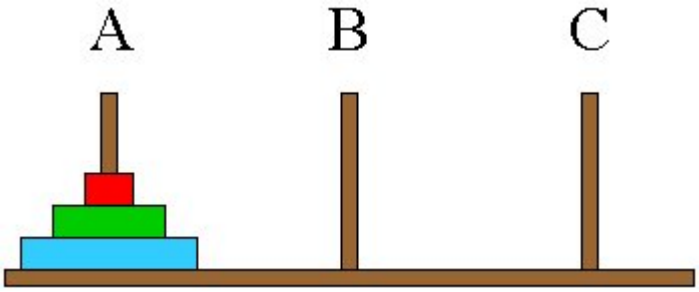
Write a recursive method which would locate an element in a **sorted** collection (i.e. Array or Vector) of **n** elements without having to check all the elements
(as in the worst case). That is, we'd like the maximum number of elements that need to be searched to be **log(n)**. The problem can be phrased recursively as follows.

- **Basis Case:** looking for an element in a collection of size 1 is easy - just check the single element.
- **Inductive Case:** Look at the middle element in the collection, if this bigger than the one you are looking for then search the portion of the collection before it. If it's smaller search the portion of the collection after it.

Notice how the problem of looking for an element in a collection of size **n**, is phrased in terms of looking for an element in a collection of size **n/2**. That's how we can be sure that only a logarithmic number of elements need to be examined.

A good exercise would be to build a **SortedCollection** class from scratch and use a binary search to implement its **contains(Object)** method (such as is implemented for class **Vector**).

The Tower of Hanoi Problem



The task is to move the disks from peg A to peg B using peg C as an intermediary (if necessary). The rules are as follows:

1. When a disk is moved, it must be placed on one of the three pegs.
2. Only one disk may be moved at a time, and it must be the top disk on one of the pegs.
3. A larger disk may never be placed on top of a smaller one.

A solution is best understood by considering the problem starting from simple cases to more complex cases.

Move (1 disk from A to B using C) => JUST DO IT

Move (2 disks from A to B using C) =>
    Move 1 A disk to C
    Move 1 A disk to B
    Move 1 C disk to B

Move (3 disks from A to B using C) =>
    Move 2 A disks to C using B (use previous step)
    Move 1 A disk to B
    Move 2 C disk to B using A
...

Move (n disks from A to B using C) =>
    Move n-1 A disks to C using B
    Move 1 A disk to B
    Move n-1 C disk to B using A

Implement a **TowersOfHanoi** application which will allow the user to specify some number of disks on one peg and then select which other peg they should be moved to. The application must them go about moving the disks over following the rules above. This is a good application to animate on a graphical user interface.

---

6 Menus and Dialogs

What's in This Set of Notes?

In most GUI designs, there are often many options and user preferences. In many cases it is too difficult to place everything onto one window. For this reason, menus and dialog boxes are used. Menus allow us to efficiently group/hide similar options together so that the screen does not become cluttered. Dialog Boxes allow us to momentarily take the user's attention away from the main window so as to obtain information or ask questions which are required at specific points in the application. We will look into both of these "tools" here.

Here are the individual topics found in this set of notes (click on one to go there):

6.1 Using Menus

What is Menu?

- Conceptually, a menu is a list of buttons each of which have their own corresponding action when selected.
- Types of menus include:
  - ***drop-down*** (or ***pull-down***) - usually associated with an application's menubar
  - ***popup*** - associated with any container component (i.e., often accessed via right button click)
  - ***cascaded*** - pops up when another menu item is selected (i.e. a sub menu)

What are the main Menu Classes?

- **JMenuBar**:
  - An object at the top of the frame that contains "pull down" menus. Tthere can be only one per frame.
  - Only the names of the pull down menus are displayed on the menubar.
  - When one of the menu names is selected, the corresponding menu appears (i.e., pops up).
- **JMenu**:
  - A menu that may contain:
    - menu items (called **JMenuItems**), which the user can select from (like buttons).
    - separator lines (called **JSeparators**) to divide up the items into logical groupings.
    - other menus (i.e., JMenus) which act as cascading menus.
  - The user selects an option and an action is performed (just like clicking a button).
  - It is added to a **JMenuBar** object by specifying its name (or label) which will appear on the menubar.

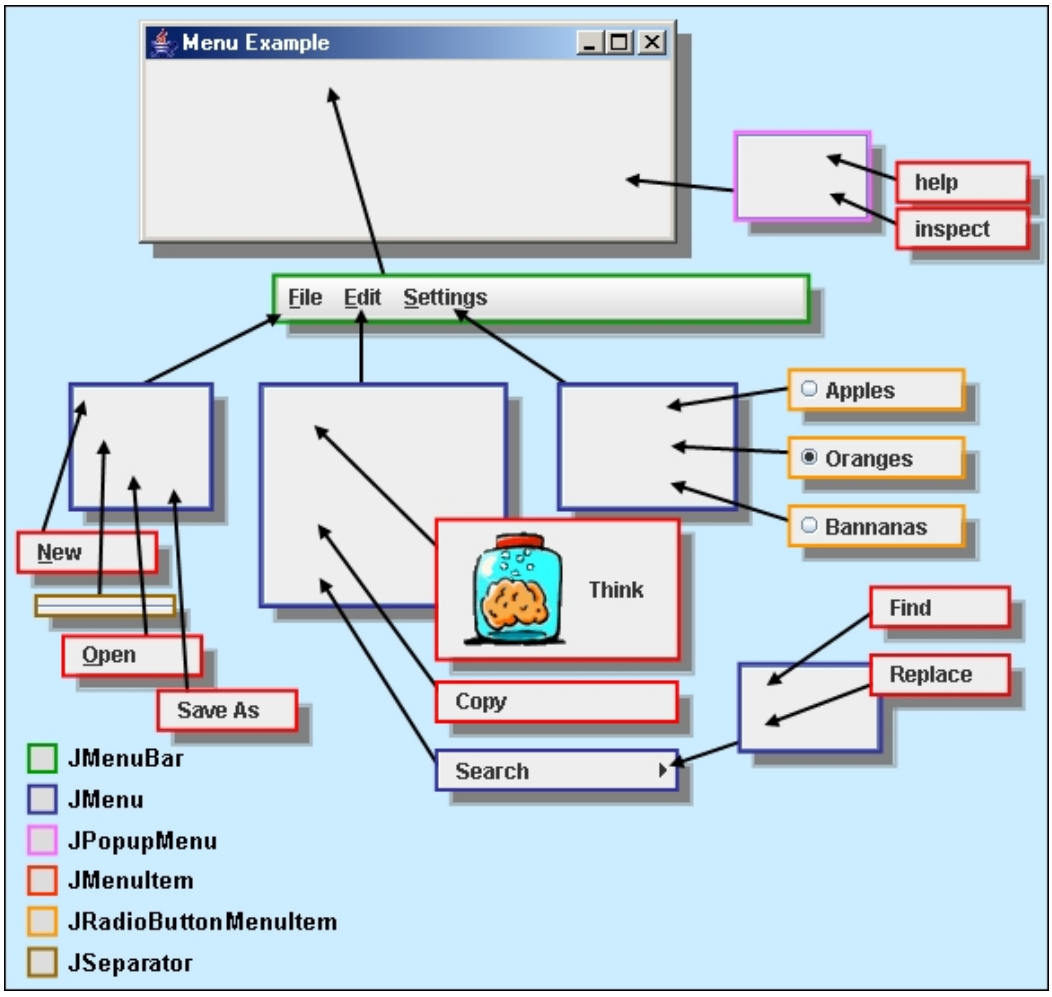
- **JMenuItem**:
  - A button on a menu. When the user selects the "button", the action associated with the menu item is performed.
  - **ImageIcons** can be used on menus as well by assigning an icon to a menuItem (just like **JButtons**).
  - There are also **JRadioButtonMenuItem**:
    - A menu item that is part of a group of menu items in which only one item in the group can be selected.

- **JPopupMenu**:
  - A small window which pops up and displays a menu of choices.
  - Used for the menu that appears when the user selects an item on the menu bar.
  - Also used for "pull-right" menus that appear when the user selects a menu item that activates it.
  - A **JPopupMenu** can also be used anywhere else you want a menu to appear (e.g., when the user right-clicks in a specified area).

How Does it all "Hook" Together ?

This diagram shows how all these components hook together. Basically ....

- A single **JMenuBar** is added to a **JFrame**
- Multiple **JMenu**s are added to the **JMenuBar**
- **JMenuItem**s and cascaded **JMenu**s are added to other menus
- **JPopupMenu**s are added to **JFrame**s



How do we write the code to get it all hooked up ?

1. We need to create and add a **JMenuBar** to our **JFrame** by doing the following in our **JFrame** constructor:

   - Create an new instance of **JMenuBar**:

```
JMenuBar myMenuBar = new JMenuBar();
```

- Set the **JFrame**'s menubar to that instance:

```
myFrame.setJMenuBar(myMenuBar);
```

2. We need to add **JMenus** to our menu bar by doing the following in our **JFrame** constructor:
   - Create a new instance of **JMenu** and give it a label:

```
JMenu  fileMenu = new JMenu("File");
```

   - Add the **JMenu** to the **JMenuBar**:

```
myMenuBar.add(fileMenu);
```

   - Optionally set the keyboard accelerators (i.e., quick keys):

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

3. We can add **JMenuItems** and/or **JSeparators** to our **JMenus**, which we can also do in our **JFrame** constructor:
   - Create an new instance of **JMenuItem** and/or **JSeparator** and give it a label:

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

   - Add these items to the **JMenu**:

```
fileMenu.add(newItem);
fileMenu.add(sepItem);
```

   - Set the keyboard accelerators for the **JMenuItems** if desired:

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

   - 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
  }
});
```

4. We can add **JRadioButtonMenuItems** to our **JMenus**, which we can also do in our **JFrame** constructor:
   - Create new instances of **JRadioButtonMenuItem** and give them labels:

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

   - Add the **JRadioButtonMenuItems** to the **JMenu**:

```
settingsMenu.add(rbItem1);
settingsMenu.add(rbItem2);
settingsMenu.add(rbItem3);
```

   - Add the **JRadioButtonMenuItems** to a **ButtonGroup()**:

```
ButtonGroup fruits = new ButtonGroup();
fruits.add(rbItem1);
fruits.add(rbItem2);
fruits.add(rbItem3);
```

- Add an **ActionListener** to each **JRadioButtonMenuItem:**

```
// they may all go to the same event handler (as here), or to separate ones
rbItem1.addActionListener(this);
rbItem2.addActionListener(this);
rbItem3.addActionListener(this);

public void actionPerformed(ActionEvent e) {
   // Handle the selection of these items from the menu
}
```

5. We can add cascading menus simply by adding a **JMenu** to another **JMenu**:
   - Create an new instance of **JMenu** and give it a label

```
JMenu  searchMenu = new JMenu("Search");
```

   - Add **JMenuItems** to the new **JMenu** and set the keyboard accelerators if desired:

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

   - Add the **JMenu** to some other **JMenu**:

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

6. We can add a **JPopupMenu** to the **JFrame**:
   - Create a new instance of **JPopupMenu** and give it a label:

```
JPopupMenu popupMenu = new JPopupMenu();
```

   - Add **JMenu** and **JMenuItems** to the new **JPopupMenu** and set the keyboard accelerators if desired:

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

   - Add a MouseListener to the JFrame and then handle a mouseReleased() event in which the **show()** message is sent to the menu:

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

We can keep in mind that there are other settings for our **JMenus** and **MenuItems**:

- To set the **Color**:

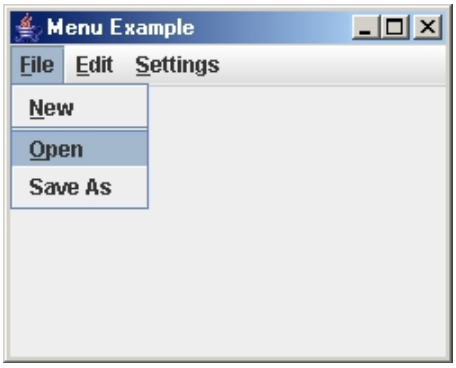
```
anItem.setBackground(Color.red);
anItem.setForeground(Color.yellow);
```

- To Enable/Disable various items:

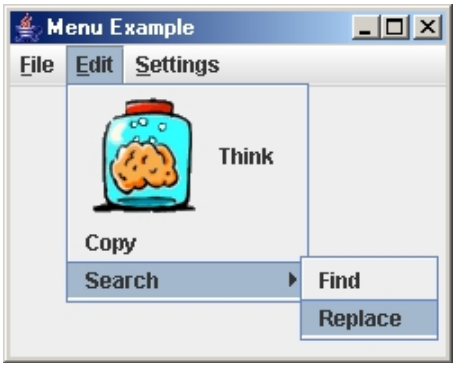
```
anItem.setEnabled(true);
anItem.setEnabled(false);
```

Example:

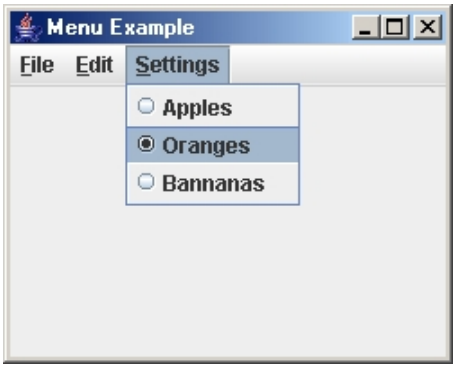
Here is an example in which we investigate the use of a menubar with menus as well as cascaded menus and a popup menu. The example has no purpose other than to show you how the different menus are created and used. Here are screen snapshots that show the menus that we will create in this example:



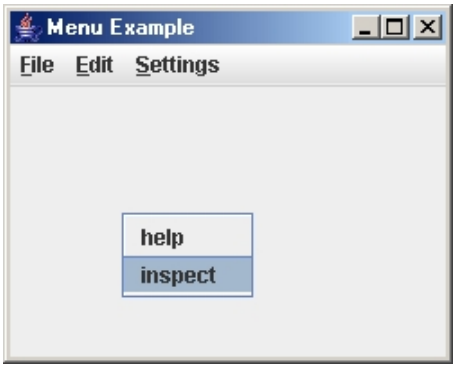
A **standard** menu



A **cascaded** menu



A menu with **radio** buttons



A **pop-up** menu

The example shows how the use of many menu items can lead to a lot of instance variables. In addition, we use a common event handler for all menu items and apply a dispatching strategy which calls the appropriate react method for the given menu item. We could have omitted the react methods and merely placed all this code within the single event handler but this could look messy if the react methods are large.

How would the example look if we used anonymous classes instead of one event handler ? It may not save much in code size but we only need to update the class in one place instead of two when a menu item is added or removed! Also, we might not need to keep all the menu items in instance variables! Why not give it a try and see what it looks like.

```
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,
                 bannanaItem, helpItem, inspectItem;

     JPopupMenu popupMenu;

     public MenuExample(String title) {

          super(title);

          // Create the menu bar
          JMenuBar menuBar = new JMenuBar();
          setJMenuBar(menuBar);

          // 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(bannanaItem = new
        JRadioButtonMenuItem("Bannanas"));
        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(bannanaItem);

        // 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 and dispatch to the appripriate helper
    method accordingly
    public void actionPerformed(ActionEvent e){

        if (e.getSource() == newItem)
            reactToNewMenuSelection();
        else if (e.getSource() == openItem)
            reactToOpenMenuSelection();
```

```
            else if (e.getSource() == saveAsItem)
                reactToSaveAsMenuSelection();
            else if (e.getSource() == copyItem)
                reactToCopyMenuSelection();
            else if (e.getSource() == thinkItem)
                reactToThinkMenuSelection();
            else if (e.getSource() == findItem)
                reactToFindMenuSelection();
            else if (e.getSource() == replaceItem)
                reactToReplaceMenuSelection();
            else if (e.getSource() == helpItem)
                reactToHelpMenuSelection();
            else if (e.getSource() == inspectItem)
                reactToInspectMenuSelection();

        }
        // Here are all the helper methods for handling the menu choices
        public void reactToNewMenuSelection() {
            System.out.println("reacting to NEW selection from menu");
        }
        public void reactToOpenMenuSelection() {
            System.out.println("reacting to OPEN selection from menu");
        }
        public void reactToSaveAsMenuSelection() {
            System.out.println("reacting to SAVE AS selection from menu");
        }
        public void reactToThinkMenuSelection() {
            System.out.println("reacting to THINK selection from menu");
        }
        public void reactToCopyMenuSelection() {
            System.out.println("reacting to COPY selection from menu");
        }
        public void reactToFindMenuSelection() {
            System.out.println("reacting to FIND selection from menu");
        }
        public void reactToReplaceMenuSelection() {
            System.out.println("reacting to REPLACE selection from menu");
        }
        public void reactToHelpMenuSelection() {
            System.out.println("reacting to HELP selection from popup menu");
        }
        public void reactToInspectMenuSelection() {
            System.out.println("reacting to INSPECT selection from popup
        menu");
        }
        public static void main(String args[]) {
            new MenuExample("Menu Example").setVisible(true);
        }

    }
```

6.2 Standard Dialog Boxes

A ***dialog box*** is:

- a separate window that pops up in response to an event occurring in a window.
- often used to obtain information from the user (e.g., entering some values such as when filling out a form).

There are various types of commonly used dialog boxes:

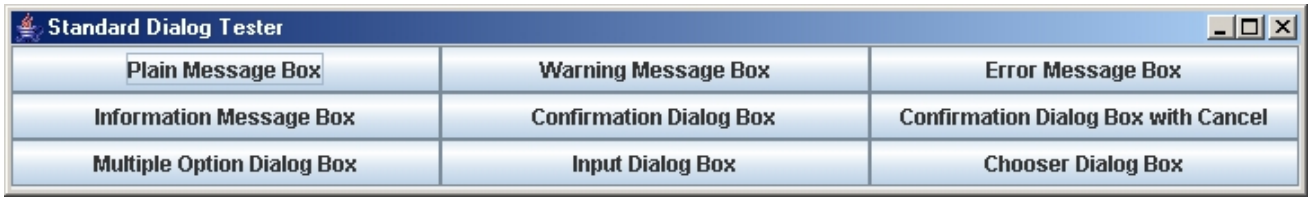
1. Message Dialog - displays a message indicating information, errors, warnings etc...
2. Confirmation Dialog - asks a question such as yes/no
3. Input Dialog - asks for some kind of input
4. Option Dialog - asks the user to select some option

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

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 StandardDialogTester extends JFrame {
    public StandardDialogTester (String title) {
        super(title);
        setLayout(new GridLayout(3, 3));

        JButton aButton;

        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);
            }});

        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);
            }});

        add(aButton = new JButton("Error Message Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               JOptionPane.showMessageDialog(null,
                    "Your program has stopped working !",
                    "Error",
                    JOptionPane.ERROR_MESSAGE);
            }});

        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);
            }});

        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!");
            }});

        add(aButton = new JButton("Confirmation Dialog Box 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, but don't come crying to me once its gone");
break;
         case 1: System.out.println("Well you should pick a new name then"); break;
         case 2: System.out.println("OK, I'll ask you again later"); break;
                }
            }});

        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]);
                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.");
            }});

        add(aButton = new JButton("Input Dialog Box"));
        aButton.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
                String inputValue = JOptionPane.showInputDialog("Please input your name");
                System.out.println("Your name is " + inputValue);
            }});

        add(aButton = new JButton("Chooser Dialog Box"));
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Object[] options = {"Apple", "Orange", "Strawberry", "Bannana", "Cherry"};
                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.");
            }});

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

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

```
    }
}
```

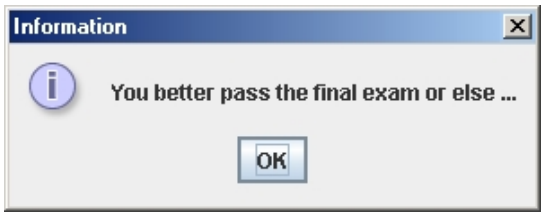
Here are the dialog boxes that will appear.

Plain Message Box:

Information Message Box:

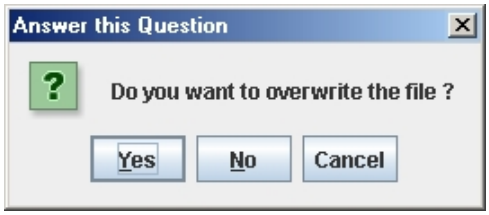
Error Message Box:

Information Message Box:



Confirmation Dialog Box:

Confirmation Dialog Box with Cancel:



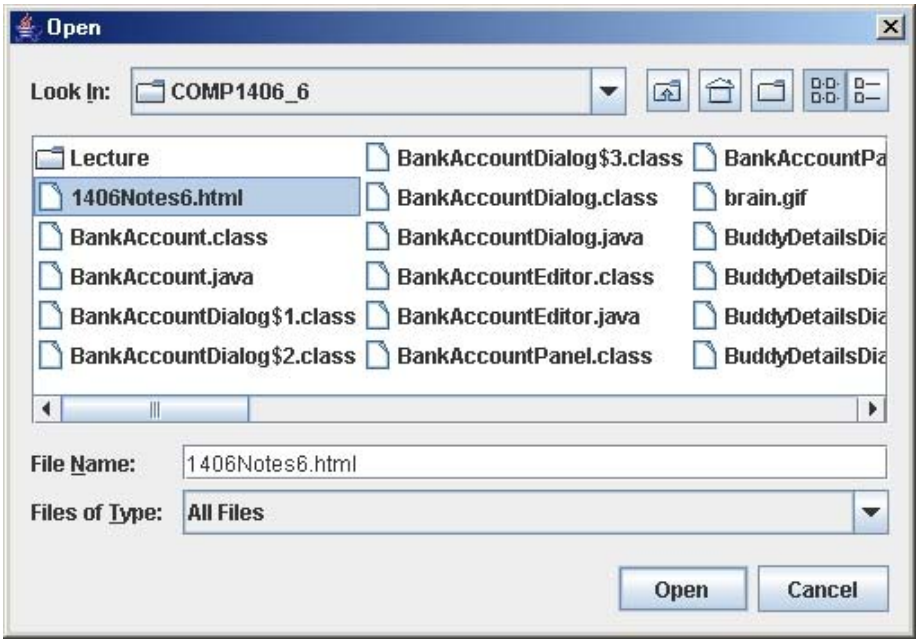
Select Option Dialog Box:

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:

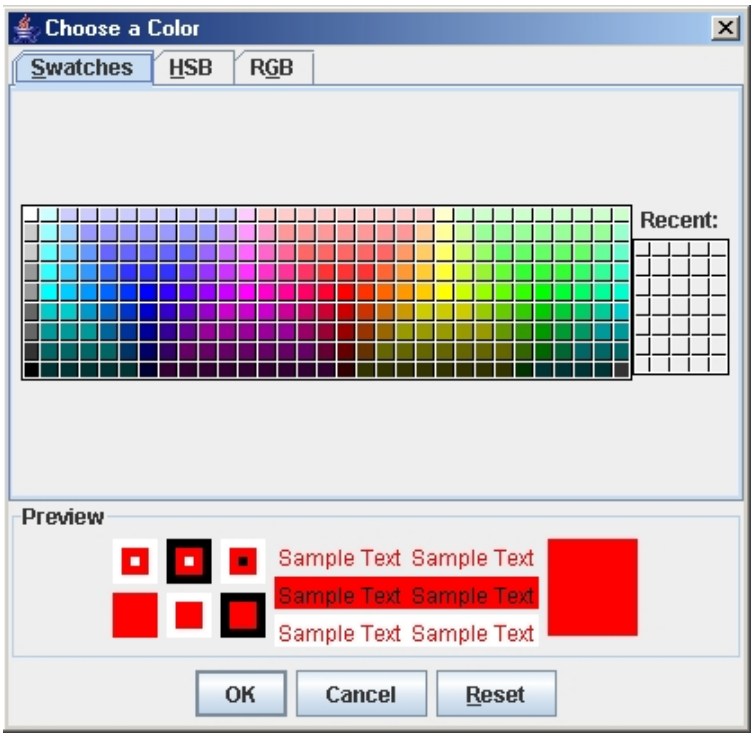


Here is some code that opens up a **JFileChooser** box and displays the filename (no path) that the user selects.

```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(this);

if (returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println("You chose to open this file: " +
                       chooser.getSelectedFile().getName());
}
```

There are more options available that allow you to set the filters and starting directories. Take a look at the Swing API.

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



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

```
Color newColor = JColorChooser.showDialog(
                     this,                     // The parent window
                     "Choose a Color",         // Title on Dialog Box
                     Color.RED);               // Initial color selected
```

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

6.3 Creating Your Own Dialog Boxes

Often, our user interfaces can become cluttered as many components are placed on them. Some of the components' data are not needed unless the user performs some specific action. For example, the user may hit an "Fill out Form" button which would often bring up another window with all the form fields in it. This "new" window is called a ***dialog box***. Dialog boxes can be:

- ***modal:*** no other application window will respond until this one is closed.
    - forces the user to "deal with" the dialog box information before continuing
- ***non-modal:*** can remain open while the user works in other windows

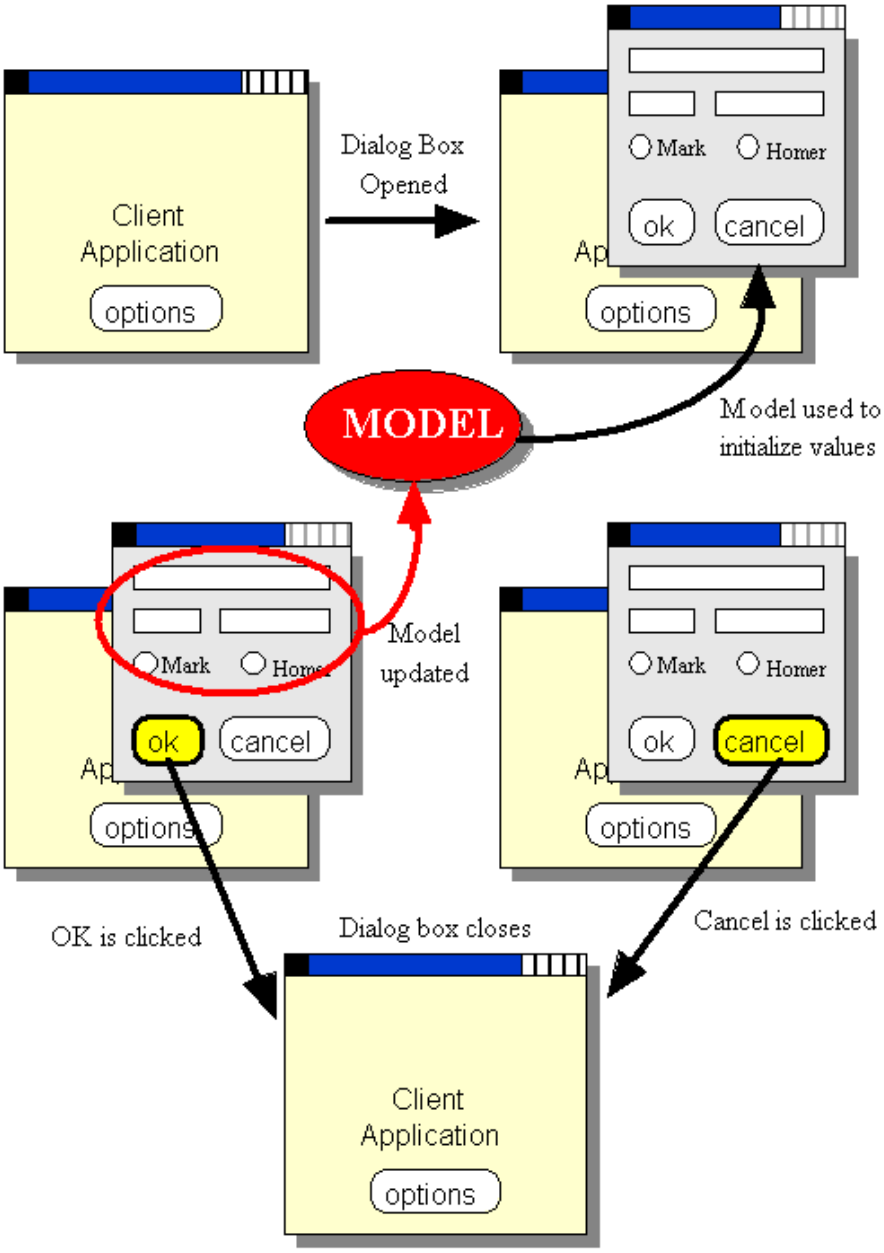
Dialog boxes have an ***owner*** which is the window that caused it to appear. This allows the owner window to close the dialog box when it closes so that all windows belonging to the same application are closed when the application shuts down. Also, ethen the owner window is minimized, the dialog boxes are also minimized.

There are two important terms pertaining to dialog boxes:

- ***Dialog client*** = the application that causes the dialog box to appear
- ***Dialog model*** = the object(s) that the dialog box should affect

Normally, an application communicates to 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.

Here is how everything should work:



Notice that the model is used as the "middle-man" between the two windows:

- 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.

How do we make sure that we can have such interaction with the two windows ?

We make the client implement the following interface which we will define ourselves:

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

So, if the client class implements this interface, it will be sure to have these two methods:

```
public class MyApplication implements DialogClientInterface {
    ...
```

```
    ...
    public void dialogFinished() {
        ...
    }
    public void dialogCancelled() {
        ...
    }
    ...
}
```

Now, since the client application class implements the interface, all other classes know that they can call the **dialogFinished()** method or the **dialogCancelled()** method.

Why would we want to call these methods from outside this class ? Dialog boxes are defined in separate classes, so the client (i.e., usually the main application) has no idea what is going on within those classes (nor should it need to know). The client does, however, need to know the following:

- whether or not the interaction with the dialog box was successful or whether or not it was cancelled
- whether or not the model has been changed

It is easy to see that the dialog box must of course know whether or not it was cancelled (i.e., it has the OK and CANCEL buttons on it). So, we can have the dialog box itself inform the client application whether the dialog box was canceled or not by calling one of these two methods defined in the **DialogClientInterface** that the client implements. That is how the dialog box informs the client of what just happened within it. So, we will need to pass in the client object itself to the dialog box so that the dialog box can send the **dialogFinished()** or the **dialogCancelled()** message to the client. In fact, we already need to pass in the owner to the dialog box ... which in this case will also implement a **DialogClientInterface**, so we do not need any new parameters for our constructor.

```
public class SomeDialog extends JDialog {
    // The client (usually the caller of this dialog box)
    private DialogClientInterface client;

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

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

Here are the steps involved with creating a dialog box:

1. Make your own dialog box class as a subclass of **JDialog**
   - build your window as you would with a normal **JFrame** ... using components/events/listeners
   - make sure to use some kind of **ok/apply** & **cancel/close** button combination. The typical behaviour is that you should not modify any model objects for the application if the **cancel** button is pressed or if window is manually closed.
2. 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,
                   ClassA modelA, ClassB modelB, ...) {
       super(owner, title, modal);
       ...
   }
   ```

   - specifying the **owner** frame ensures that the dialog box is attached to the main application. In our case, the **owner** will also need to be a class that implements the **DialogClientInterface**.
   - the **title** will appear on the dialog box titlebar
   - **modal** indicates whether or not the dialog box is to be modal
   - we may supply numerous **model**-related parameters to represent any information that is to be shared between the main application (i.e., client) and the dialog box. This model data will be

used for both input and output information:

- **input:** use information in the model to set up initial contents of the dialog window's components
- **output:** when the dialog closes the model has the information required from the dialog box interaction

- Notice the call to the superclass constructor (this is a standard **JDialog** constructor being called).

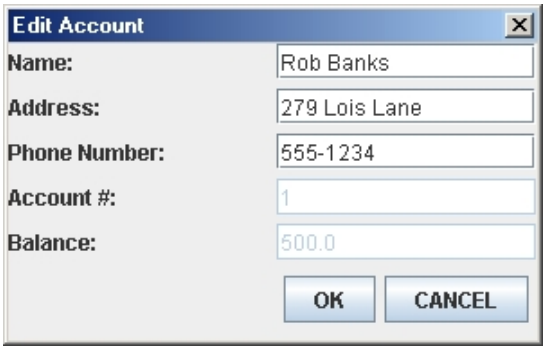
In some cases, we may not want the user of the dialog box to decide whether or not it should be modal, nor may we want them to specify the title. We can simply hard-code these into the dialog box if we wish:

```
public MyDialog(Frame owner, ClassA modelA, ClassB modelB, ...) {

   super(owner, "Mt Cool Dialog Box", true);
   ...

}
```

In addition to this, we will use the **`dispose()`** message to dispose of (i.e., close and delete) the dialog box from within your code.

Example:

In this example we will create a **BankAccountDialog** that allows us to modify a BankAccount's owner name, address and phone number. It will also show us the account number and balance, but we will not be allowed to alter that information. We use an extended version of the **BankAccount** class that was used in COMP1405/1005 (we added here an address and phone number). We will design the dialog box to look like this:



Notice that it shows the name, address, phone number, account number and balance, but that the account number and balance are disabled (grayed out) so that we cannot edit them. Also, notice the OK and CANCEL buttons (which typically appear at the bottom right of a dialog box). Also, notice that there are no minimize or maximize buttons on the titlebar ... we will make this window non-resizable.

We will create this window by first creating a **JPanel** containing the 5 text fields and their labels. We will set it up as the *view* part of the window so that the panel class will not have any behaviour, it will simply allow us to access the 3 editable fields through public **get** methods.

```
import java.awt.*;
import javax.swing.*;
public class BankAccountPanel extends JPanel {
    // The components needed to be used outside of this class
    private JTextField nameTextField;
    private JTextField addressTextField;
    private JTextField phoneTextField;

    // Make a get method so that the name/address/phone can be accessed
externally
    public JTextField getNameTextField() { return nameTextField; }
    public JTextField getAddressTextField() { return addressTextField; }
    public JTextField getPhoneTextField() { return phoneTextField; }

    // Add a constructor that takes a BankAccount, so that we can populate the
```

```
text fields
    public BankAccountPanel (BankAccount account) {

        // Fill in the text fields with bank account's information
        nameTextField = new JTextField(account.getName());
        addressTextField = new JTextField(account.getAddress());
        phoneTextField = new JTextField(account.getPhone());

        JTextField accField = new
JTextField(String.valueOf(account.getAccountNumber()));
        JTextField balField = new
JTextField(String.valueOf(account.getBalance()));

        // Disallow changing of balance and account number
        accField.setEnabled(false);
        balField.setEnabled(false);

        // Set the layoutManager and add the components
        setLayout(new GridLayout(5,3,5,5));
        add(new JLabel("Name:"));
        add(nameTextField);
        add(new JLabel("Address:"));
        add(addressTextField);
        add(new JLabel("Phone Number:"));
        add(phoneTextField);
        add(new JLabel("Account #:"));
        add(accField);
        add(new JLabel("Balance:"));
        add(balField);
    }
}
```

Notice:

- The constructor takes a BankAccount object. This is used to "fill-in" the initial values of the panel.
- instance variables & get methods are made only for the name/address/phone text fields since the other text fields may not be changed (i.e., they are disabled).
- The code here does not make changes to the model bank account in any way!!!

Now let us use this panel in our dialog box:

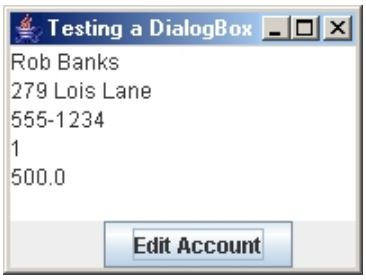
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class BankAccountDialog extends JDialog {
    private BankAccount account;  // The model

    // The buttons and main panel
    private JButton okButton;
    private JButton cancelButton;
    private BankAccountPanel bankAccountPanel;

    // A constructor that takes the model and client as parameters
    public BankAccountDialog(Frame owner, String title, boolean modal,
BankAccount acc) {

        // Call the super constructor that does all the work of setting up the
dialog
        super(owner,title,modal);

        account = acc;  // Store the model

        // Make a panel with two buttons (placed side by side)
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
        buttonPanel.add(okButton = new JButton("OK"));
        buttonPanel.add(cancelButton = new JButton("CANCEL"));

        // Make the dialog box by adding the bank account panel and the button
panel
        setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
        bankAccountPanel  = new BankAccountPanel(account);
```

```
            add(bankAccountPanel);
            add(buttonPanel);

            // Prevent the window from being resized
            setResizable(false);

            // Listen for ok button click
            okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event){
                    okButtonClicked();
            }});

            // Listen for cancel button click
            cancelButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event){
                    cancelButtonClicked();
            }});

            // Listen for window closing: treat like cancel button
            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent event) {
                    cancelButtonClicked();
            }});

            // Set the size of the dialog box
            setSize(300, 190);
        }

        private void okButtonClicked(){
            // Update model to show changed owner name
            account.setName(bankAccountPanel.getNameTextField().getText());
            account.setAddress(bankAccountPanel.getAddressTextField().getText());
            account.setPhone(bankAccountPanel.getPhoneTextField().getText());

            // Tell the client that ok was clicked, in case something needs to be
    done there
            if (getOwner() != null)
                ((DialogClientInterface)getOwner()).dialogFinished();

            dispose();  // destroy this dialog box
        }

        private void cancelButtonClicked(){
            // Tell the client that cancel was clicked, in case something needs to
    be done there
            if (client != null)
                ((DialogClientInterface)getOwner()).dialogCancelled();

            dispose();  // destroy this dialog box
        }
    }
```

Notice:

- The window is made non-resizable by using **setResizable(false)**;
- When the OK button is clicked, the name/address/phone text of the dialog box is accessed and updated in the model.
- The client is informed when OK or CANCEL is clicked.
- After informing the client, the dialog box is disposed of.
- Window closing is treated the same as when pressing the CANCEL button.

So, how do we test out this dialog box ? We should create an application that opens it (perhaps due to a button press). Here is the application that we will create:



This application will maintain a **BankAccount** object as the model and show its contents in a **JTextArea** object. When the user clicks the **Edit Account** button, we will create/open the dialog box. The dialog box will be ***modal***, so we will have to finish working with it before we go back to the main window here. Once the dialog box has been closed, any changes that were made should be reflected in the text area. Here is the code:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class BankAccountEditor extends JFrame implements DialogClientInterface,
ActionListener {
    private BankAccount model;
    private JTextArea info;

    public BankAccountEditor(String title, BankAccount account){
        super(title);
        model = account; // store the model

        // create a text area and an edit button
        info = new JTextArea();
        JButton editButton = new JButton("Edit Account");
        editButton.addActionListener(this);
        setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
        add(info);
        add(editButton);

        update();  // fill in the text area

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(200, 150);
    }

    // Handle the EDIT button
    public void actionPerformed(ActionEvent e) {
        // Create a new dialog box
        BankAccountDialog dialog =  new BankAccountDialog (this, "Edit Account",
true, model);

        System.out.println("About to open the dialog box ...");

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

        System.out.println("Dialog box has been closed.");
    }

    public void dialogFinished() {
        System.out.println("Changes accepted, Account has been changed");
        update();
    }

    public void dialogCancelled() {
        System.out.println("Changes aborted, Account has not been changed");
        //no need to call update, since nothing has changed
    }

    private void update() {
        //update the info text area to reflect the account balance
        info.setText(
            model.getName() + '\n' + model.getAddress() + '\n' +
            model.getPhone() + '\n' + model.getAccountNumber() + '\n' +
            model.getBalance());
    }
```

```
    public static void main(String args[]) {
        BankAccount b = new BankAccount("Rob Banks", "279 Lois Lane", "555-1234");
        b.deposit(500);

        BankAccountEditor frame = new BankAccountEditor("Testing a DialogBox", b);
        frame.setVisible(true);
    }
}
```

Try out the code yourself. Notice that the dialog box is indeed modal. Try changing information in the dialog box and see if changes are reflected back in the editor application window. Notice that when the dialog box is not modal, we can have multiple instances of it open at the same time.

6.4 E-mail Buddy Dialog Box Example

Here we discuss another example application that shows the use of a dialog box. Consider having many buddies that you send e-mail to. 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 model of an **EmailBuddy** as follows:

```
// This class represents a "buddy" whose email address is kept.
// An additional boolean indicates whether or not this is a
// friend that is "hot" (i.e. contacted often)
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);
    }
}
```

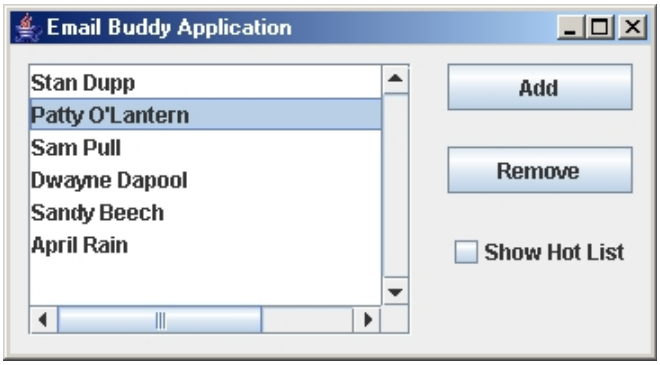
As can be seen, there is nothing difficult here ... just your standard "run-of-the-mill" model class. However, this class alone doe not represent the whole model for our GUI since we will have many of these **EmailBuddy** objects. So, we will make a **Vector** of them when we make the interface.

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

// This is the Panel that contains represents the view of the
// Email buddy application.
public class EmailBuddyPanel extends JPanel {

    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(){
        super();

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

        // Add the buddy list
        buddyList = new JList();

buddyList.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
        JScrollPane scrollPane = new JScrollPane( buddyList,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        layoutConstraints.gridx = 0;  layoutConstraints.gridy = 0;
        layoutConstraints.gridwidth = 3;  layoutConstraints.gridheight = 5;
        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 = 3;  layoutConstraints.gridy = 0;
            layoutConstraints.gridwidth = 1;  layoutConstraints.gridheight = 1;
            layoutConstraints.fill = GridBagConstraints.BOTH;
            layoutConstraints.insets = new Insets(10, 10, 10, 10);
            layoutConstraints.anchor = GridBagConstraints.EAST;
            layoutConstraints.weightx = 0.0;  layoutConstraints.weighty = 0.0;
            layout.setConstraints(addButton, layoutConstraints);
            add(addButton);

            // Add the Remove button
            removeButton = new JButton("Remove");
            layoutConstraints.gridx = 3;  layoutConstraints.gridy = 1;
            layoutConstraints.gridwidth = 1;  layoutConstraints.gridheight = 1;
            layoutConstraints.fill = GridBagConstraints.BOTH;
            layoutConstraints.insets = new Insets(10, 10, 10, 10);
            layoutConstraints.anchor = GridBagConstraints.EAST;
            layoutConstraints.weightx = 0.0;  layoutConstraints.weighty = 0.0;
            layout.setConstraints(removeButton, layoutConstraints);
            add(removeButton);

            // Add the ShowHotList button
            hotListButton = new JCheckBox("Show Hot List");
            layoutConstraints.gridx = 3;  layoutConstraints.gridy = 3;
            layoutConstraints.gridwidth = 1;  layoutConstraints.gridheight = 1;
            layoutConstraints.fill = GridBagConstraints.BOTH;
            layoutConstraints.insets = new Insets(10, 10, 10, 10);
            layoutConstraints.anchor = GridBagConstraints.EAST;
            layoutConstraints.weightx = 0.0;  layoutConstraints.weighty = 0.0;
            layout.setConstraints(hotListButton, layoutConstraints);
            add(hotListButton);
        }
    }
```

Notice that there is nothing really new here either. We did however, make some "get" methods for the components so that we can access them from outside this class.

Now for the actual dialog box. Ask yourself these questions:

1. What is the purpose of the dialog box ?
2. What causes the dialog box to appear ?

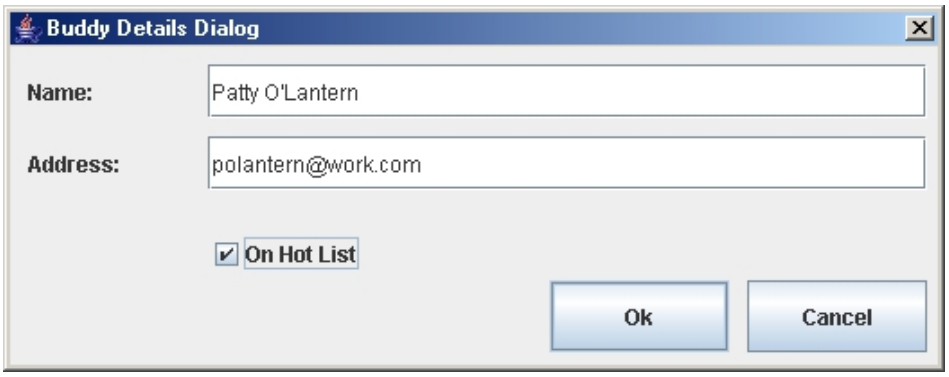
The dialog box is used to enter information/details about a particular buddy. It should appear when the user attempts to add an **EmailBuddy**. If we "*play our cards just right*", we will be able to use the same dialog box to allow an "already existing" **EmailBuddy** to have his/her details changed.

Here are some more questions that need to be answered:

- What information should the dialog box show ?
- What information needs to be changed by the dialog box ?
- What is a good descriptive title for the dialog box ?
- How can the information be shown nicely ?

Here are the answers:

1. The dialog box should show all information about a particular EmailBuddy. This includes name, address and hot list status.
2. The user should be able to change all 3 pieces of information about the buddy
3. We should call it something like "Buddy Details" or "Buddy Information"
4. Lay it out nicely. Here is an idea:



Don't forget that we need to add the OK and CANCEL buttons as well. Also, there is no need to be able to resize the dialog box so we can just disable the resizing.

Below is a method that will be called from our dialog class to add the components to the dialog box. It will take an **EmailBuddy** object as a parameter so that when the dialog box opens, we can populate the text fields with values indicating the **EmailBuddy**'s current information. This parameter will represent the model that is affected by the dialog box.

```
// This code adds the necessary components to the interface
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);
}
```

We will now look at the code needed to create the dialog box and get its behaviour working correctly:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
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(Frame 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);
    }

    private void buildDialogWindow(EmailBuddy aBuddy) {
        // This code is given above
    }

    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();
    }
}
```

Once again, we see that we just add listeners for the **OK** and **CANCEL** buttons as well as the window closing event. Then we merely make methods that are called for each.

Notice that when the **OK** button is clicked, the 3 pieces of changed buddy data are stored in the model buddy so that the buddy will have been altered by this dialog box. Then we inform the client that **OK** was pressed. For the cancel button, there is no work to do, just informing the client that **CANCEL** was pressed.

We are not done yet ! Now we need to work on the actual application that will be calling the dialog box.

We will call the class **`EmailBuddyApp`** and it will extend **JFrame**. It will be the class that opens the dialog box and so it must implement the `DialogClientInterface`. We will need to store the buddies that we will be making, so we make a Vector as an instance variable. We will first make the application work such that we will be able to add buddies to the list. Here is the basic framework for the application:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;
public class EmailBuddyApp extends JFrame implements DialogClientInterface {
    // Store the model as a vector of email buddies
    private Vector      buddies;

    // Store the view that contains the components
    EmailBuddyPanel     view;

    // Here are the component listeners
    ActionListener      theAddButtonListener;

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

        // Initially, no buddies
        buddies = new Vector();

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

        // Make a listener for the add button
        theAddButtonListener = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                addBuddy();
            }};
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600,300);

        // Start off with everything updated properly
        update();
    }

    // Enable all listeners
    private void enableListeners() {
        view.getAddButton().addActionListener(theAddButtonListener);
    }

    // Disable all listeners
    private void disableListeners() {
        view.getAddButton().removeActionListener(theAddButtonListener);
    }

    // This is called when the user clicks the add button
    private void addBuddy() {
        EmailBuddy  aBuddy = new EmailBuddy();

        // Add the buddy to the end of the Vector
        buddies.add(aBuddy);

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

    // Called when the dialog box is closed with the Ok button
    public void dialogFinished() {
        update();
    }

    // Called when the dialog box is closed with the cancel button or manually closed
```

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

    // Update the list
    private void updateList() {
        // Update the list contents and select the last buddy
        view.getBuddyList().setListData(buddies);
        view.getBuddyList().setSelectedValue((EmailBuddy)buddies.get(buddies.size()-1),
true);
    }

    // Update the GUI
    private void update() {
        disableListeners();
        updateList();
        enableListeners();
    }

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

Perhaps the most interesting portions of the code are the **addBuddy**(), **dialogCancelled()** and **updateList()** methods. When the user adds a buddy, we:

- make a new Buddy
- add the buddy to the end of the Vector of buddies
- open a dialog box on this buddy

Since the dialog box is modal, nothing else happens until the dialog box is closed. When closed, either the **dialogFinished** or **dialogCancelled** methods are called. If the OK button was pressed, then **dialogFinished** is called and there is no work to be done except to update the screen. This is because the dialog box already made the appropriate changes to the EmailBuddy and we merely need to reflect the changes in the interface. If **dialogCancelled** was called, then the user has canceled his "request to make changes" and therefore we need to remove the buddy that we added just before the dialog box was opened. We do not need to update anything however, since the interface appearance will not have changed.

Now ... what about the remove button ? To get the remove button to work, we will make some additions and changes to the code. What buddy gets removed ? Probably the one that is currently selected from the list. Here are the additions:

1. Add a new instance variable:

```
ActionListener        theRemoveButtonListener;
```

2. Make a new listener in the constructor:

```
theRemoveButtonListener = new ActionListener() {
  public void actionPerformed(ActionEvent event){
        removeBuddy();
   }};
```

3. Add these lines to the **enableListeners()** and **disableListeners()** methods, respectively:

```
view.getRemoveButton().addActionListener(theRemoveButtonListener);
view.getRemoveButton().removeActionListener(theRemoveButtonListener);
```

4. Add this method to do the removing:

```
private void removeBuddy() {
   EmailBuddy aBuddy = (EmailBuddy)(view.getBuddyList().getSelectedValue());

    if (aBuddy != null) {
        buddies.remove(aBuddy);
```

```
            update();
        }
    }
```

5. Add a line to the **`update()`** method (after the list is updated):

```
    private void update() {
        disableListeners();
        updateList();
        updateRemove();
        enableListeners();
    }
```

6. Add this method to update the remove button so that it is disabled when nothing is selected in the list:

```
    private void updateRemove() {
        view.getRemoveButton().setEnabled(view.getBuddyList().
                                          getSelectedValue() != null);
    }
```

Notice however, that the **Remove** button code is dependent on the buddy list **JList**. That is, it accesses the selected value from this list. In order to keep our interface clean, we should alter the code by storing the selected value in a field and accessing this from the Remove button code. To do this, we create an instance field called **selectedBuddy** and add a listener to the list to set it:

```
// Add these instance variable
private EmailBuddy              selectedBuddy;
private ListSelectionListener   buddyListSelectionListener;

// In the constructor, make a listener to allow selection of buddies from the list
buddyListSelectionListener = new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent event){
        selectBuddy();
    }};

// This is called when the user selects a buddy from the list
private void selectBuddy() {
    selectedBuddy = (EmailBuddy)(view.getBuddyList().getSelectedValue());
    update();
}

// In the constructor, set the field to null
selectedBuddy = null;
```

- Change these methods now to use the **selectedBuddy**:

```
    private void updateRemove() {
        view.getRemoveButton().setEnabled(selectedBuddy != null);
    }

    private void removeBuddy() {
       if (selectedBuddy != null) {
            buddies.remove(selectedBuddy);
            update();
            selectBuddy();
        }
    }
```

There are a few other places that we can now start using the **selectedBuddy** field.

What about the hot list ? Well we have to "hide" some of the buddies when it is on. We will have to make a listener as well so that when the check box is toggled, the changes occur right away. Here are the additions:

1. Add a new instance variable:

```
    ActionListener    hotListListener;
```

2. Make a new listener in the constructor:

```
hotListListener = new ActionListener() {
   public void actionPerformed(ActionEvent event){
        toggleHotList();
    }
};
```

3. Add these lines to the **`enableListeners()`** and **`disableListeners()`** methods, respectively:

```
 view.getHotListButton().addActionListener(hotListListener);
 view.getHotListButton().removeActionListener(hotListListener);
```

4. Add this method to do the toggling:

```
private void toggleHotList() {
    update();
}
```

5. Change the **`updateList()`** method so that we display the appropriate items in the list:

```
private void updateList() {
    boolean  foundSelected = false;

    // If the hot list is on, find all buddies that are on the hot list
    if (view.getHotListButton().isSelected()) {
        Vector temp = new Vector();

       for (int i=0; i<buddies.size(); i++) {
            EmailBuddy aBuddy = (EmailBuddy)buddies.get(i);
           if (aBuddy.onHotList()) {
                temp.add(aBuddy);
               if (aBuddy == selectedBuddy)
                    foundSelected = true;
            }
        }
        view.getBuddyList().setListData(temp);
        if (!foundSelected)
            selectedBuddy = null;
    }
    else
        view.getBuddyList().setListData(buddies);

    if (selectedBuddy != null)
        view.getBuddyList().setSelectedValue(selectedBuddy, true);
}
```

Notice that we look for which buddy is currently selected and update the **selectedBuddy** variable accordingly.

What if we want to **edit** a buddy, not just to add one ? We will have to decide what action will cause the editing to take place. One approach is to have the user double click on the buddy in the list and the dialog box will come up with that buddy's details within it. We can then make changes to the data and close the dialog box.

To do this, notice that we currently have some small problems:

- When **CANCEL** is clicked, we are removing the last EmailBuddy from the list. This should not happen when editing.

We can solve this problem by keeping a boolean flag which indicates whether or not we are in the midst of adding a buddy or whether or not we are editing. Here are the changes:

1. Add a new instance variable to maintain the current "mode" (i.e., add or edit). Also, we will be making a listener for the double-click action.

```
private boolean       inAddMode;
MouseListener         doubleClickSelectionListener;
```

2. Make a new listener in the constructor:

```
doubleClickSelectionListener = new MouseAdapter() {
    public void mouseClicked(MouseEvent event){
```

```
            if (event.getClickCount() == 2)
                editBuddy();
        }};
```

3. Add these lines to the **enableListeners()** and **disableListeners()** methods, respectively:

```
view.getBuddyList().addMouseListener(doubleClickSelectionListener);
view.getBuddyList().removeMouseListener(doubleClickSelectionListener);
```

4. Add this method to do the editing. It brings up a dialog box for the selected item in the list:

```
    private void editBuddy() {
        inAddMode = false;
        if (view.getBuddyList().getSelectedValue() == null) return;
        BuddyDetailsDialog dialog =
            new BuddyDetailsDialog(this, "Buddy Details Dialog", true,
selectedBuddy);
        dialog.setVisible(true);
    }
```

Notice that we check to make sure there is a buddy selected before we open a dialog box.

5. Add this line to the **addBuddy()** method before the dialog box is opened:

```
inAddMode = true;
```

6. Make this change to the **dialogCancelled()** method:

```
    public void dialogCancelled() {
        if (inAddMode) {
            // Remove the latest buddy that was added if in add mode
            buddies.remove(buddies.get(buddies.size()-1));
        }
        inAddMode = false;
        update();
    }
```

We are almost done now. The remove button does not refresh properly. That is, when nothing is selected, the remove button is disabled. If we then make a selection in the list (i.e., single click), the button still remains disabled. We should fix this. Do you know how ? Of course you do. Just add an event handler for making list selections. It just needs to call **update()** and the remove button will re-enable itself.

---

7 More Collections: Sets and HashMaps

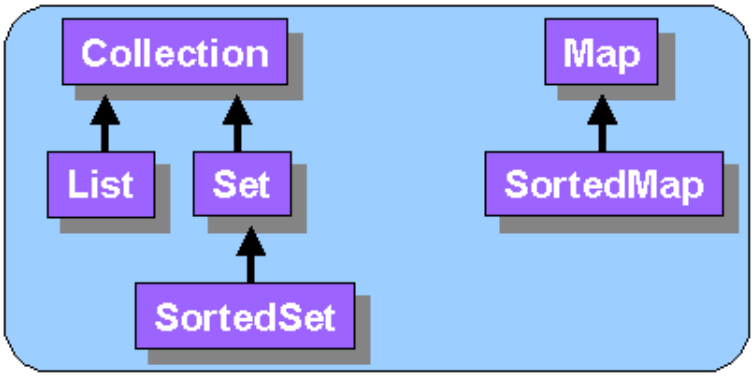
What's in This Set of Notes ?

In computer science, we are often concerned about issues pertaining to efficiency. That is, we always want to write code that is fast and efficient. In terms of collections, we want to make sure that when we put stuff in there, we can find it quickly. Think about how you unpack your groceries. Don't you place things in your cupboards in an organized way ? It helps you find what you are looking for quickly. Also, how will you find the items at the grocery store if they are not organized by sections ? **HashMaps** are collections in JAVA which store objects like **ArrayLists**. However, the items are stored in a **HashMap** in a way that the items can be retrieved quickly, when compared to an **ArrayList**. We will discuss the **Set, TreeSet, HashSet, Map, HashMap/Hashtable, TreeMap** collections in JAVA and also give a larger example using **HashMaps** to store data in a MovieStore.


Here are the individual topics found in this set of notes (click on one to go there):

7.1 Collections Re-Visited

In COMP1405/1005, we examined some **Collection** classes and worked quite a bit with **ArrayLists.** We also briefly looked at the **List** interface. Recall that there were other **Collection** classes as well that implemented the **Set** and **Map** interfaces:



Recall that an interface merely specifies a list of method signatures ... not actual code. Recall as well that the **Collection** interface defined many messages. Below is a list of a few of these grouped as querying methods and modifying methods.

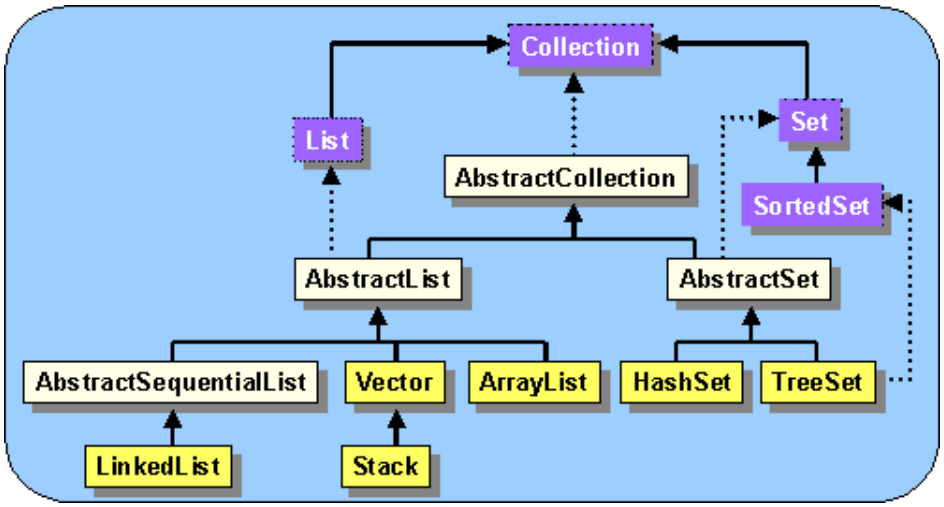
Querying methods (returns some kind of value):

- **size()** - returns the number of elements in the collection
- **isEmpty()** - returns whether or not there are any elements in the collection
- **contains(Object obj)** - returns whether or not the given object is in the collection (uses **.equals()** method for comparison)
- **containsAll(Collection c)** - same as above but looks for ALL elements specified in the given collection parameter.

Modifying methods (changes the collection in some way):

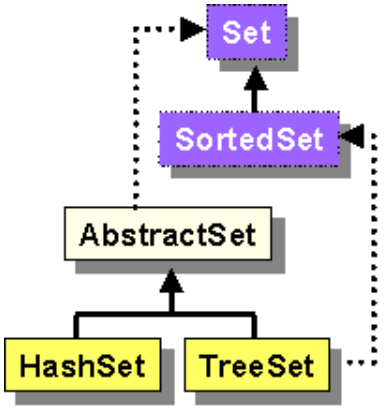
- **add(Object obj)** - adds the given object as an element to the collection (location is not specified)
- **addAll(Collection c)** - same as above but adds ALL elements specified in the given collection parameter.
- **remove(Object obj)** - removes the given object from the collection (uses **.equals()** method for comparison)
- **removeAll(Collection c)** - same as above but removes ALL elements specified in the given collection parameter.
- **retainAll(Collection c)** - same as above but removes all elements EXCEPT THOSE specified in the given collection parameter.
- **clear()** - empties out the collection by removing all elements.

Do you remember the hierarchy of classes implementing the **Collection** interface ?



7.2 The Set Classes

Let us continue where we left off from COMP1405/1005 and discuss the **Set** interface as well as the subclasses that implement it: **HashSet** and **TreeSet.**



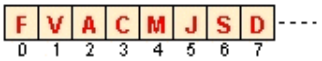
The **Set** classes are much like **Lists**, except that they do not allow duplicates.

- there cannot be two elements e1 and e2 such that e1.equals(e2)
- any modifications to the elements that affect the equality results in unspecified behaviour
- attempts to add duplicates are not satisfied

There are 2 main **Set** implementations:

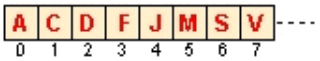
HashSet

- elements are **not kept in order**
- **fast adding/removing** operations
- **searching is slow** for a particular element
- elements MUST implement **`.equals()`** and **`.hashCode()`**



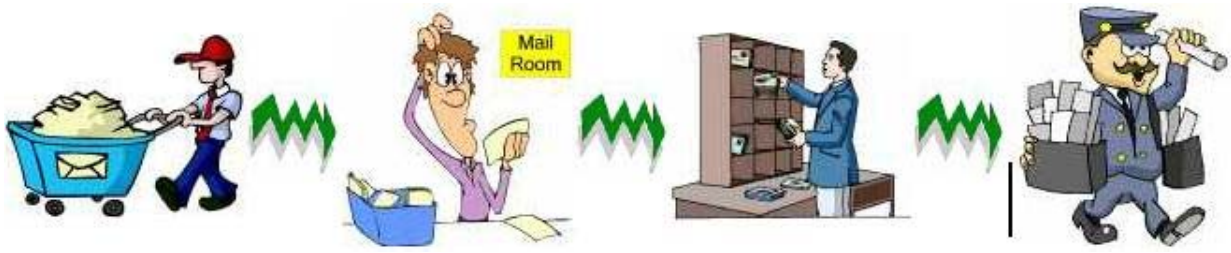
TreeSet

- elements are kept in **sorted order**, but not indexable
- order is user-defined (default is ascending)
- **adding is slow** since it takes longer to find the proper location
- **searching is fast** now since items are in order



Hashing is a high speed scheme for converting sparse (i.e., quite spread apart) keys into unique array subscripts. Hashing essentially generates for each item, a number (called the ***hashcode***) which is used as the ***key*** (or index). The generated number has no significant meaning other than to allow efficient storage and retrieval of data.

It is used as a way of getting quickly to the "vicinity" of the object you are looking for. For example, this is exactly what post offices do when they sort mail. They use the postal code to determine "roughly" where in the city your mail should go. People living in the same area have the same postal code, so it is easier for the post office to locate, gather and deliver your mail.



So **HashSets** (as well as **Hashtables** and **HashMaps** which we will see later) store their objects according

to a ***hash function*** which:

- returns an integer representation of the object (i.e., its *hashcode*).
- the integer should reflect the object's characteristics.
- equal objects should return the same hashcode.

Let us now look at an example that helps us understand the differences between these two sets.

Consider this code that adds some **BankAccounts** to an **ArrayList**:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack", "Zack",
"Sally"};
ArrayList<String> aCollection = new ArrayList<String>();

// Fill up the collection
for(int i=0; i<names.length; i++)
    aCollection.add(names[i]);

// Now print out the elements
for(String s: aCollection)
    System.out.println(s);
```

Here is the output, as expected:

```
Al
Zack
Sally
Al
Mel
Zack
Zack
Sally
```

Consider what happens when we replace the line:

```
ArrayList<String> aCollection = new ArrayList<String>();
```

with:

```
HashSet<String> aCollection = new HashSet<String>();
```

When we run the code, we obtain the following output:

```
Mel
Al
Sally
Zack
```

What happened to the duplicates ? They were removed. We only see the unique names coming out on the console window. But hold on! This is not the order that we added the names into the collection!! Well ... recall that the elements are not kept in any kind of order, and in fact, JAVA has its own pre-determined order based on hashcode values of the elements in which we added.

Now what about a **TreeSet** ? Shouldn't that keep them in order ?

Consider what happens when we replace:

```
HashSet<String> aCollection = new HashSet<String>();
```

with:

```
TreeSet<String> aCollection = new TreeSet<String>();
```

Now we obtain this output:

```
Al
Mel
Sally
Zack
```

Notice that the items are now sorted alphabetically. If we wanted to sort in a different order, we would need to make our own objects, instead of using Strings so that we could implement the **Comparable** interface.

What if we want to store **Customer** objects in the sets instead of simply strings:

```
public class Customer {
    private String name;

    public Customer(String n) { name = n; }
    public String getName() { return name; }
    public String toString() { return "Customer: " + name; }
}
```

Here is some code that makes a **HashSet** of **Customer** objects and prints them:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack", "Zack",
"Sally"};
HashSet<Customer> aCollection = new HashSet<Customer>();
for(int i=0; i<names.length; i++) {
    Customer c = new Customer(names[i]);
    aCollection.add(c);
}
for(Customer c: aCollection)
    System.out.println(c);
```

Here is the output:

```
Customer: Sally
Customer: Zack
Customer: Zack
Customer: Mel
Customer: Al
Customer: Sally
Customer: Al
Customer: Zack
```

Hey! There are duplicates! What's up with that ?

A unique **Customer** object is actually created for each customer and so they are all unique by default regardless of their name. How can we fix it so that only **Customers** with unique names can be added ? Recall that in a set, there cannot be two elements e1 and e2 such that **e1.equals(e2)**. But we don't have an **equals()** method in our **Customer** class! So in fact, we inherit a default one that checks identity, not equality. So ... we have to create our own **equals()** method for the **Customer** class:

```
public boolean equals(Object obj) {
    if (!(obj instanceof Customer))
        return false;
    return getName().equals(((Customer)obj).getName());
}
```

In addition, since objects are "hashed" in order to find their position in the ***Hash***Set, we must also implement a **hashCode()** method. The **hashCode()** method should return an integer that attempts to represent the object uniquely. Usually, it simply returns the combined hashcodes of its instance variables:

```
public int hashCode() {
    return name.hashCode();
}
```

Now when we run the code, we see that the duplicates are gone:

```
Customer: Mel
Customer: Al
Customer: Sally
Customer: Zack
```

We could change **HashSet** to **TreeSet** to get the items in sorted order. However, we would then need to make sure that our **Customer** objects are **Comparable**. We could add the following to our **Customer** class:

```
public class Customer implements Comparable {
    ...

    public int compareTo(Object obj) {
        if (obj instanceof Customer)
            return getName().compareTo(((Customer)obj).getName());
        else
            throw new ClassCastException();
    }
}
```

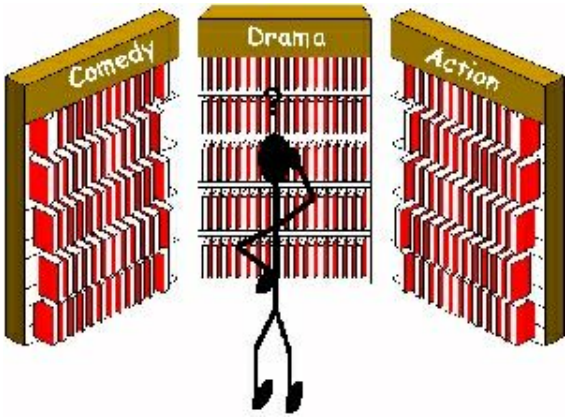
Then when we run the code, we see that the duplicates are gone and that the items are in sorted order:

```
Customer: Al
Customer: Mel
Customer: Sally
Customer: Zac
```

So remember ... we can use a **HashSet** or **TreeSet** to eliminate duplicates from a collection.

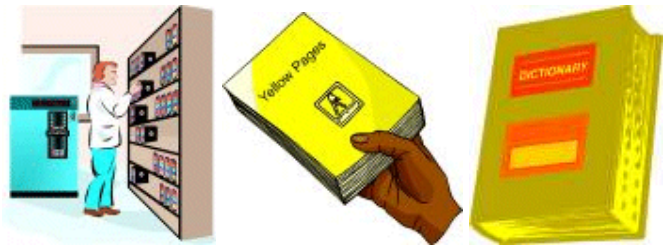
7.3 The Map Interface and Its Classes

It is often necessary to store objects in collections in such a way that they can be retrieved quickly later on. That is, there is a need for a collection which allows quick searching. For example, consider a video store. Isn't it a nice idea to have the movies arranged by category so that you don't waste time looking over movies that are not of an interesting nature (such as musicals or perhaps drama) ?



Like the **Collection** interface, the **Map** interface stores objects as well. So what is different ? Well, a **Map** stores things in a particular way such that the objects can be easily located later on. A ***Map*** really means a "Mapping" of one object to another. Maps are used when it is necessary to access the elements quickly by particular ***keys***. Examples are:

- storing many items by category (e.g., a video store is organized by sections (comedy/action/drama)).
- phone books where a value (e.g., number) is associated with a particular key (e.g., name).
- dictionaries where a value (e.g., definition) is associated with a particular key (e.g., word).



All **Maps** store a group of object pairs called ***entries***.

Each map ***entry*** has:

- ***key*** **-** identifies values uniquely (maps cannot have duplicate keys)
- ***value*** **-** accessed by their keys (each key maps to at most one value)

So, the ***key*** MUST be used to obtain a particular value from the **Map**.

The **Map** interface defines many messages:

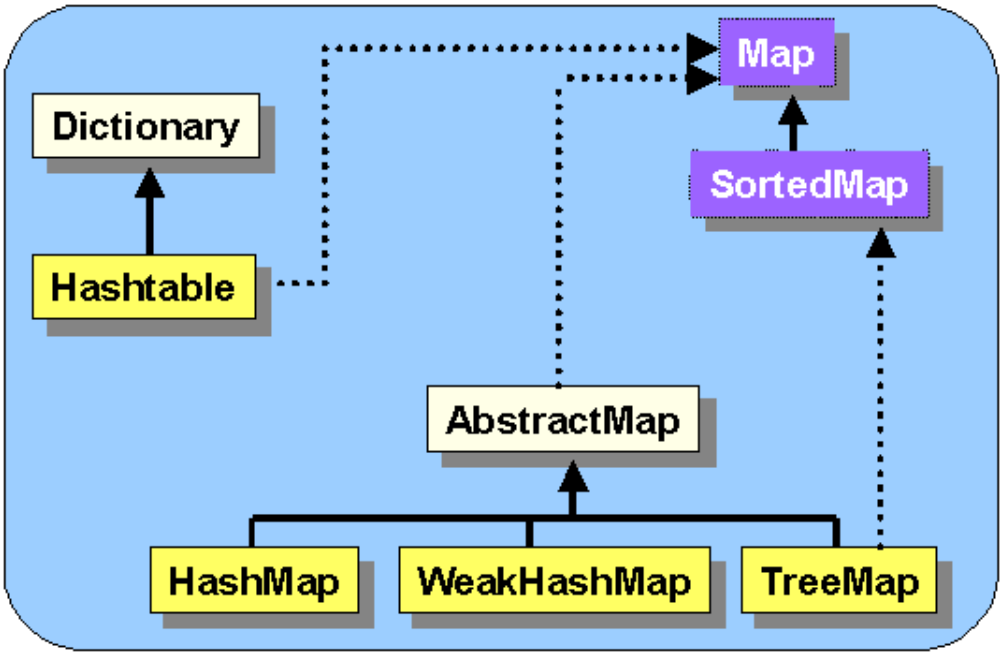
Querying methods (returns some kind of value):

- **size()** - returns the number of elements in the map.
- **isEmpty()** - returns whether or not there are any elements in the map.
- **containsKey(Object key)** - returns whether or not the given object is a key of the map (uses **.equals()** method for comparison).
- **containsValue(Object val)** - returns whether or not the given object is a value in the map (uses **.equals()** method for comparison).
- **getKey(Object key)** - returns the **Object** in the map that is associated with the given key.
- **values(Object key)** - returns a **Collection** containing all the values in the map.
- **keySet(Object key)** - returns a **Set** of all the keys in the map.
- **entrySet(Object key)** - returns a **Set** of all the key/value pairs in the map (i.e., **Map.Entry** objects represent key.value pairs).

Modifying methods (changes the collection in some way):

- **put(Object key, Object val)** - adds a new entry in the map with the given key and value. The key MUST be unique, otherwise this method replaces the value that was previously there by the given value.
- **putAll(Map m)** - adds all entries that are in the given map to the receiver map.
- **remove(Object key)** - removes the given key/value entry from the map based on the key only.
- **clear()** - empties out the map by removing all elements.

Here is the hierarchy showing most of the **Map** classes:



Notice that there are 4 main **Map** implementations:

Hashtable

- elements **not kept in order**
- **fast adding/removing**
- **fast searching** for a particular element
- elements MUST implement `.equals()` and `.hashCode()`
- cannot have **null** key
- synchronized

HashMap

- works similar to **Hashtables**
- can have one **null** key
- unsynchronized

TreeMap

- works similar to **HashMap**
- elements are **kept in sorted order**, but not indexable
- **slow adding** since it takes longer to find the proper location
- **efficient access and modification** methods

WeakHashMap

- works similar to **HashMap**
- **keys removed more efficiently** by the garbage collector.

7.4 HashMaps

HashMaps are not fixed-size, so they can grow as needed (just like with **ArrayLists**). Items are added to the **HashMap** by specifying the key AND the value. The key is used as a unique identifier that distinguishes its associated value from any other value. Both the keys and values can be arbitrary objects (but no **null** key for **Hashtables**). We will look at **HashMaps** here, but the **Hashtable** class works the

same way, but is synchronized (slower than HashMap).

To create a general **HashMap** that can store arbitrary keys and values we can use this constructor:

```
HashMap table = new HashMap();
```

However, typically we will use a **HashMap** that has all the same type of keys and the same type of values. In this case, we can specify the keys and values when declaring our variables. For example, if we want a HashMap where the keys are peoples names and the values for each person is a collection of items, we could declare our **HashMap** something like this:

```
HashMap<String,ArrayList> table = new HashMap<String,ArrayList>();
```

Of course, this will prevent us from making non-String keys and non-ArrayList values ... but it will simplify the usage of the class by eliminating the need for typecasting in many places where we access/modify the **HashMap**.

Remember, if you want to use your own created objects within **Hashtables, HashMaps, HashSets** or **TreeSets**, you MUST implement a **hashCode()** method. All objects inherit a default **hashCode()** from **Object**, but it may not be efficient. The **hashcode()** method should look similar to an **equals()** method in what it checks.

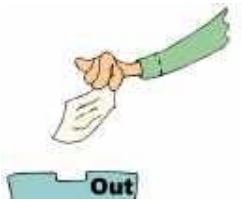
Here are some of the standard **HashMap** methods along with a simple example of how they work:

Method	Description and Example

`Object put(Object key, Object value)`
 
	Add the given value to the **HashMap** with the given key. If there is no value there for that key then **null** is returned, otherwise the original value in the **HashMap** is returned.
	`HashMap<String,String> aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`System.out.println(aPhoneBook); // displays {Norm=555-3789, Mark=555-2238, Arthur=555-8813}`

`Object get(Object key)`
 
	Return the value associated with the given key. If there is no value, **null** is returned.
	`HashMap<String,String> aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.get("Mark"); // returns "555-2238"`
	`aPhoneBook.get("Betty"); // returns null`
	`aPhoneBook.get("555-3789"); // returns null (must specify key, not value)`

`Object remove(Object key)`	Remove the key/value pair from the **HashMap** and return the value that was removed. If it is not there, return **null**.
	`HashMap<String,String> aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.remove("555-8813"); // No error but`




```
returns null
aPhoneBook.remove("Arthur");    // returns
"555-8813"
System.out.println(aPhoneBook);
// displays {Norm=555-3789, Mark=555-2238}
```

boolean isEmpty()

Return whether or not there are any values in the **HashMap**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.isEmpty(); // returns true
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.isEmpty(); // returns false
```

boolean containsKey(Object key)

Return whether or not the given key is in the HashMap.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
aPhoneBook.containsKey("555-8813"); // returns
false
aPhoneBook.containsKey("Mark");     // returns
true
aPhoneBook.remove("Mark");
aPhoneBook.containsKey("Mark");     // returns
false
```

boolean containsValue(Object value)

Return whether or not the given value is in the **HashMap**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
aPhoneBook.containsValue("Mark");      // returns
false
aPhoneBook.containsValue("555-3789");  // returns
true
aPhoneBook.remove("Norm");
aPhoneBook.containsValue("555-3789");  // returns
false
```

void clear()

Empty the **HashMap**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
System.out.println(aPhoneBook); // displays
{Mark=555-2238, Arthur=555-8813}
aPhoneBook.clear();
System.out.println(aPhoneBook); // displays {}
```

Collection **values()**

Return a **Collection** of the values in the **HashMap**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
```

	aPhoneBook.put("Norm", "555-3789");
	aPhoneBook.values(); // returns Collection
	[824-3789, 744-2238, 741-8813]

Set **keySet()**
 
	Return a **Set** of the keys in the **HashMap**.
	
	HashMap<String,String> aPhoneBook = new
	HashMap<String,String>();
	aPhoneBook.put("Arthur", "555-8813");
	aPhoneBook.put("Mark", "555-2238");
	aPhoneBook.put("Norm", "555-3789");
	aPhoneBook.keySet(); // returns Set [Norm, Mark,
	Arthur]

Set **entrySet()**
 
	Return a **Set** of the key/value pairs (i.e., as **Map.Entry** objects) in the **HashMap**.
	
	HashMap<String,String> aPhoneBook = new
	HashMap<String,String>();
	aPhoneBook.put("Arthur", "555-8813");
	aPhoneBook.put("Mark", "555-2238");
	aPhoneBook.put("Norm", "555-3789");
	aPhoneBook.entrySet(); // returns Set [Norm=824-3789,
	Mark=744-2238, Arthur=741-8813]

What is the difference between a **HashMap** and a **TreeMap** ? As with **HashSets** and **TreeSets**, **TreeMaps** maintain the keys in sorted order, whereas **HashMaps** do not maintain the keys in sorted order.

Consider this code:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack", "Zack",
"Sally"};
HashMap<String,Customer> aMap = new HashMap<String,Customer>();

// Fill up the collection
for (int i=0; i<names.length; i++)
    aMap.put(names[i], new Customer(names[i]));

System.out.println("Here are the keys:");
for (String key: aMap.keySet())
    System.out.println("    " + key);

System.out.println("Here are the values:");
for (Customer val: aMap.values())
    System.out.println("    " + val);

System.out.println("Here are the key/value pairs:");
for (Map.Entry pair: aMap.entrySet())
    System.out.println("    " + pair);
```

Here we see that a **HashMap** is formed with keys being the names of the **Customer** and values being the **Customers** themselves. This is the output:

```
Here are the keys:
    Mel
    Al
    Sally
    Zack
Here are the values:
    Customer: Mel
    Customer: Al
    Customer: Sally
    Customer: Zack
Here are the key/value pairs:
    Mel=Customer: Mel
```

```
    Al=Customer: Al
    Sally=Customer: Sally
    Zack=Customer: Zack
```

Notice that there are only 4 keys, even though we added many items ... this is because one key overwrites another when we call the **put()** method more than once with the same key.

If we replace the **HashMap** with a **TreeMap** in the above code, the code still works, we just get the items in sorted order according to the keys:

```
Here are the keys:
    Al
    Mel
    Sally
    Zack
Here are the values:
    Customer: Al
    Customer: Mel
    Customer: Sally
    Customer: Zack
Here are the key/value pairs:
    Al=Customer: Al
    Mel=Customer: Mel
    Sally=Customer: Sally
    Zack=Customer: Zack
```

Using **WeakHashMap**, you won't notice a difference. We will not talk about this class in this course.

7.5 The MovieStore Example

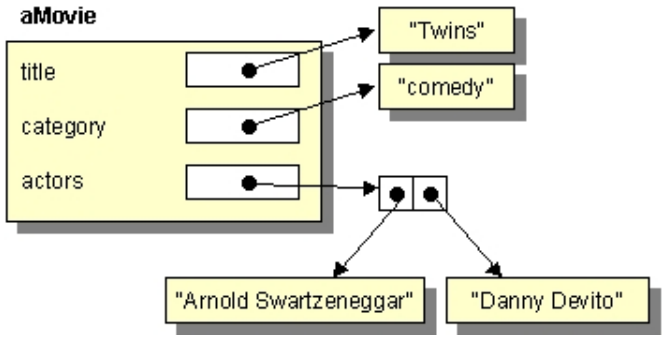
Consider an application which represents a movie store that maintains movies to be rented out. Assume that we have a collection of movies. When renting, we would like to be able to find movies quickly. For example, we may want to:

- ask for a movie by title and have it found right away
- search for movies in a certain category (e.g., new release, comedy, action)
- find movies containing a specific actor/actress (e.g., Jackie Chan, Peter Sellers, Jude Law etc...)

Obviously, we could simply store all moves in one big **ArrayList**. But how long would we waste finding our movies ? Imagine a video store in which the movies are not sorted in any particular order ... just randomly placed on the shelves !! We would have to go through them one by one !!!

We will use **HashMaps** to store our movies efficiently so that we can quickly get access to the movies that we want.

Let us start out with the representation of a **Movie** object. Each movie will maintain a **title**, list of **actors** and a **type** (category). Obviously, in a real system, we would need to keep much more information such as ID, rental history, new releases vs. oldies, etc... Here is the diagram representing the **Movie** object:



Let us now define this **Movie** class.

```
import java.util.*;
public class Movie {
    private String             title, type;
    private ArrayList<String>  actors;

    public String getTitle() { return title; }
    public String getType() { return type; }
    public ArrayList<String> getActors() { return actors; }
    public void setTitle(String aTitle) { title = aTitle; }
    public void setType(String aType) { type = aType; }

    public Movie() { this("???", "???"); }

    public Movie(String aTitle, String aType) {
        title = aTitle;
        type = aType;
        actors = new ArrayList<String>();
    }

    public String toString() { return("Movie: " + "\"" + title + "\""); }
    public void addActor(String anActor) { actors.add(anActor); }
}
```

Notice that there is no "set" method for the actors. We do not need it (just like in the autoshow example). Now lets look at the **`addActor()`** method. It merely adds the given actor (just a name) to the actors arrayList. We can make some example methods to represent some movies. Add the following methods to the **Movie** class:

```
    public static Movie example1() {
        Movie aMovie = new Movie("The Matrix","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example2() {
        Movie aMovie = new Movie("Blazing Saddles","Comedy");
        aMovie.addActor("Cleavon Little");
        aMovie.addActor("Gene Wilder");
        return aMovie;
    }

    public static Movie example3() {
        Movie aMovie = new Movie("The Matrix Reloaded","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example4() {
        Movie aMovie = new Movie("The Adventure of Sherlock Holmes' Smarter
Brother","Comedy");
        aMovie.addActor("Gene Wilder");
```

```
        aMovie.addActor("Madeline Kahn");
        aMovie.addActor("Marty Feldman");
        aMovie.addActor("Dom DeLuise");
        return aMovie;
    }

    public static Movie example5() {
        Movie aMovie = new Movie("The Matrix Revolutions","SciFic"));
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example6() {
        Movie aMovie = new Movie("Meet the Fockers","Comedy");
        aMovie.addActor("Robert De Niro");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Dustin Hoffman");
        return aMovie;
    }

    public static Movie example7() {
        Movie aMovie = new Movie("Runaway Jury","Drama");
        aMovie.addActor("John Cusack");
        aMovie.addActor("Gene Hackman");
        aMovie.addActor("Dustin Hoffman");
        return aMovie;
    }

    public static Movie example8() {
        Movie aMovie = new Movie("Meet the Parents","Comedy");
        aMovie.addActor("Robert De Niro");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Teri Polo");
        aMovie.addActor("Blythe Danner");
        return aMovie;
    }

    public static Movie example9() {
        Movie aMovie = new Movie("The Aviator","Drama");
        aMovie.addActor("Leonardo DiCaprio");
        aMovie.addActor("Cate Blanchett");
        return aMovie;
    }

    public static Movie example10() {
        Movie aMovie = new Movie("Envy","Comedy");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Jack Black");
        aMovie.addActor("Rachel Weisz");
        aMovie.addActor("Amy Poehler");
        aMovie.addActor("Christopher");
        return aMovie;
    }
```

Of course, we should test our class:

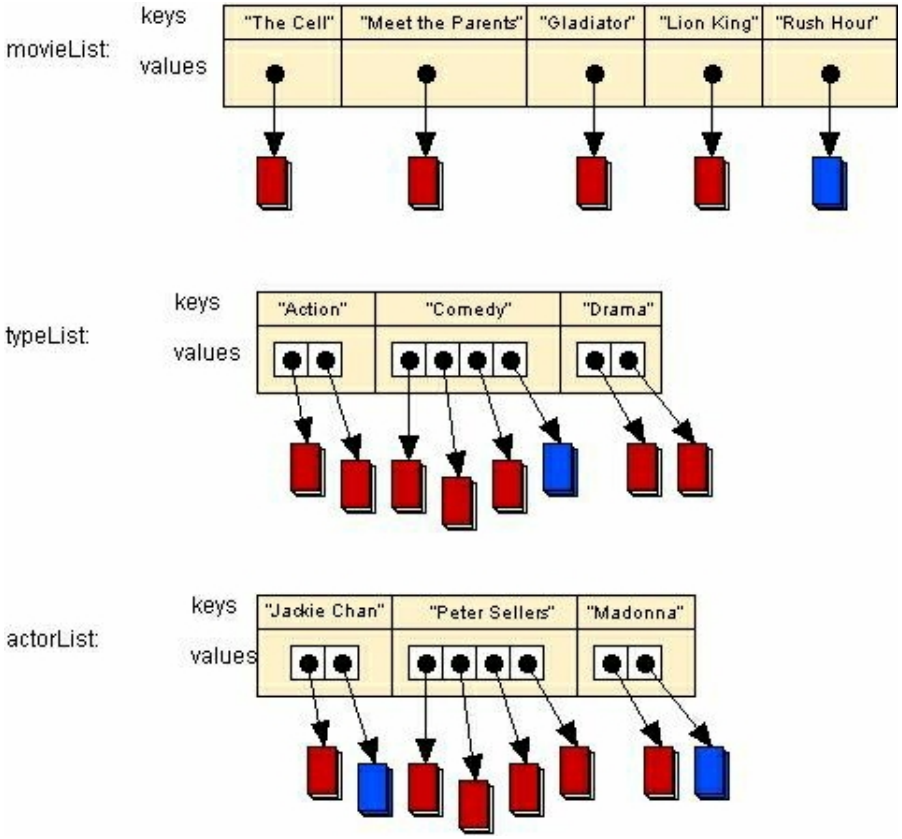
```
 public class MovieTester {
     public static void main(String args[]) {
         Movie     aMovie, anotherMovie;

         aMovie = Movie.example1();
         anotherMovie = Movie.example2();
         System.out.println(aMovie);
         System.out.println("is a " + aMovie.getType() +
                    " with actors " + aMovie.getActors());
         System.out.println(anotherMovie);
         System.out.println("is a " + anotherMovie.getType() +
                    " with actors " + anotherMovie.getActors());
     }
 }
```

Here is the output:

```
Movie: "The Matrix"
is a SciFic with actors [Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss]
Movie: "Blazing Saddles"
is a Comedy with actors [Cleavon Little, Gene Wilder]
```

Now we need to consider the making a **MovieStore** object. Recall, that we want to store movies efficiently using **HashMaps**.

For the **MovieStore**, we will maintain three **HashMaps**. One will be the **movieList** where the keys are titles and the values are the movie objects with that title. The second will be the **actorList** which will keep actor/actress names as keys and the values will be ArrayLists of all movies that the actor/actress stars in. The last one will be the **typeList** in which the keys will be the "types" (or categories) of movies and the values will be ArrayLists of all movies belonging to that type.



Notice that one of the movies is "blue" in the picture. Why ?
This represents the same exact movie. So in fact, the reference to this movie is stored in 4 different places.

Isn't this wasteful ? Keep in mind that we are not duplicating all the movie's data ... we are only duplicating the pointer to the movie. So in fact, each time we duplicate a movie in our **HashMaps**, we are simply duplicating its reference (or pointer) which takes 4 bytes.

So, yes, we are taking slightly more space, but at the benefit of allowing quick access to the data. You will learn more about efficiency when you do your second-year course on data structures.

The basic **MovieStore** definition is as follows:

```
import java.util.*
public class MovieStore {
    //These are the instance variables
    private HashMap<String,Movie>              movieList;
    private HashMap<String,ArrayList<Movie>>   actorList;
    private HashMap<String,ArrayList<Movie>>   typeList;

    //These are the get methods, not set methods are needed
```

```
    public HashMap<String,Movie> getMovieList() { return movieList; }
    public HashMap<String,ArrayList<Movie>> getActorList() { return actorList; }
    public HashMap<String,ArrayList<Movie>> getTypeList() { return typeList; }

    //This is the constructor
    public MovieStore() {
        movieList = new HashMap<String,Movie>();
        actorList = new HashMap<String,ArrayList<Movie>>();
        typeList  = new HashMap<String,ArrayList<Movie>>();
    }

    //This method returns a String representation of the Movie
    public String toString() {
        return ("MovieStore with " + movieList.size() + " movies.");
    }
}
```

Why do not we need "set" methods for the **HashMaps** ? You should be able to reason on that ;).

Now, how do we add a movie to the store ? Well ... how do the instance variables change ?

- the movie must be added to the movieList
- the movie must be added to the typeList. What if it is the first/last movie from this category ?
- the movie must be added to the actorList. What if it is the first/last movie for this actor ?

Here is the code:

```
//This method adds a movie to the movieStore.
public void addMovie(Movie aMovie) {
    //Add to the movieList
    movieList.put(aMovie.getTitle(), aMovie);

    //If there is no category yet matching this movie's type, make a new category
    if (!typeList.containsKey(aMovie.getType()))
        typeList.put(aMovie.getType(), new ArrayList<Movie>());

    //Add the movie to the proper category.
    typeList.get(aMovie.getType()).add(aMovie);

    //Now add all of the actors
    for (String anActor: aMovie.getActors()) {
        //If there is no actor yet matching this actor, make a new actor key
       if (!actorList.containsKey(anActor))
            actorList.put(anActor, new ArrayList<Movie>());

        //Add the movie for this actor
        actorList.get(anActor).add(aMovie);
    }
}
```

In fact, removing a movie is just as easy:

```
//This private method removes a movie from the movie store
private void removeMovie(Movie aMovie) {
    //Remove from the movieList
    movieList.remove(aMovie.getTitle());

    //Remove from the type list vector.  If last one, remove the type.
    typeList.get(aMovie.getType()).remove(aMovie);
    if (typeList.get(aMovie.getType()).isEmpty())
        typeList.remove(aMovie.getType());

    //Now Remove from the actors list.  If actor has no more, remove him.
    for(String anActor: aMovie.getActors()) {
        actorList.get(anActor).remove(aMovie);
        if (actorList.get(anActor).isEmpty())
            actorList.remove(anActor);
    }
}
```

However, what if we do not have a hold of the **Movie** object that we want to delete ? Perhaps we just know the title of the movie that needs to be removed. We can write a method which asks to remove a movie with a certain title. All it needs to do is grab a hold of the movie and then call the remove method that we just wrote.

```
//This method removes a movie (given its title) from the movie store
public void removeMovieWithTitle(String aTitle) {
    if (movieList.get(aTitle) == null)
        System.out.println("No movie with that title");
    else
        removeMovie(movieList.get(aTitle));
}
```

Well, perhaps the final thing we need to do is list the movies (or print them out). How do we do this ? What if we want them in some kind of order ? Perhaps any order, by actor/actress, or by type. Here's how to display them in the order that they were added to the MovieStore:

```
//This method lists all movie titles that are in the store
public void listMovies() {
    for (String s: movieList.keySet())
        System.out.println(s);
}
```

As you can see, with the automatic type-casting due to the generics of JAVA 1.5, everything is easy. What about listing movies that star a certain actor/actress ? Well it just requires an additional search. Can you guess what HashMap is needed ?

```
//This method lists all movies that star the given actor
public void listMoviesWithActor(String anActor) {
    for (Movie m: typeList.get(aType))
         System.out.println(m);
}
```

Lastly, let us list all of the movies that belong to a certain category (type). For example, someone may wish to have a list of all comedy movies in the store. It is actually very similar to the actor version.

```
//This method lists all movies that have the given type
public void listMoviesOfType(String aType) {
    for (Movie m: actorList.get(anActor))
        System.out.println(m);
}
```

Ok, now we better test everything:

```
public class MovieStoreTester {

     public static void main(String args[]) {
         MovieStore  aStore = new MovieStore();
         aStore.addMovie(Movie.example1());
         aStore.addMovie(Movie.example2());
         aStore.addMovie(Movie.example3());
         aStore.addMovie(Movie.example4());
         aStore.addMovie(Movie.example5());
         aStore.addMovie(Movie.example6());
         aStore.addMovie(Movie.example7());
         aStore.addMovie(Movie.example8());
         aStore.addMovie(Movie.example9());
         aStore.addMovie(Movie.example10());

         System.out.println("Here are the movies in: " + aStore);
         aStore.listMovies();
         System.out.println();

         //Try some removing now
         System.out.println("Removing The Matrix");
         aStore.removeMovieWithTitle("The Matrix");
         System.out.println("Trying to remove Mark's Movie");
         aStore.removeMovieWithTitle("Mark's Movie");
```

```
            //Do some listing of movies
            System.out.println("\nHere are the Comedy movies in: " + aStore);
            aStore.listMoviesOfType("Comedy");
            System.out.println("\nHere are the Science Fiction movies in: " +
        aStore);
            aStore.listMoviesOfType("SciFic");
            System.out.println("\nHere are the movies with Ben Stiller:");
            aStore.listMoviesWithActor("Ben Stiller");
            System.out.println("\nHere are the movies with Keanu Reeves:");
            aStore.listMoviesWithActor("Keanu Reeves");
        }

    }
```

Here is the output:

```
Here are the movies in: MovieStore with 10 movies.
Envy
Blazing Saddles
The Matrix
The Matrix Reloaded
Meet the Fockers
Meet the Parents
Runaway Jury
The Matrix Revolutions
The Adventure of Sherlock Holmes' Smarter Brother
The Aviator

Removing The Matrix
Trying to remove Mark's Movie
No movie with that title

Here are the Comedy movies in: MovieStore with 9 movies.
Movie: "Blazing Saddles"
Movie: "The Adventure of Sherlock Holmes' Smarter Brother"
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the Science Fiction movies in: MovieStore with 9 movies.
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"

Here are the movies with Ben Stiller:
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the movies with Keanu Reeves:
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"
```

8 Graphics

What's in This Set of Notes?

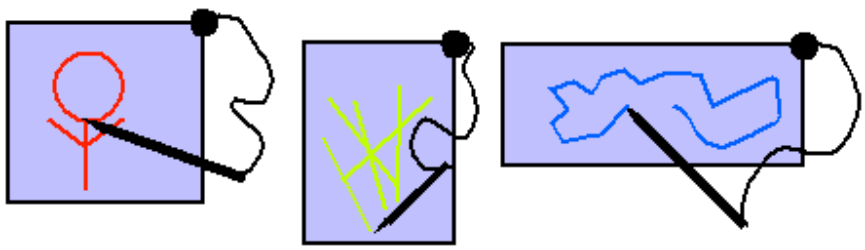
As programmers, we will likely all eventually come across a situation in which we need to display graphics. Graphics may be pictures or perhaps drawings consisting of lines, circles, rectangles etc... For example, if we want to have an application that displays a bar graph, there is no "magical" component in JAVA that does this for us. We will learn here the basics of displaying and manipulating graphics in our JAVA applications.

Here are the individual topics found in this set of notes (click on one to go there):

8.1 Doing Simple Graphics

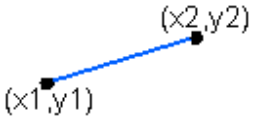
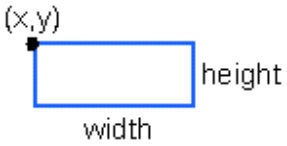
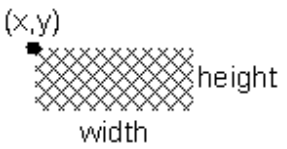
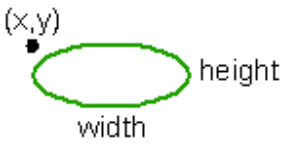
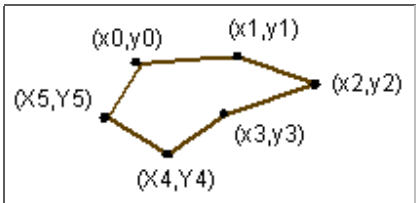
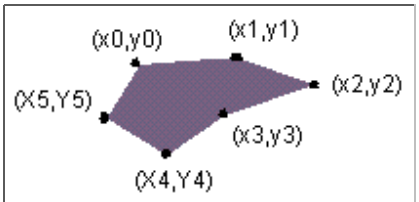
Graphics are used in many applications to display graphs, statistics, diagrams, pictures etc... Some applications are even completely based on graphics such as games, paint programs, MS PowerPoint etc... We have already seen that **ImageIcons** can be used to display images on your application window inside labels, buttons etc... Now we will see how to actually draw our own graphics, as when drawing graphs or diagrams.

The **java.awt** package has a class called **Graphics** that permits the drawing of various shapes. The class is **abstract** and so there is no constructor. Instead, JAVA provides a **getGraphics()** method that can be sent to any window component which returns an instance of this **Graphics** class (i.e., each component keeps an instance of that class by default). Think of each component having its own "pen" that can only be used to draw in that component's "space", just like the pens attached to kiosks at the bank.

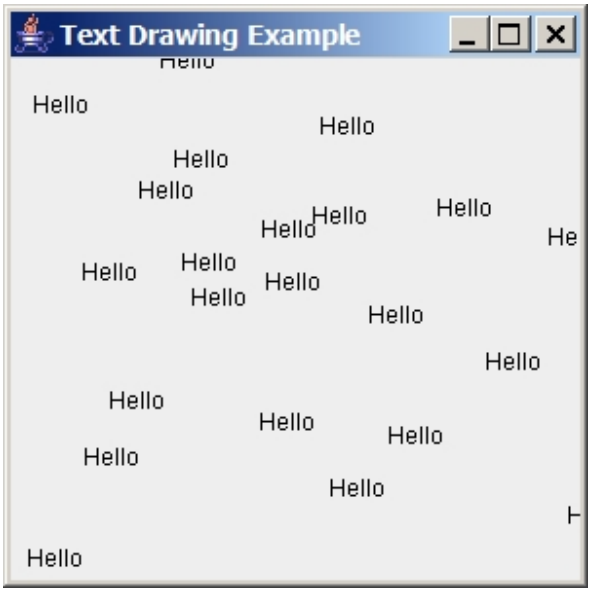


There are a set of drawing functions that allow you to draw onto a component's area. Since a particular graphics object belongs to one specific component, you can only draw on that component with it. Most drawing functions allow you to specify x and y coordinates. The coordinate `(x,y)=(0,0)` is at the top left corner of the component's area. So all coordinates are with respect to the component's area.

Here are just some of the methods available in the **Graphics** class (look in the JDK API for more info):

	`// Draw a line from (x1, y1) to (x2,y2)` `public abstract void drawLine(int x1, int y1, int x2, int y2);`
	`// Draw a rectangle with its top left at (x, y) having the given width and height` `public abstract void drawRect(int x, int y, int width, int height);`
	`// Draw a filled rectangle with its top left at (x, y) having the given width and height` `public abstract void fillRect(int x, int y, int width, int height);`
	`// Erase a rectangular area by filling it in with the background color` `public abstract void clearRect(int x, int y, int width, int height);`
	`// Draw an oval with its top left at (x, y) having the given width and height` `public abstract void drawOval(int x, int y, int width, int height);`
	`// Draw a filled oval with its top left at (x, y) having the given width and height` `public abstract void fillOval(int x, int y, int width, int height);`
	`// Draw the given String with its bottom left at (x, y)` `public abstract void drawString(String str, int x, int y);`
	`// Draw a polygon with the given coordinates` `public abstract void drawPolygon(int[] x, int[] y, int numEdges);`
	`// Draw a filled polygon with the given coordinates` `public abstract void fillPolygon(int[] x, int[] y, int numEdges);`
	`// Set the foreground and fill color of the Graphics object` `public abstract void setColor(Color c);`
ABcDEFG	`// Set the Font for use with drawString` `public abstract void setFont(Font font);`

Example:



This code makes a simple **JFrame** and then draws some text on it wherever the user clicks the mouse. As it turns out, we can draw directly to the frame of a window. We don't need to add any components for this example. To the right is a snapshot of the running program.

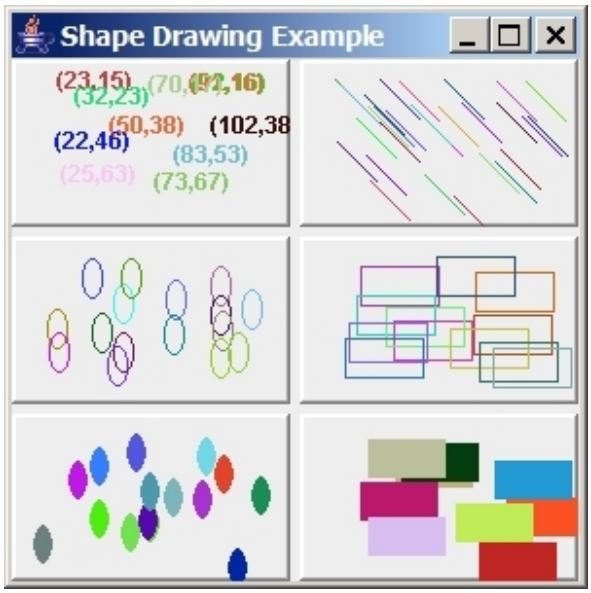
You will notice three things about this example:

1. The text is drawn such that the bottom left corner of the text appears at the location which the mouse is clicked.
2. The text is erased whenever we alter the size of the window.
3. We can ask a mouseEvent for the x and y position of the mouse.

Below is the code.

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

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

         addMouseListener {new MouseAdapter() {
             public void mousePressed(MouseEvent e) {
                 getGraphics().drawString("Hello", e.getX(), e.getY());
             }});

         setDefaultCloseOperation(EXIT_ON_CLOSE);
         setSize(300, 300);
     }
     public static void main(String args[]) {
         new TextDrawingExample("Text Drawing Example").setVisible(true);
     }

}
```

Example:



In this example, we will set up six **JLabel**s, each one allowing a different shape to be drawn onto it. We will set up a single event handler for all mouse presses and within that method we will ask which label has been clicked on and then draw the corresponding shape onto the label. The shapes will be drawn with different colors each time. We use **`Math.random()`** to get a random number for creating a random color. To the right is a snapshot of the working program. You will notice that:

1. The **`getGraphics()`** message is sent to the component, not to the frame.
2. The labels have neat little borders which were

created as **`BorderFactory.createRaisedBevelBorder()`**. You can take a look at the Java API to find out more about the different kinds of borders that are possible.

Here is the code:

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

    private JLabel labels[];

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

        setLayout(new GridLayout(3,2,5,5));
        labels = new JLabel[6];
        for (int i=0; i<6; i++) {
            getContentPane().add(labels[i] = new JLabel());
            labels[i].setBorder(BorderFactory.createRaisedBevelBorder());
        }
        addListeners();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
    }

    // Add listener for a mouse press
    private void addListeners() {
        MouseAdapter  anAdapter = new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                JLabel  area = (JLabel)e.getSource();
                Graphics g = area.getGraphics();

                // Get a random color
                g.setColor(new Color((float)Math.random(),
                                     (float)Math.random(),
                                     (float)Math.random()));

                // Find the label that caused this event
                int   labelNumber;
                for (labelNumber=0; labelNumber<6; labelNumber++) {
                    if (area == labels[labelNumber]) break;
                }
                int   x = e.getX();
                int   y = e.getY();

                // Now decide what to draw
                switch (labelNumber) {
                  case 0: g.drawString("(" + String.valueOf(x) + "," +
                                       String.valueOf(y) + ")", x, y);  break;
                  case 1: g.drawLine(x, y, x+20, y+20);  break;
                  case 2: g.drawOval(x, y, 10, 20);  break;
                  case 3: g.drawRect(x, y, 40, 20);  break;
                  case 4: g.fillOval(x, y, 10, 20);  break;
                  case 5: g.fillRect(x, y, 40, 20);  break;
                }
            }};

        // Add mouse listeners to all labels (for doing something upon mouse
presses)
```

```
        for (int i=0; i<6; i++)
            labels[i].addMouseListener(anAdapter);
    }

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

8.2 Repainting Components

You may have noticed in our examples so far that all the drawings we do are erased when the window is resized. When a window is resized, each of the components needs to be redrawn. Every **JComponent** has (or inherits) a `repaint()` method which is called by JAVA automatically when the window is resized in order to redraw the component. JAVA redraws these components as it already knows how to do, but it will not automatically redraw anything that we may have drawn manually, unless we tell it to. In fact, we too can call this `repaint()` method any time we want our component to be redrawn.

The `repaint()` method actually calls a method called `paintComponent(Graphics g)`, which is also inherited from the **JComponent** class. However, the default inherited `paintComponent()` method does not know what you want to be painted. In order to tell it what to actually redraw, you need to override this method by writing your own `paintComponent()` method which will specify exactly how to draw your graphics.

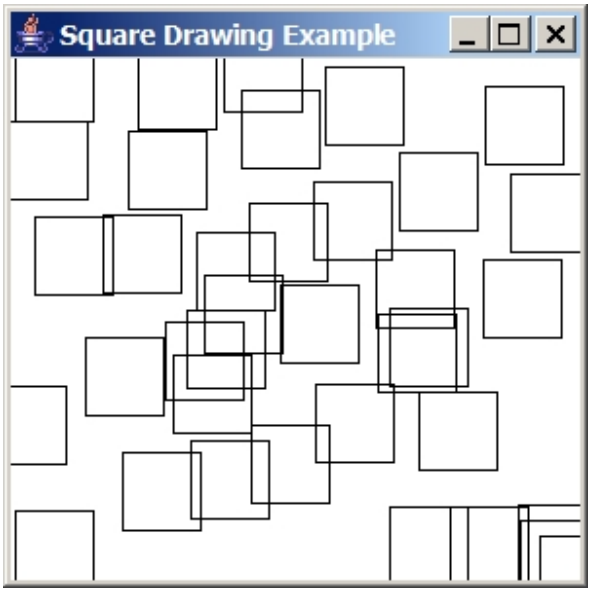
To add this functionality to our previous two examples, we would have to "keep track of" all the graphical shapes that we have been drawing (as well as their attributes, such as location, dimension and colour) so that in our `paintComponent()` method, we can redraw all of them properly each time.

The previous two examples showed how simple graphics can be drawn effortlessly on a frame or on a label. In fact, you can draw on any component. The component that is intended for general purpose drawing is a **JPanel**.

> Note in the older AWT framework of JAVA, a special class called a **Canvas** was used for drawing using a **paint()** method, not the **paintComponent()** method. **JPanel**s in the newer Swing library have all the capabilities of the old Canvas class built-in and should be used instead. In fact if the older **paint()** method is used you can expect bugs, so use the **JPanel**s and **paintComponent()** method instead.

The common strategy in JAVA for drawing on a blank area is to make your own class which is a subclass of **JPanel**. This class should implement, or override, the `paintComponent()` method. When we override this method however, we will be sure to call the **super** method so that the default drawing of the component still occurs.

Example:



In this example, we create a subclass of **JPanel** on which we will keep track of mouse click locations and draw 40x40 pixel squares centered at each of these locations. We will override the **`paintComponent()`** method so that the squares will be properly redrawn whenever we (or JAVA) call **`repaint()`** or when the window is resized. The application itself is not so exciting to look at, but rather the underlying concept of painting on the panel is what is important.

In this example, you may notice a couple of things:

- The **`getPoint()`** method is sent to a **MouseEvent** object to obtain the **`Point`** object representing the location that was clicked.
- Since all squares will be the same size, we don't store the size, just their center locations.

Here is the **SquareCanvas** class that does all the hard work:

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

// This class represents a panel on which 40x40 pixel squares can be drawn.
// The squares are centered around where the user clicks.
public class SquareCanvas extends JPanel implements MouseListener {

    // Keep track of all square center positions
    private ArrayList<Point>  squares;

    // Default constructor
    public SquareCanvas() {
        squares = new ArrayList<Point>();
        setBackground(Color.white);
        addMouseListener(this);
    }

    // This is the method that is responsible for displaying the contents of the
canvas
    public void paintComponent(Graphics graphics) {
        // Draw the component as before (i.e., default look)
        super.paintComponent(graphics);

        // Now draw all of our squares
        graphics.setColor(Color.black);
        for (Point center: squares)
            graphics.drawRect(center.x - 20, center.y - 20, 40, 40);
    }

    // These are unused MouseEventHandlers.  Note that we could have
    // used an Adapter class here.  However, a typical drawing
    // application would make use of these other events as well.
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
```

```
    // Store the mouse location when it is pressed
    public void mousePressed(MouseEvent event) {
        squares.add(event.getPoint());
        repaint(); // this will call paintComponent()
    }

    public static void main(String args[]) {
        JFrame frame = new JFrame("Square Drawing Example");
        frame.add(new SquareCanvas());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

Notice how we are redrawing the panel by first making a call to **super.paintComponent()**. This ensures that the panel's background is redrawn (i.e., erased) before we start drawing again. If we did not do this call, our squares would still be drawn, but the background color for the window (i.e., white in this case) would not be shown. We would end up with the light gray default background coloring of the **JFrame**.

8.3 Displaying Images

We have seen how to draw shapes of different colors onto components, now we will find out how to draw an image on the screen. JAVA lets you load and display both ".gif" files as well as ".jpg" files. We have seen the use of **ImageIcons** with components so that we can display an icon along with text or as a label of a button. Icons, however, are meant to be small images and are not meant for large images. When larger pictures are to be shown, you should use **Image** objects. In fact, the **Image** class is abstract, but there are two useful subclasses. In JAVA, there is much to learn about **Image** objects. There are many classes relating to the manipulation of images and a thorough investigation into these classes is well beyond the scope of this course. Here, we will look simply at the basic displaying of images in our applications.

A typical scenario is to load and display an image (such as a .gif or .jpg) from a file. Unfortunately, the way images are obtained from files is a platform-specific issue. This means that it is not always done the same way, depending on what machine you run your code. Fortunately, JAVA supplies a **Toolkit** class that has common "special" methods for doing various platform-specific things such as loading images.

We can load an image from the disk by asking the **Toolkit** class for an instance of Toolkit (i.e., default will do fine) and then get the image as follows:

```
Image myImage = Toolkit.getDefaultToolkit().createImage("picture.gif");
```

The code loads and returns an **Image** object from the file entitled **picture.gif** but it does not display the image. We can then display the image by asking a **Graphics** object to draw the image:

```
g.drawImage(anImage, x, y, null);
```

The image is drawn with its top-left corner at (x, y) in this graphics context's coordinate space. The 4th parameter can be any class that implements the **ImageObserver** interface. This interface is used as a means of informing a class when an image is done being loaded or drawn (since images in general may take a while to load or draw ... especially if being loaded from a network). This strategy of informing interested classes of image completion, allows more efficient use of process cycles so that the program does not sit idly by doing nothing while the image is being loaded/drawn. We will keep things simple in our example and set this value to **null** so that nobody is informed when the image is loaded or drawn.

One final issue that we are interested in is with respect to the image size. We may want to create a

JPanel that has the exact same size of the image (e.g., for use as a background image for the panel). In this case, we can ask an image for its width and height before choosing the size of our panel. There are **getWidth()** and **getHeight()** methods that we can send to our Image object to obtain these values. However, there is one minor issue. While the image is being loaded (which may take a while), the value returned from **getWidth()** and **getHeight()** is -1. So, we have to introduce a delay in our program by waiting until these methods return valid results:

```
while ((anImage.getWidth(null) == -1) && (anImage.getHeight(null) ==
-1));
```

Notice as well that these methods take an ImageObserver as a parameter (which we set to null). By using a proper **ImageObserver,** we would not have to put in this delay, but could perform other application-specific tasks while we wait for the image to be loaded.

Now we may set the "preferred size" of the panel. Note that setting the "size" of the panel is not useful since when placed on a frame, the frame's layout manager will automatically resize all of its components.

```
   setPreferredSize(new Dimension(anImage.getWidth(null),
anImage.getHeight(null)));
```

So here is the code we can use to test:

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

// This application displays an image
public class ImagePanel extends JPanel {
    private Image anImage;

    public ImagePanel() {
        anImage = Toolkit.getDefaultToolkit().createImage("altree.gif");

        while ((anImage.getWidth(null) == -1) && (anImage.getHeight(null)
== -1));
        setPreferredSize(new Dimension(anImage.getWidth(null),
anImage.getHeight(null)));

    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(anImage, 0, 0, null);
    }

    public static void main(String args[]) {
        JFrame frame = new JFrame("Image Display Test");
        frame.add(new ImagePanel());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();    // Makes size according to panel's preference
        frame.setVisible(true);
    }
}
```

Here is the result:

Note that aince we used the panel's **paintComponent()** method to draw the image, the image is drawn as a background and so any components we add to the panel will appear on top of the image. So you can see that it is quite easy to create a window as shown below simply by adding components to the panel as usual:

There are many more things that you can do with images:

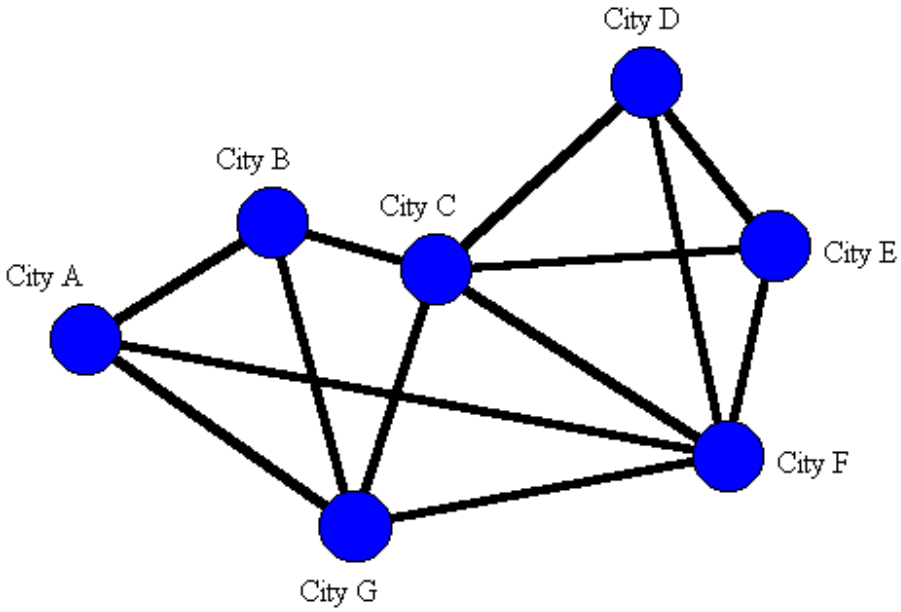
- Shrink/Grow them
- Fade them
- Warp them
- other filters ...

We do not have time to fully investigate these other features of the API library. Feel free to experiment on your own.

8.4 Creating a Simple Graph Editor

This section of the notes describes a step-by-step approach for creating a simple graph editor. It introduces the notion of "drag and drop" as well as selecting objects.

What is a graph ? There are many types of graphs. We are interested in graphs that form topological and/or spatial information. Our graphs will consist of ***nodes*** and ***edges***. The nodes may represent cities in a map while the edges may represent roads between cities:



We would like to make a graph editor with the ability to:

- add/remove nodes
- add/remove edges
- move nodes around (edges between them will remain connected)
- "select" groups of nodes and edges for removal or moving
- do some other useful graph-manipulation features

The Graph Model:

We will begin our application as usual by developing the model. We know that our graph itself is going to be the model, but we must first think about what components make up the graph. These are the nodes and edges. We will first implement some basic **node** and **edge** classes.

Let us create a **Node** class. What ***state*** should each node maintain ? Well, it depends on the application that will be using it. Since we know that the graph will be displayed, each node will need to keep track of its **location**. Also, we may wish to **label** nodes (e.g., a city's name). Here is a basic model for the nodes:

```
import java.awt.Point;
public class Node  {
    private String    label;
    private Point     location;

    public Node() { this("",new Point(0,0)); }
    public Node(String aLabel) { this(aLabel, new Point(0,0)); }
    public Node(Point aPoint) { this("", aPoint); }
    public Node(String aLabel, Point aPoint) {
        label = aLabel;
        location = aPoint;
    }

    public String getLabel() { return label; }
    public Point getLocation() { return location; }
```

```
    public void setLabel(String newLabel) { label = newLabel; }
    public void setLocation(Point aPoint) { location = aPoint; }
    public void setLocation(int x, int y) { location = new Point(x, y); }

    // Nodes look like this: label(12,43)
    public String toString() {
        return(label + "(" + location.x + "," + location.y + ")");
    }
}
```

Notice that we don't have much in terms of behaviour ... simply some get/set methods and a **toString()** method. Notice the two different set methods for location. This gives us flexibility in cases where we the coordinates are either **Point** objects or **ints**.

What state do we need for a graph edge ? Well ... they must *start* at some node and *end* at another so we may want to know which nodes these are. Does it make sense for a graph edge to exist when one or both of its endpoints are not nodes ? Probably not. So an edge should keep track of the node from which it starts and the node at which it ends. We will call them **startNode** and **endNode**. What about a label ? Sure ... roads have names (as well as lengths). Here is a basic **Edge** class:

```
public class Edge  {
    private String   label;
    private Node     startNode, endNode;

    public Edge(Node start, Node end) { this("", start, end); }
    public Edge(String aLabel, Node start, Node end) {
        label = aLabel;
        startNode = start;
        endNode = end;
    }

    public String getLabel() { return label; }
    public Node getStartNode() { return startNode; }
    public Node getEndNode() { return endNode; }

    public void setLabel(String newLabel) { label = newLabel; }
    public void setStartNode(Node aNode) { startNode = aNode; }
    public void setEndNode(Node aNode) { endNode = aNode; }

    // Edges look like this: sNode(12,43) --> eNode(67,34)
    public String toString() {
        return(startNode.toString() + " --> " + endNode.toString());
    }
}
```

Now what about the graph itself ? What do we need for the state of the graph ?
Well ... a graph is just a bunch of nodes and edges.

Still, we have a few choices for representing the Graph:

1. Keep a collection of all nodes AND another collection of all edges
2. Keep only a collection of all nodes
3. Keep only a collection of all edges
4. Keep only 1 node OR 1 edge (this seems weird doesn't it ?)

Let us examine each of these:

1. The 1st strategy would provide quick access for nodes and edges since they are readily available. However, it does take more space than the other strategies.
2. The 2nd strategy allows quick access to nodes, but if we ever needed to get all the edges, we would have to build up the collection, which takes time. This can be done by iterating through all *incident edges* of all nodes and adding the edges (this is slower, but more space efficient). So each node would have to keep track of the edges from/to it.

3. The 3rd strategy is similar to the 2nd except that the edges are efficiently accessible and the nodes are not.
4. The 4th strategy is weird. If we keep one node, we would have to traverse along one of its *incident edges* to the other end and continue in this manner throughout the graph in order to collect all the nodes or edges. However, this will ONLY work if the graph is ***connected*** (i.e., every node can be reached from every other node through a sequence of graph edges).

We will choose the 2nd strategy for our implementation, although you should realize that all three are possible.

Let us examine our **Node** and **Edge** classes a little further and try to imagine additional behaviour that we may want to have.

Notice that each edge keeps track of the nodes that it connects to. But shouldn't a node also keep track of the edges are connected to it ? Think of "real life". Wouldn't it be nice to know which roads lead "into" and "out of" a city ? Obviously, we can always consult the graph itself and check ALL edges to see if they connect to a given city. This is NOT what you would do if you had a map though. You don't find this information out by looking at ALL roads on a map. You find the city of interest, then look at the roads around that area (i.e., only the ones heading into/out of the city).

The point is ... for time efficiency reasons, we will probably want each node to keep track of the edges that it is connected to. Of course, we won't make copies of these edges, we will just keep "pointers" to them so the additional memory usage is not too bad.

We should go back and add the following instance variable to the **Node** class:

```
private ArrayList<Edge>    incidentEdges;
```

We will also need the following get method and another for adding an edge:

```
public ArrayList<Edge> incidentEdges() { return incidentEdges; }
public void addIncidentEdge(Edge e) { incidentEdges.add(e); }
```

We will also have to add this line to the last of the **Node** constructors:

```
incidentEdges = new ArrayList<Edge>();
```

While we are making changes to the **Node** class, we will also add another interesting method called **`neighbours`** that returns the nodes that are connected to the receiver node by a graph edge. That is, it will return an **ArrayList** of all nodes that share an edge with this receiver node. It is very much like asking: "which cities can I reach from this one if I travel on only one highway ?".

We can obtain these neighbours by iterating through the **incidentEdges** of the receiver and extracting the node at the other end of the edge. We will have to determine if this other node is the start or end node of the edge:

```
public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();
    for (Edge e: incidentEdges) {
        if (e.getStartNode() == this)
            result.add(e.getEndNode());
        else
            result.add(e.getStartNode());
    }
    return result;
}
```

As we write this method, it seems that we are writing a portion of code that is *potentially* useful for other situations. That code is the code responsible for finding the opposite node of an edge. We should extract this code and make it a method for the **Edge** class:

```
public Node otherEndFrom(Node aNode) {
    if (startNode == aNode)
        return endNode;
    else
        return startNode;
}
```

Now, we can rewrite the **`neighbours()`** method to use the **`otherEndFrom()`** method:

```
public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();
    for (Edge e: incidentEdges)
        result.add(e.otherEndFrom(this));
    return result;
}
```

Ok. Now we will look at the **Graph** class. We have decided that we were going to store just the nodes, and not the edges. We will also store a label for the graph ... after all ... provinces have names don't they ?

```
import java.util.*;
public class Graph  {
    private String            label;
    private ArrayList<Node>   nodes;

    public Graph() { this("", new ArrayList<Node>()); }
    public Graph(String aLabel) { this(aLabel, new ArrayList<Node>()); }
    public Graph(String aLabel, ArrayList<Node> initialNodes) {
        label = aLabel;
        nodes = initialNodes;
    }
    public ArrayList<Node> getNodes() { return nodes; }
    public String getLabel() { return label; }
    public void setLabel(String newLabel) { label = newLabel; }

    // Graphs look like this: label(6 nodes, 15 edges)
    public String toString() {
        return(label + "(" + nodes.size() + " nodes, " +
               getEdges().size() + " edges)");
    }
}
```

Let us write a method to return all the edges of the graph. It will have to go and collect all the **Edge** objects from the incident edges of the **Node** objects and return them as an **ArrayList**. Can you foresee a small problem ?

```
// Get all the edges of the graph by asking the nodes for them
public ArrayList<Edge> getEdges() {
    ArrayList<Edge>  edges = new ArrayList<Edge>();
    for (Node n: nodes) {
        for (Edge e: n.incidentEdges()) {
            if (!edges.contains(e)) //so that it is not added twice
                edges.add(e);
        }
    }
    return edges;
}
```

Now we need methods for adding/removing nodes/edges. Adding a node or edge is easy, assuming that we already have the node or edge:

```
public void addNode(Node aNode) { nodes.add(aNode); }

public void addEdge(Edge anEdge) {
    // ??????  What ?????? ...
}
```

Wait a minute ! How do we add an edge if we do not store them explicitly ? Perhaps we don't want an **`addEdge`** method that takes an "already created" edge. Instead, we should have an **`addEdge`** method that takes the **startNode** and **endNode** as parameters, then it creates the edge:

```
public void addEdge(Node start, Node end) {
    // First make the edge
    Edge anEdge = new Edge(start, end);

    // Now tell the nodes about the edge
    start.addIncidentEdge(anEdge);
    end.addIncidentEdge(anEdge);
}
```

There ... that is better. What about removing/deleting a node or edge ? Deleting an **Edge** is easy, we just ask the edge's start and end nodes to remove the edge from their lists. Removing a **Node** is a little more involved since all of the incident edges must be removed as well. After all ... we cannot have edges dangling with one of its **Nodes** missing !

```
public void deleteEdge(Edge anEdge) {
    // Just ask the nodes to remove it
    anEdge.getStartNode().incidentEdges().remove(anEdge);
    anEdge.getEndNode().incidentEdges().remove(anEdge);
}

public void deleteNode(Node aNode) {
    // Remove the opposite node's incident edges
    for (Edge e:   aNode.incidentEdges())
        e.otherEndFrom(aNode).incidentEdges().remove(e);
    nodes.remove(aNode); // Remove the node now
}
```

OK. Let us write some code that now tests the model classes. Here is **static** method for the **Graph** class that creates and returns a graph:

```
public static Graph example() {
    Graph    myMap = new Graph("Ontario and Quebec");
    Node     ottawa, toronto, kingston, montreal;

    myMap.addNode(ottawa = new Node("Ottawa", new Point(250,100)));
    myMap.addNode(toronto = new Node("Toronto", new Point(100,170)));
    myMap.addNode(kingston = new Node("Kingston", new Point(180,110)));
    myMap.addNode(montreal = new Node("Montreal", new Point(300,90)));
    myMap.addEdge(ottawa, toronto);
    myMap.addEdge(ottawa, montreal);
    myMap.addEdge(ottawa, kingston);
    myMap.addEdge(kingston, toronto);

    return myMap;
}
```

We can test it by writing **`Graph.example()`** anywhere. This looks fine and peachy, but if we have 100 nodes, we would need 100 local variables (or a big array) just for the purpose of adding edges !! Maybe this would be a better way to write the code:

```
public static Graph example() {
    Graph    myMap = new Graph("Ontario and Quebec");

    myMap.addNode(new Node("Ottawa", new Point(250,100)));
    myMap.addNode(new Node("Toronto", new Point(100,120)));
    myMap.addNode(new Node("Kingston", new Point(200,130)));
    myMap.addNode(new Node("Montreal", new Point(300,70)));
    myMap.addEdge("Ottawa", "Toronto");
    myMap.addEdge("Ottawa", "Montreal");
    myMap.addEdge("Ottawa", "Kingston");
    myMap.addEdge("Kingston", "Toronto");
```

```
        return myMap;
    }
```

This way, we can access the nodes of the graph by their names (assuming that they are all unique names). How can we make this happen ? We just need to make another **`addEdge()`** method that takes two **String** arguments and finds the nodes that have those labels. Perhaps we could make a nice little helper method in the **Graph** class that will find a node with a given name (label):

```
    public Node nodeNamed(String aLabel) {
        for (Node n:  nodes)
            if (n.getLabel().equals(aLabel)) return n;
        return null; // If we don't find one
    }
```

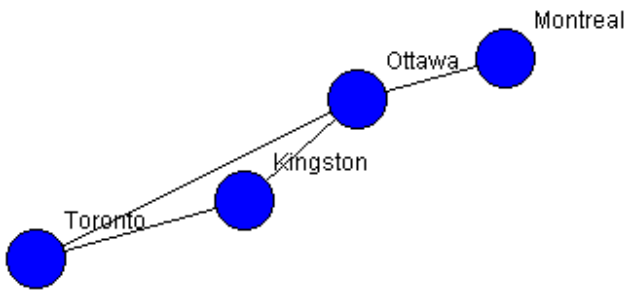
Now we can write another **`addEdge()`** method that takes **String** parameters representing **Node** names:

```
    public void addEdge(String startLabel, String endLabel) {
        Node start = nodeNamed(startLabel);
        Node end = nodeNamed(endLabel);

        if ((start != null) && (end != null))
            addEdge(start, end);
    }
```

Notice the way we share code by making use of the "already existing" **`addEdge()`** method. Also notice the careful checking for valid node labels. After this new addition, the 2nd **`main()`** method that we created above will now work.

Displaying the Graph:

If we are going to be displaying the graph, we need to think about how we want to draw it. Here is what we "may" want to see:



So where do we start ? Let us work on writing code that draws each of the graph components separately. We will start by writing methods for drawing **Nodes** and **Edges,** then use these to draw the **Graph.** We can pass around the **Graphics** object that corresponds to the "pen" that belongs to the panel.

Here is a method for the **Node** class that will instruct a **Node** to draw itself using the given **Graphics** object:

```
    public void draw(Graphics aPen) {
        int    radius = 15;

        // Draw a blue-filled circle around the center of the node
        aPen.setColor(Color.blue);
        aPen.fillOval(location.x - radius, location.y - radius, radius * 2,
    radius * 2);

        // Draw a black border around the circle
        aPen.setColor(Color.black);
```

```
        aPen.drawOval(location.x - radius, location.y - radius, radius * 2,
    radius * 2);

        // Draw a label at the top right corner of the node
        aPen.drawString(label, location.x + radius, location.y - radius);
    }
```

Notice that we draw the node twice ... once for the blue color ... once for the black border.
Here is now a similar method for the **Edge** class that draws an edge:

```
    public void draw(Graphics aPen) {
        // Draw a black line from the center of he startNode to the center of
    the endNode
        aPen.setColor(Color.black);
        aPen.drawLine(startNode.getLocation().x, startNode.getLocation().y,
                      endNode.getLocation().x, endNode.getLocation().y);
    }
```

When drawing the graph, we should draw edges first, then draw the nodes on top. Why not the other way around ? Here is the corresponding draw method for the **Graph** class:

```
    public void draw(Graphics aPen) {
        ArrayList<Edge>    edges = getEdges();

        for (Edge e: edges)  // Draw the edges first
            e.draw(aPen);
        for (Node n: nodes)  // Draw the nodes second
            n.draw(aPen);
    }
```

The User Interface:

Now we can start the creation of our **GraphEditor** user interface. We will begin by making a panel on which we will display the graph:

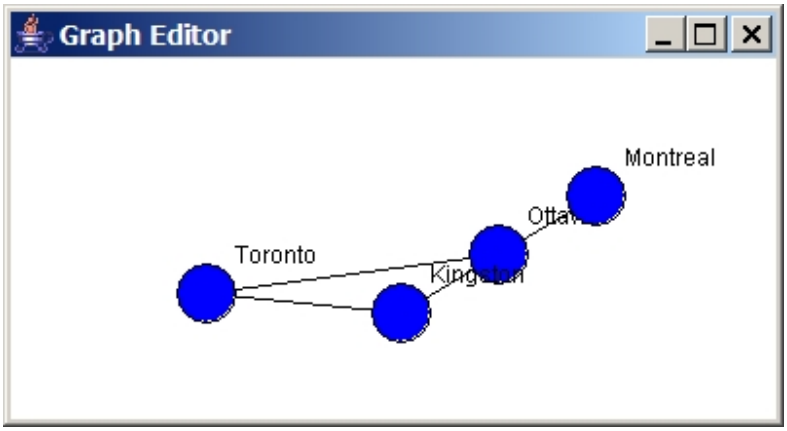
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GraphEditor extends JPanel {
    private Graph  aGraph;  // The model (i.e. the graph)

    public GraphEditor() { this(new Graph()); }
    public GraphEditor(Graph g) {
        aGraph = g;
        setBackground(Color.white);
    }

    // This is the method that is responsible for displaying the graph
    public void paintComponent(Graphics aPen) {
        super.paintComponent(aPen);
        aGraph.draw(aPen);
    }
}
```

Now we will make a class called **GraphEditorFrame** that represents a simple view which holds only our **GraphEditor** panel:

```
import javax.swing.*;
public class GraphEditorFrame extends JFrame {
    private GraphEditor  editor;

    public GraphEditorFrame (String title) { this(title, new Graph()); }
    public GraphEditorFrame (String title, Graph g) {
```

```
        super(title);
        add(editor = new GraphEditor(g));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 400);
    }

    public static void main(String args[]) {
        new GraphEditorFrame("Graph Editor", Graph.example()).setVisible(true);
    }
}
```

Notice that we can run the example by running the **GraphEditorFrame** class. Our example Ontario/Quebec graph comes up right away ! This is because the **`paintComponent()`** method of **`GraphEditor()`** class is called upon startup.



Manipulating Nodes:

What kind of action should the user perform to add a node to the graph ? There are many possibilities (i.e., menu options, buttons, mouse clicks). We will allow nodes to be added to the graph via double clicks of the mouse. When the user double-clicks on the panel, a new node will be added at that click location. We must have the **GraphEditor** class implement the ***MouseListener*** interface. When we receive a click count of 2 on a **mouseClick** event, we will add the node at that location. For now, we will leave the other mouse listeners blank:

```
    public void mouseClicked(MouseEvent event) {
        // If this was a double-click, then add a node at the mouse location
        if (event.getClickCount() == 2) {
            aGraph.addNode(new Node(event.getPoint()));
            // We have changed the model, so now we update
            update();
        }
    }
    public void mousePressed(MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited(MouseEvent event) { }
```

Of course, we will have to add the **MouseListener** in the constructor. We will do this by calling **addEventHandlers()** which we will be adding to later on:

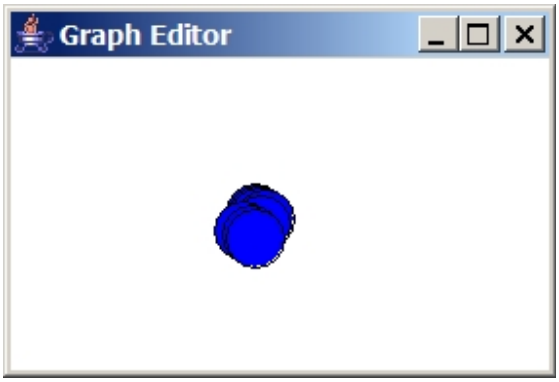
```
    public void addEventHandlers() {
        addMouseListener(this);
    }
    public void removeEventHandlers() {
        removeMouseListener(this);
    }
```

The **update()** method itself is quite simple since there is only one component on the window ! It merely

calls **repaint()** after temporarily disabling the event handlers:

```
public void update() {
    removeEventHandlers();
    repaint();
    addEventHandlers();
}
```

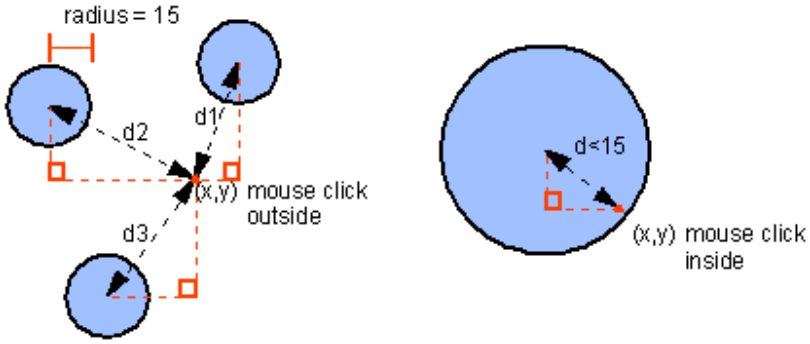
If we run our code, we will notice something that is not so pleasant. Our strategy of using the double click allows us to add nodes on top of each other, making them possibly indistinguishable:



Perhaps instead of having nodes lying on top of each other, we could check to determine whether or not the user clicks within a node. Then we can decide to "not add" the node if there is already one there. What do we do then ... ignore the click ? Maybe we should cause the node to be somehow "selected" so that we can move it around. To do this, we will need to add functionality that allows nodes to be selected and unselected.

If we attempt to re-select an "already selected" node, it should probably become unselected (i.e., toggle on/off). We should make the node appear different as well (perhaps red). We will need to detect which node has been selected. This sounds like it could be a nice little helper method in the **Graph** class.

We can just check the distance from the given point to the center of all nodes. If the distance is <= the radius, then we are inside that node.



In fact, we are not really computing the distance, we are computing the square of the distance. This is more efficient since we do not need to compute the root.

```
// Return the first node in which point p is contained, if none, return
null
public Node nodeAt(Point p) {
    for (Node n:  nodes) {
        Point c = n.getLocation();
        int d = (p.x - c.x) * (p.x - c.x) + (p.y - c.y) * (p.y - c.y);
        if (d <= (15*15)) return n;
    }
    return null;
}
```

The 15 looks like a "magic" number. It seems like this number may be used a lot. We should define a static constant in the **Node** class. Go back and change the draw method as well to use this new static value:

```
public static int    RADIUS = 15;
```

Here is the better code:

```
// Return the first node in which point p is contained, if none, return
null
public Node nodeAt(Point p) {
    for (Node n:  nodes) {
        Point c = n.getLocation();
        d = (p.x - c.x) * (p.x - c.x) + (p.y - c.y) * (p.y - c.y);
        if (d <= (Node.RADIUS * Node.RADIUS)) return n;
    }
    return null;
}
```

We should go back into our drawing routines and adjust the code so that it uses this new RADIUS constant.

Now since we are allowing **Nodes** to be selected, we will have to somehow keep track of all the selected nodes. We have two choices:

- Let the graph keep track of the selected nodes separately
- Let each node keep track of whether or not it is selected

We will choose the second strategy (do you understand the tradeoffs of each ?).
Add the following instance variable and methods to the **Node** class:

```
private boolean    selected;

public boolean isSelected() { return selected; }
public void setSelected(boolean state) { selected = state; }
public void toggleSelected() { selected = !selected; }
```

Now we should modify the draw method to allow nodes to be selected and unselected:

```
public void draw(Graphics aPen) {

    // Draw a blue or red-filled circle around the center of the node
    if (selected)
        aPen.setColor(Color.red);
    else
        aPen.setColor(Color.blue);
    aPen.fillOval(location.x-RADIUS, location.y-RADIUS, RADIUS*2,
RADIUS*2);

    // Draw a black border around the circle
    aPen.setColor(Color.black);
    aPen.drawOval(location.x-RADIUS, location.y-RADIUS, RADIUS*2,
RADIUS*2);

    // Draw a label at the top right corner of the node
    aPen.drawString(label, location.x + radius, location.y - radius);

}
```

Of course, now to make it all work, we must use it in the mouseClicked event handler:

```
public void mouseClicked(MouseEvent event) {
    // If this was a double-click, then add a node at the mouse location
    if (event.getClickCount() == 2) {
```

```
            Node     aNode = aGraph.nodeAt(event.getPoint());
            if (aNode == null)
                aGraph.addNode(new Node(event.getPoint()));
            else
                aNode.toggleSelected();
            // We have changed the model, so now we update
            update();
        }
    }
```

Now how do we allow nodes to be deleted ? Perhaps, the user must select the node(s) first and then hit the **delete** key. Perhaps when the **delete** key is pressed, ALL of the currently selected nodes should be deleted. So we will make a method that first returns all the selected nodes. We will need to add this method to the **Graph** class which returns a vector of all the selected nodes:

```
    // Get all the nodes that are selected
    public ArrayList<Node> selectedNodes() {
        ArrayList<Node>   selected = new ArrayList<Node>();
        for (Node n:  nodes)
            if (n.isSelected()) selected.add(n);
        return selected;
    }
```

We already took care of the node selection, now we must handle the **delete** key. We should have the **GraphEditor** implement the **KeyListener** interface.

```
    public void addEventHandlers() {
        addMouseListener(this);
        addKeyListener(this);
    }
    public void removeEventHandlers() {
        removeMouseListener(this);
        removeKeyListener(this);
    }

    public void keyTyped(KeyEvent event) {}
    public void keyReleased(KeyEvent event) {}

    public void keyPressed(KeyEvent event) {
        if (event.getKeyCode() == KeyEvent.VK_DELETE) {
            for (Node n:  aGraph.selectedNodes())
                    aGraph.deleteNode(n);
            update();
        }
    }
```

There is a SLIGHT problem. It seems that even though we have only one component in our window (i.e., the **JPanel** which is the **GraphEditor** itself), this component does not have the focus by default. In order for the keystrokes to be detectable, the component MUST have the focus. So we will add the following line to the beginning of the **update()** method:

```
    public void update() {
        requestFocus();  // Need this for handling KeyPress
        removeEventHandlers();
        repaint();
        addEventHandlers();
    }
```

Now, how can we move nodes around once they are created ? Once again, we must decide how we want the interface to work. It is most natural to allow the user to move nodes by pressing the mouse down while on top of a node and holding it down while dragging the node to the new location, then release the mouse button to cause the node to appear in the new location. We will need the mousePressed and mouseDragged events of the **MouseListener** and **MouseMotionListener** interfaces, respectively. Here is what we will have to do:

- When the user presses the mouse (i.e., a "press", not a "click"), then determine if he/she pressed on top of a node.
- If yes, then remember this node as being the one selected, otherwise do nothing
- As the mouse moves (while button being held down), we must update the chosen node's location

We will have to remember which node is being dragged so that we can keep changing its location as the mouse is dragged. We will add an instance variable called **dragNode** to keep this node.

```
private Node   dragNode;
```

Here are the mousePressed , mouseDragged and mouseReleased event handlers:

```
// Mouse press event handler
public void mousePressed(MouseEvent event) {
    // First check to see if we are about to drag a node
    Node   aNode = aGraph.nodeAt(event.getPoint());
    if (aNode != null) {
        // If we pressed on a node, store it
        dragNode = aNode;
    }
}

// Mouse drag event handler
public void mouseDragged(MouseEvent event) {
    if (dragNode != null)
        dragNode.setLocation(event.getPoint());

    // We have changed the model, so now update
    update();
}

// Mouse release event handler (i.e. stop dragging process)
public void mouseReleased(MouseEvent event) {
    dragNode = null;
}
```

Notice that the pressing of the mouse merely stores the node to be moved. The releasing of the mouse button merely resets this stored node to **null**. All of the moving occurs in the dragging event handler. If we drag the mouse, we just make sure that we had first clicked on a node by examining the stored node just mentioned. If this stored node is not **null**, we then update its position and then update the rest of the graph.

Notice that all the edges connected to a node move along with the node itself. Can you explain why ?

Manipulating Edges:

We have exhausted almost all the fun out of manipulating the graph nodes and we are now left with the "fun" of adding/deleting/selecting and moving edges. First we will consider adding edges. We must decide again on what action the user needs to perform in order to add the edge:

1. We can have the user double-click on the **startNode**, double click on the **endNode** and then have the edge magically appear.
2. We can select any two nodes of the graph and then perform some magic action (menu item, button press, triple click) to cause an edge to appear between the two selected edges.
3. We can click on a node and then drag the mouse to the destination node while showing the created edge as we go.

I hope you will agree that the 3rd approach is nicer in that it is more intuitive and provides the user with a nice user-friendly interface. We will see that this strategy is called ***elastic banding***.

To start, we will need to make the following assumptions:

- When the user presses and holds the mouse button down on a node, this node becomes the **startNode** for the edge to be created.
- As the user moves the mouse (i.e., **mouseDragged** event) a line should be drawn from this **startNode** to the current mouse position.
- When the user lets go of the mouse button on top of a different node, an edge is created between the two.
- We should abort the process of adding an edge if the user releases the mouse button while: a) not on a node or b) on the same node as he/she started.

We will have to modify the `mousePressed` , `mouseDragged` and `mouseReleased` methods.

As it turns out, the `mousePressed` event handler already stores the "start" node in the **dragNode** variable. But now look at the `mouseDragged` event handler. Currently, if we press the mouse on a node and then drag it, this will end up causing the node to be moved. But we need to allow an elastic band edge to be drawn instead of moving the node. So, we now have two behaviours that we want to do from the same action of pressing the mouse on a node. This presents a conflict since we cannot do both behaviours. Let us modify our node-moving behaviour as follows:

- If the node initially clicked on is a **selected** node, then we will move it, otherwise we will assume that an edge is to be added.

The `mousePressed` event handler currently just stores the selected node. There is really nothing more to do there.

But now during the `mouseDragged` event handler, we will have to make a decision so as to either move the node (if it was a selectedNode) or to merely draw an edge from the pressed node to the current mouse location. We cannot however, do the drawing within this method. Why ? Well, our **`paintComponent()`** method does the drawing and will draw over any of our drawing done here!! The drawing doesn't belong here. Drawing should happen in the **`paintComponent()`** method ONLY. All we will do here is just store the current mouse location in an **elasticEndLocation** variable and use it within the **`paintComponent()`** method. Here are the new changes:

```
// Mouse drag event handler
public void mouseDragged(MouseEvent event) {
    if (dragNode != null) {
        if (dragNode.isSelected())
            dragNode.setLocation(event.getPoint());
        else
            elasticEndLocation = event.getPoint();
    }
    // We have changed the model, so now update
    update();
}
```

Here is the updated **`paintComponent()`** method for the **GraphEditor** class:

```
// This is the method that is responsible for displaying the graph
public void paintComponent(Graphics aPen) {
    super.paintComponent(aPen);
    aGraph.draw(aPen);
    if (dragNode != null)
        if (!dragNode.isSelected())
            aPen.drawLine(dragNode.getLocation().x,
dragNode.getLocation().y,
                          elasticEndLocation.x, elasticEndLocation.y);
}
```

Notice that this method makes use of the **dragNode** and **elasticEndLocation** variables but still needs to decide whether or not to draw the elastic band line. We draw the elastic line ONLY if we are adding an edge. How do we know we are adding an edge ? Well, we must have pressed on a starting node, so

the **dragNode** must not be **null**. Also, that **dragNode** must not be selected, otherwise we are in the middle of a "node moving" operation, not an "edge adding" one.

Our last piece to this trilogy of event handler changes is to have the **mouseReleased** event handler add the new edge ONLY if we let go of the mouse button on top of a node that is not the same as the one we started with. If it is, or we let go somewhere off a node, then we must repaint everything either way to erase the elastic band:

```
// Mouse released event handler (i.e., stop dragging process)
public void mouseReleased(MouseEvent event) {
    // Check to see if we have let go on a node
    Node   aNode = aGraph.nodeAt(event.getPoint());
    if ((aNode != null) && (aNode != dragNode))
        aGraph.addEdge(dragNode, aNode);

    // Refresh the panel either way
    dragNode = null;
    update();
}
```

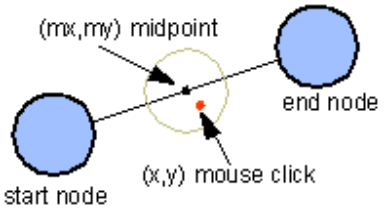
One of our last tasks is to allow edges to be selected and removed. We can similarly add an instance variable and methods to the edge class:

```
private boolean   selected;

public boolean isSelected() { return selected; }
public void setSelected(boolean state) { selected = state; }
public void toggleSelected() { selected = !selected; }
```

Of course ... again we must initialize the instance variable in the constructor. Now we make selected edges appear different (i.e., red).

```
// Draw the edge using the given Graphics object
public void draw(Graphics aPen) {
    // Draw a black or red line from the center of the startNode to the
center of the endNode
    if (selected)
        aPen.setColor(Color.red);
    else
        aPen.setColor(Color.black);
    aPen.drawLine(startNode.getLocation().x, startNode.getLocation().y,
                  endNode.getLocation().x, endNode.getLocation().y);
}
```

To see if an edge has been selected, we will have to decide on where we should click. Perhaps near the midpoint of the edge. But how accurate must we be ? We should allow some tolerance. Maybe a tolerance roughly equivalent to the node's radius is acceptable.



Add the following method to the **Graph** class:

```
// Return the first edge in which point p is near the midpoint; if none,
return null
public Edge edgeAt(Point p) {
    int      midPointX, midPointY;

    for (Edge e:  getEdges()) {
```

```
            mX = (e.getStartNode().getLocation().x +
                  e.getEndNode().getLocation().x) / 2;
            mY = (e.getStartNode().getLocation().y +
                  e.getEndNode().getLocation().y) / 2;
            int distance = (p.x - mX) * (p.x - mX) +
                           (p.y - mY) * (p.y - mY);
            if (distance <= (Node.RADIUS * Node.RADIUS))
               return e;
        }
        return null;
    }
```

Now, upon a double click, we must check for edges. We will first check to see if we clicked on a Node, then if we find that we did not click on a Node, we will check to see if we clicked on an Edge:

```
    public void mouseClicked(MouseEvent event) {
        // If this was a double-click, then add/select a node or select an
    edge
        if (event.getClickCount() == 2) {
            Node aNode = aGraph.nodeAt(event.getPoint());
            if (aNode == null) {
                // We missed a node, now try for an edge midpoint
                Edge anEdge = aGraph.edgeAt(event.getPoint());
                if (anEdge == null)
                    aGraph.addNode(new Node(event.getPoint()));
                else
                    anEdge.toggleSelected();
            }
            else
                aNode.toggleSelected();

            // We have changed the model, so now we update
            update();
        }
    }
```

We can change the keyPressed event handler to delete all selected **Nodes** AND **Edges**. Of course, we will need a method to get the selectedEdges first:

```
    // Get all the edges that are selected
    public ArrayList<Edge> selectedEdges() {
        ArrayList<Edge>   selected = new ArrayList<Edge>();
        for (Edge e:  getEdges())
            if (e.isSelected()) selected.add(e);
        return selected;
    }

    public void keyPressed(KeyEvent event) {
        if (event.getKeyCode() == KeyEvent.VK_DELETE) {
            // First remove the selected edges
            for (Edge e:  aGraph.selectedEdges())
                aGraph.deleteEdge(e);

            // Now remove the selected nodes
            for (Node n:  aGraph.selectedNodes())
                aGraph.deleteNode(n);
            update();
        }
    }
```

8.5 Adding Features to the Graph Editor

We have implemented a basic graph editor. There are many features that can be added. Below are solutions to some added features to the **GraphEditor.** You may want to try to add these features

yourself without looking at the solutions.

Dragging Edges

- Add the following two instance variables to the **GraphEditor** class:

```
private Edge    dragEdge;
private Point   dragPoint;
```

- Add code to the **mousePressed** event handler in the **GraphEditor** class to store the edge to be dragged

```
public void mousePressed(MouseEvent event) {
    // First check to see if we are about to drag a node
    Node   aNode = aGraph.nodeAt(event.getPoint());
    if (aNode != null) {
        // If we pressed on a node, store it
        dragNode = aNode;
    }
    else
        dragEdge = aGraph.edgeAt(event.getPoint());
    dragPoint = event.getPoint();
}
```

- Add code to the **mousePressed** event handler in the **GraphEditor** class to store the edge to be dragged

```
public void mouseDragged(MouseEvent event) {
    if (dragNode != null)  {
        if (dragNode.isSelected())
            dragNode.setLocation(event.getPoint());
        else
            elasticEndLocation = event.getPoint();
    }
    if (dragEdge != null) {
        if (dragEdge.isSelected()) {
            dragEdge.getStartNode().getLocation().translate(
                    event.getPoint().x - dragPoint.x,
                    event.getPoint().y - dragPoint.y);
            dragEdge.getEndNode().getLocation().translate(
                    event.getPoint().x - dragPoint.x,
                    event.getPoint().y - dragPoint.y);
            dragPoint = event.getPoint();
        }
    }

    // We have changed the model, so now update
    update();
}
```

Moving Groups of Vertices/Edges

- Add the following instance variable to the **GraphEditor** class:

```
private Point   dragPoint;
```

- Add the following line at the bottom of the **mousePressed** event handler in the **GraphEditor** class:

```
dragPoint = event.getPoint();
```

- In the **mouseDragged** event handler for the **GraphEditor** class, change

```
if (dragNode != null)  {
    if (dragNode.isSelected())
        dragNode.setLocation(event.getPoint());
    else
        elasticEndLocation = event.getPoint();
}
```

to this:

```
if (dragNode != null)  {
    if (dragNode.isSelected()) {
        for (Node n:  aGraph.selectedNodes()) {
            n.getLocation().translate(
                event.getPoint().x - dragPoint.x,
                event.getPoint().y - dragPoint.y);
        }
        dragPoint = event.getPoint();
    }
    else
        elasticEndLocation = event.getPoint();
}
```

Drawing Selected Edges with Different Thicknesses

- Add the following instance variable to the **Edge** class:

```
public static final int    WIDTH = 7;
```

- Modify the **draw()** method in the **Edge** class:

```
public void draw(Graphics aPen) {
    if (selected) {
        aPen.setColor(Color.RED);
        int xDiff = Math.abs(startNode.getLocation().x -
endNode.getLocation().x);
        int yDiff = Math.abs(startNode.getLocation().y -
endNode.getLocation().y);
        for (int i= -WIDTH/2; i<=WIDTH/2; i++) {
            if (yDiff > xDiff)
                aPen.drawLine(startNode.getLocation().x+i,
startNode.getLocation().y,
                              endNode.getLocation().x+i,
endNode.getLocation().y);
            else
                aPen.drawLine(startNode.getLocation().x,
startNode.getLocation().y+i,
                              endNode.getLocation().x,
endNode.getLocation().y+i);
        }
    }
    else {
        aPen.setColor(Color.black);
        aPen.drawLine(startNode.getLocation().x,
startNode.getLocation().y,
                      endNode.getLocation().x,
endNode.getLocation().y);
    }

}
```

Loading and Saving Graphs

- Add the following methods to the **Node** class:

```
// Save the node to the given file.  Note that the incident edges are
```

```
not saved.
    public void saveTo(PrintWriter aFile) {
        aFile.println(label);
        aFile.println(location.x);
        aFile.println(location.y);
        aFile.println(selected);
    }

    // Load a node from the given file.  Note that the incident edges are
not connected
    public static Node loadFrom(BufferedReader aFile) throws IOException
{
        Node   aNode = new Node();

        aNode.setLabel(aFile.readLine());
        aNode.setLocation(Integer.parseInt(aFile.readLine()),
                          Integer.parseInt(aFile.readLine()));

aNode.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());
        return aNode;
    }
```

- Add the following methods to the **Edge** class:

```
    // Save the edge to the given file.  Note that the nodes
    themselves are not saved.
    // We assume here that node locations are unique identifiers
    for the nodes.
    public void saveTo(PrintWriter aFile) {
        aFile.println(label);
        aFile.println(startNode.getLocation().x);
        aFile.println(startNode.getLocation().y);
        aFile.println(endNode.getLocation().x);
        aFile.println(endNode.getLocation().y);
        aFile.println(selected);
    }

    // Load an edge from the given file.  Note that the nodes
    themselves are not loaded.
    // We are actually making temporary nodes here that do not
    correspond to the actual
    // graph nodes that this edge connects.  We'll have to throw
    out these TEMP nodes later
    // and replace them with the actual graph nodes that connect to
    this edge.
    public static Edge loadFrom(BufferedReader aFile) throws
    IOException {
        Edge    anEdge;
        String  aLabel = aFile.readLine();
        Node    start = new Node("TEMP");
        Node    end = new Node("TEMP");

        start.setLocation(Integer.parseInt(aFile.readLine()),
                          Integer.parseInt(aFile.readLine()));
        end.setLocation(Integer.parseInt(aFile.readLine()),
                          Integer.parseInt(aFile.readLine()));
        anEdge = new Edge(aLabel, start, end);

    anEdge.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());

        return anEdge;
    }
```

- Add the following methods to the **Graph** class:

```
    // Save the graph to the given file.
    public void saveTo(PrintWriter aFile) {
        aFile.println(label);
```

```
        // Output the nodes
        aFile.println(nodes.size());
        for (Node n:  nodes)
            n.saveTo(aFile);

        // Output the edges
        ArrayList<Edge> edges = getEdges();
        aFile.println(edges.size());
        for (Edge e:  edges)
            e.saveTo(aFile);
    }

    // Load a Graph from the given file.  After the nodes and edges are
loaded,
    // We'll have to go through and connect the nodes and edges properly.
    public static Graph loadFrom(BufferedReader aFile) throws IOException
{
        // Read the label from the file and make the graph
        Graph    aGraph = new Graph(aFile.readLine());

        // Get the nodes and edges
        int numNodes = Integer.parseInt(aFile.readLine());
        for (int i=0; i<numNodes; i++)
            aGraph.addNode(Node.loadFrom(aFile));

        // Now connect them with new edges
        int numEdges = Integer.parseInt(aFile.readLine());
        for (int i=0; i<numEdges; i++) {
            Edge tempEdge = Edge.loadFrom(aFile);
            Node start =
aGraph.nodeAt(tempEdge.getStartNode().getLocation());
            Node end =
aGraph.nodeAt(tempEdge.getEndNode().getLocation());
            aGraph.addEdge(start, end);
        }

        return aGraph;
    }
```

- Change the **GraphEditorFrame** class definition to implement the **ActionListener** interface:

```
public class GraphEditorFrame extends JFrame implements
ActionListener
```

- Add the following methods to the **GraphEditor** class:

```
public Graph getGraph() { return aGraph; }
public void setGraph(Graph g) { aGraph = g; update(); }
```

- Add the following to the constructor of the **GraphEditorFrame** class:

```
JMenuBar menubar = new JMenuBar();
setJMenuBar(menubar);
JMenu file = new JMenu("File");
menubar.add(file);
JMenuItem load = new JMenuItem("Load");
JMenuItem save = new JMenuItem("Save");
file.add(load);
file.add(save);
load.addActionListener(this);
save.addActionListener(this);
```

- Add the following event handler to the **GraphEditorFrame** class:

```
public void actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    if (e.getActionCommand().equals("Load")) {
```

```
                int returnVal = chooser.showOpenDialog(this);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    try {
                        editor.setGraph(Graph.loadFrom(new
java.io.BufferedReader(
                                new
java.io.FileReader(chooser.getSelectedFile().getAbsoluteFile()))));
                    }
                    catch (Exception ex) {
                        JOptionPane.showMessageDialog(null,
                                "Error Loading Graph From File !",
                                "Error",
                                JOptionPane.ERROR_MESSAGE);
                    }
                }
            }
            else {
                int returnVal = chooser.showSaveDialog(null);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    try {
                        editor.getGraph().saveTo(new java.io.PrintWriter(
                            new
java.io.FileWriter(chooser.getSelectedFile().getAbsoluteFile())));
                    }
                    catch (java.io.IOException ex) {}
                }
            }
        }
```

Other Features:

There are also other features we can add. Feel free to experiment with the graph editor:

- showing labels on edges
- adjusting labels so that they don't overlap
- scaling (growing or shrinking) of the graph
- repositioning and resizing the graph so that it always fits in the window, even when the window is reduced.

---

9 Networking

What's in This Set of Notes ?

One of the more advanced topics in programming is that of networking and client/server communications. This topic considers multiple applications communicating with one another over a network (such as the internet). We will find out here that JAVA has some nice communication packages that allow us to build programs that communicate with one another in a fairly simply manner.

Here are the individual topics found in this set of notes (click on one to go there):

9.1 Networking Basics

Networking allows you to create multiple JAVA applications and have them communicate with one another. So, we can set up what is known as ***distributed*** applications in which there are client/server relationships. A ***server*** is an application that provides a "service" to various ***clients*** who request the service. There are many client/server scenarios in real life:

- Bank tellers (server) provides a service for the account owners (client)
- Waitresses (server) provides a service for customers (client)
- Travel agents (server) provide a service for people wishing to go on vacation (client)

In some cases, servers themselves may become clients at some times.

- For example, the travel agent will become a client when they phone the airline to make a reservation or contact a hotel to book a room.

In the general networking scenario, everybody can either be a client or a server at any time. This is known as ***peer-to-peer*** computing. In terms of writing java applications it is similar to having many applications communicating among one another.

- For example, Napster worked this way. Thousands of people all act as clients (trying to download songs from another person) as well as servers (in that they allow others to download their songs).

There are many different strategies for allowing communication between applications. JAVA technology allows:

- internet clients to connect to servlets or back-end business systems (or databases).
- applications to connect to one another using sockets.
- applications to connect to one another using RMI.
- some others

We will look at the simplest strategy of connecting applications using sockets.

A ***Protocol*** is:

- a standard pattern of exchanging information.
- like rules/steps for communication

The simplest example of a protocol is a phone conversation:

JIM dials a phone number
MARY says "Hello..."
JIM says "Hello..."
The conversation ensues ...
JIM says "Goodbye"
MARY says "Goodbye"

Perhaps another person gets involved:

JIM dials a phone number
MARY says "Hello..."
JIM says "Hello" and perhaps asks to speak to **FRED**
MARY says "Just a minute"
FRED says "Hello..."
JIM says "Hello..."
The conversation ensues ...
JIM says "Goodbye"
FRED says "Goodbye"

Either way, there is an "expected" set of steps or responses involved during the initiation and conclusion of the conversation. If these steps are not followed, confusion occurs (like when you phone someone and they pick up the phone but do not say anything).

Computer protocols are similar in that a certain amount of "handshaking" goes on to establish a valid connection between two machines. Just as we know that there are different ways to shake hands, there are also different protocols. There are actually layered levels of protocols in that some low level layers deal with how to transfer the data bits, others deal with more higher-level issues such as "where to send the data to".

Computers running on the Internet typically use one of the following high-level **Application Layer** protocols to allow applications to communicate:

- **HyperText Transfer Protocol** (HTTP)
- **File Transfer Protocol** (FTP)
- **Telnet**

This is analogous to having multiple strategies for communicating with someone (in person, by phone, through electronic means, by post office mail etc...).

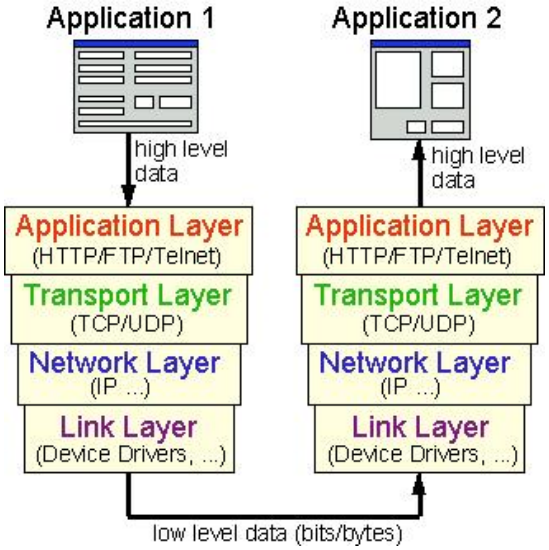
In a lower **Transport Layer** of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

- **Transport Control Protocol** (TCP)
- **User Datagram Protocol** (UDP)

This is analogous to having multiple ways of actually delivering a package to someone (Email, Fax, UPS, Fed-Ex etc...)

Beneath that layer is a **Network Layer** for determining how to locate destinations for the data (i.e., address). And at the lowest level (for computers) there is a **Link Layer** which actually handles the transferring of bits/bytes.

So, internet communication is built of several layers:

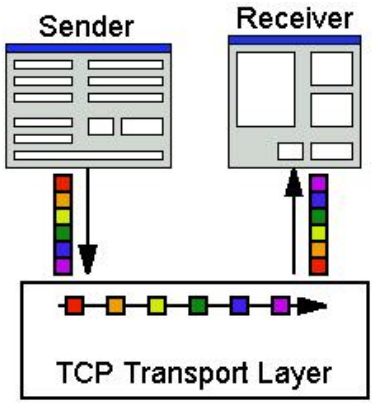


When you write JAVA applications that communicate over a network, you are programming in the Application Layer.

JAVA allows two types of communication via two main types of **Transport Layer** protocols:

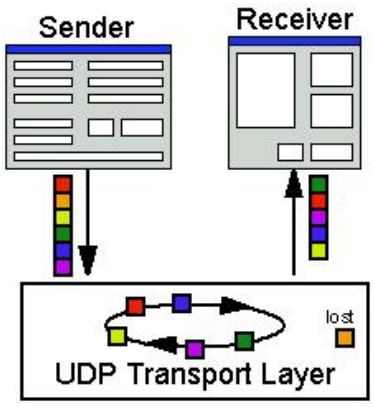
TCP

- a ***connection-based*** protocol that provides a reliable flow of data between two computers.
- guarantees that data sent from one end of the connection actually gets to the other end and in the same order
  - similar to a phone call. Your words come out in the order that you say them.
- provides a point-to-point channel for applications that require **reliable communications**.
- **slow overhead time** of setting up an end-to-end connection.



UDP

- a protocol that sends independent packets of data, called ***datagrams***, from one computer to another.
- no guarantees about arrival. UDP is not connection-based like TCP.
- provides **communication that is not guaranteed** between the two ends
  - sending packets is like sending a letter through the postal service
  - the order of delivery is not important and not guaranteed
  - each message is independent of any other.
- **faster** since no overhead of setting up end-to-end connection
- many firewalls and routers have been configured NOT TO allow UDP packets.



Why would anyone want to use UDP protocol if information may get lost ? Well, why do we use email or the post office ? We are never guaranteed that our mail will make it to the person that we send them to, yet we still rely on them. It may still be quicker than trying to contact a person via phone (i.e., like a TCP protocol).

One more important definition we need to understand is that of a ***port***. Although a computer usually has a single physical connection to the network, data sent by different applications or delivered to them do so through the use of ***ports*** configured on same physical connection. A ***port*** is used as a gateway or "entry point" into an application.

Data transmitted over the internet to an application requires the address of the destination computer and the application's port number. A computer's address is a 32-bit IP address. The port number is a 16-bit number ranging from 0 to 65,535,

with ports 0-1023 restricted by well-known applications like HTTP and FTP.

9.2 URLs

URL is an acronym for **Uniform Resource Locator** and is a reference (an address) to a resource on the Internet. So, it is used to represent the "location" of a webpage or web-based application.

URLs:

- are Strings that describe how to find a resource on the Internet
- represent names of resources which can be files, databases, applications, etc..
- resource names contain a host machine name, filename, port number, and other information.
- may also specify a ***protocol identifier*** (e.g., http, ftp)

Here is an example of a full URL:

http://www.scs.carleton.ca/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs

- **http://** is the protocol identifier which indicates the protocol that will be used to obtain the resource.
- the remaining part is the ***resource name***, and its format depends on the protocol used to access it.

The complete list of components that can be found in a URL resource name are as follows:

Host Name	The name of the machine on which the resource lives: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Port # (optional)	The port number to which to connect: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Filename	The pathname to the file on the machine: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Reference (optional)	A reference to a named *anchor* (a.k.a. *target*) within a resource that usually identifies a specific location within a file: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**

In JAVA, there is a **URL** class defined in the **java.net** package. We can create our own URL objects as follows:

```
URL carleton = new URL("http://www.scs.carleton.ca/~courses/COMP1006/");
```

JAVA will "dissect" the given String in order to obtain information about protocol, hostName, file etc.... Due to this, JAVA may throw a **MalformedURLException** ... so we will need to do this:

```
try {
    URL carleton = new URL("http://www.scs.carleton.ca/~courses/COMP1006/");
} catch(MalformedURLException e) {
    ...
}
```

Another way to create a URL is to break it into its various components:

```
try {
    URL theseNotes = new URL("http", "www.scs.carleton.ca", 80,
                             "/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
} catch(MalformedURLException e) {
    ...
}
```

If you take a look at the JAVA API, you will notice some other constructors as well.

The URL class also supplies methods for extracting the parts (protocol, host, file, port and reference) of a URL object. Here is an example that demonstrates what can be accessed. Note that this example only manipulates a URL object, it does not go off to grab any webpages :) :

```
import java.net.*;
public class URLExample {

     public static void main(String[] args) {

          URL theseNotes = null;
          try {
              theseNotes = new URL("http", "www.scs.carleton.ca", 80,
                 "/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs");
          } catch(MalformedURLException e) {
              e.printStackTrace();
          }
          System.out.println(theseNotes);
          System.out.println("protocol = " + theseNotes.getProtocol());
          System.out.println("host = " + theseNotes.getHost());
          System.out.println("filename = " + theseNotes.getFile());
          System.out.println("port = " + theseNotes.getPort());
          System.out.println("ref = " + theseNotes.getRef());

    }
}
```

Here is the output:

```
http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs
protocol = http
host = www.scs.carleton.ca
filename = /~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html
port = 80
ref = URLs
```

After creating a URL object, you can actually connect to that webpage and read the contents of the URL by using its **openStream()** method which returns an **InputStream**. You actually read from the webpage as if it were a simple text file. If an attempt is made to read from a URL that does not exist, JAVA will throw an **UnknownHostException**

Here is an example that reads a URL directly. It actually reads the file representing this set of notes and displays it line by line to the console. Notice that it reads the file as a text file, so we simply get the HTML code. Also, you must be connected to the internet to run this code:

```
import java.net.*;
import java.io.*;
public class URLReaderExample {
    public static void main(String[] args) {
        URL theseNotes = null;
        try {
            theseNotes = new URL("http", "www.scs.carleton.ca", 80,
                 "/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(theseNotes.openStream()));

            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);

            in.close(); // Close the connection to the net
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " + theseNotes);
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " + theseNotes);
        }
    }
}
```

The output should look something like this, assuming you could connect to the webpage:

```
<!DOCTYPE html PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type"
 content="text/html; charset=iso-8859-1">
  <meta name="Author" content="Mark Lanthier">
  <meta name="GENERATOR"
 content="Mozilla/4.7 [en]C-CCK-MCD EBM-Compaq1  (Win95; U) [Netscape]">
  <title>COMP1006/1406 Notes 9 - Networking</title>
</head>
<body>
 
<table width="100%">
```

```
  <tbody>
    <tr>
      <td><i><font color="#006600">COMP1406/1006 - Design and
Implementation of Computer
Applications</font></i></td>
...
```

Example:

Here is a modification to the above example that reads the URL by making a **URLConnection** first. Since the tasks of opening a connection to a webpage and reading the contents may both generate an IOException, we cannot distinguish the kind of error that occured. By trying to establish the connection first, if any IOExceptions occur, we know they are due to a connection problem. Once the connection has been established, then any further IOException errors would be due to the reading of the webpage data.

```
import java.net.*;
import java.io.*;
public class URLConnectionReaderExample {
    public static void main(String[] args) {
        URL theseNotes = null;
        BufferedReader in = null;
        try {
            theseNotes = new URL("http", "www.scs.carleton.ca", 80,
                 "/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " + theseNotes);
            System.exit(-1);
        }
        try {
            URLConnection  aConnection = theseNotes.openConnection();
            in = new BufferedReader(
                        new InputStreamReader(aConnection.getInputStream()));
        }
        catch (IOException e) {
            System.out.println("Cannot connect to webpage " + theseNotes);
            System.exit(-1);
        }
        try {
            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);
            in.close(); // Close the connection to the net
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " + theseNotes);
        }
    }
}
```

9.3 Client/Server Communications

Many companies today sell services or products. In addition, there are a large number of companies turning towards E-business solutions and various kinds of webserver/database technologies that allow them to conduct business over the internet as well as over other networks.

Such applications usually represent a client/server scenario in which one or more servers serve multiple clients.

Our definition of a ***server*** here will be: *any application that provides a service and allows clients to communicate with it.* Such services may provide:

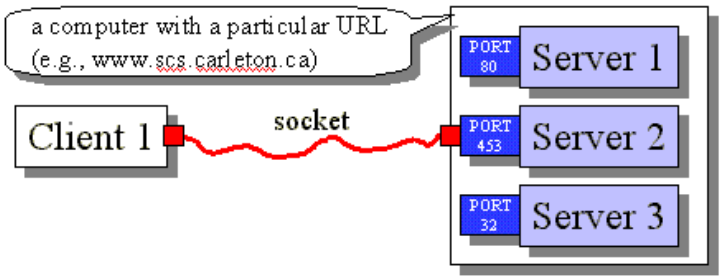
- a recent stock quote
- transactions for bank accounts
- an ability to order products
- an ability to make reservations
- a way to allow multiple clients to interact (Auction)

The ***client***, of course, will be: *any application that requests a service from a server*. The client typically "uses" the service and then displays results to the user. Normally, communication between the client and server must be reliable (no data can be dropped or missing):

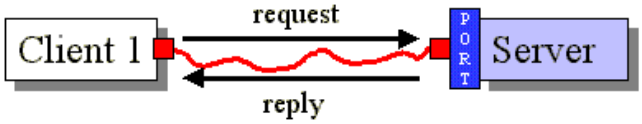
- stock quotes must be accurate and timely
- banking transactions must be accurate and stable
- reservations/orders must be acknowledged

The TCP protocol, mentioned earlier, provides reliable point-to-point communication. Using TCP the client and server must establish a connection in order to communicate. To do this, each program binds a ***socket*** to its end of the connection. A ***socket*** is one endpoint of a two-way communication link between 2 programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application to which the data is to be sent. It is similar to the idea of plugging the two together with a cable.

The ***port number*** is used as the server's location on the machine that the server application is running. So if a computer is running many different server applications on the same physical machine, the port number uniquely identifies the particular server that the client wishes to communicate with:



The client and server may then each read and write to the socket bound to its end of the connection.



In JAVA, the server application uses a **ServerSocket** object to wait for client connection requests. When you create a **ServerSocket**, you must specify a port number (an **int**). It is possible that the server cannot set up a socket and so we have to expect a possible **IOException**. Here is an example:

```
public static int SERVER_PORT = 5000;

ServerSocket serverSocket;
try {
    serverSocket  = new ServerSocket(SERVER_PORT);
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open connection to Server",
                                  "Error", JOptionPane.ERROR_MESSAGE);
}
```

The server can communicate with only one client at a time. The server waits for an incoming client request through the use of the **accept()** message:

```
Socket aClientSocket;
try {
    aClientSocket = serverSocket.accept();
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot accept incoming client connection",
                                  "Error", JOptionPane.ERROR_MESSAGE);
```

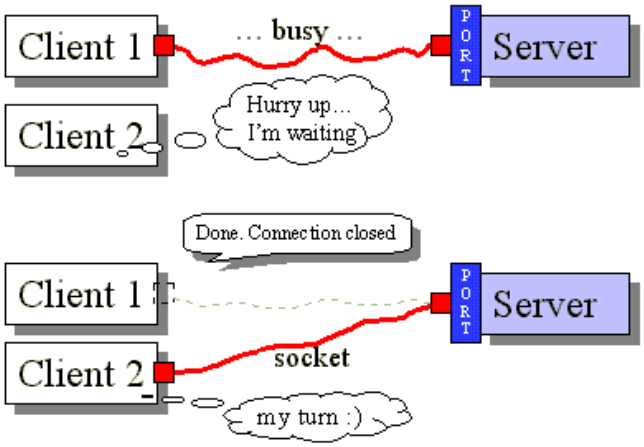
```
}
```

When the **`accept()`** method is called, the server program actually waits (i.e., ***blocks***) until a client becomes available (i.e., an incoming client request arrives. Then it creates and returns a **Socket** object through which communication takes place.

Once the client and server have completed their interaction, the socket is then closed:

```
aClientSocket.close();
```

Only then may the next client open a socket connection to the server. So, remember ... if one client has a connection, everybody else has to wait until they are done:



So how does the client connect to the server ? Well, the client must know the address of the server as well as the PORT number. The server's address is stored as an **InetAddress** object which represents any IP address (i.e., an internet address, an ftp site, local machine etc,...). If the server and client are on the same machine, the static method **getLocatHost()** in the **InetAddress** class may be used to get an address representing the local machine.

```
public static int SERVER_PORT = 5000;

try {
    InetAddress address = InetAddress.getLocalHost();
    Socket socket = new Socket(address, SERVER_PORT);
} catch(UnknownHostException e) {
    JOptionPane.showMessageDialog(null, "Host Unknown",
                                  "Error", JOptionPane.ERROR_MESSAGE);
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot connect to server",
                                  "Error", JOptionPane.ERROR_MESSAGE);
}
```

Once again, a socket object is returned which can then be used for communication.
Here is an example of what a local host may look like:

```
cr850205-a/169.254.180.32
```

The **getLocalHost()** method may, however, generate an **UnknownHostException**.
You can also make an **InetAddress** object by specifying the network IP address directly or the machine name directly as follows:

```
InetAddress.getByName("169.254.1.61");
InetAddress.getByName("www.scs.carleton.ca");
```

So how do we actually do communication between the client and the server ? Well, each socket has an **inputStream** and an **outputStream**. So, once we have the sockets, we simply ask for these streams and then reading and writing may occur.

```
try {
    InputStream   in = socket.getInputStream();
    OutputStream  out = socket.getOutputStream();
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open I/O Streams",
                                  "Error", JOptionPane.ERROR_MESSAGE);
}
```

Normally, however, we actually wrap these input/output streams with text-based, datatype-based or object-based wrappers:

```
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream());

DataInputStream in = new DataInputStream(socket.getInputStream());
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

You may look back at the notes on streams to see how to write to the streams. However, one more point ... when data is sent through the output stream, the **flush()** method should be sent to the output stream so that the data is not buffered, but actually sent right away.

Also, you must be careful when using **ObjectInputStreams** and **ObjectOutputStreams**. When you create an **ObjectInputStream**, it blocks while it tries to read a header from the underlying **SocketInputStream**. When you create the corresponding **ObjectOutputStream** at the far end, it writes the header that the **ObjectInputStream** is waiting for, and both are able to continue. If you try to create both **ObjectInputStreams** first, each end of the connection is waiting for the other to complete before proceeding which results in a deadlock situation (i.e., the programs seems to hang/halt). This behaviour is described in the API documentation for the **ObjectInputStream** and **ObjectOutoutStream** constructors.

9.4 Client/Server Example

Lets now take a look at a real example. In this example, a client will attempt to:

1. connect to a server
2. ask the server for the current time
3. ask the server for the number of requests that the server has handled so far
4. ask the server for an invalid request (i.e., for a pizza)

Here is the client application:

```
import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;
public class Client  {
    private Socket          socket;
    private BufferedReader  in;
    private PrintWriter     out;

    // Make a connection to the server
    private void connectToServer() {
        try {
            socket = new Socket(InetAddress.getLocalHost(), Server.SERVER_PORT);

            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot connect to server",
                                          "Error", JOptionPane.ERROR_MESSAGE);
            System.exit(-1);
        }
    }

    // Disconnect from the server
    private void disconnectFromServer() {
        try {
            socket.close();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot disconnect from server",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    // Ask the server for the current time
    private void askForTime() {
        connectToServer();
        out.println("What Time is It ?");
        out.flush();
        try {
            String time = in.readLine();
```

```
            System.out.println("CLIENT: The time is " + time);
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive time from
server",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
        disconnectFromServer();
    }

    // Ask the server for the number of requests obtained
    private void askForNumberOfRequests() {
        connectToServer();
        out.println("How many requests have you handled ?");
        out.flush();

        int count = 0;
        try {
            count = Integer.parseInt(in.readLine());
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive num requests
from server",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
        System.out.println("CLIENT: The number of requests are " + count);
        disconnectFromServer();
    }

    // Ask the server to order a pizza
    private void askForAPizza() {
        connectToServer();
        out.println("Give me a pizza");
        out.flush();
        disconnectFromServer();
    }

    public static void main (String args[]) {
        Client c = new Client();
        c.askForTime();
        c.askForNumberOfRequests();
        c.askForAPizza();
        c.askForTime();
        c.askForNumberOfRequests();
    }
}
```

Now the server application runs forever, continually waiting for incoming client requests:

```
import java.net.*;   // all socket stuff is in here
import java.io.*;
import javax.swing.JOptionPane;
public class Server {
    public static int SERVER_PORT = 6000;  // arbitrary, but above 1023
    private int counter = 0;

    // Helper method to get the ServerSocket started
    private ServerSocket goOnline() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(SERVER_PORT);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error creating network
connection",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
        System.out.println("SERVER online");
        return serverSocket;
     }

     // Handle all requests
     private void handleRequests(ServerSocket serverSocket) {
        while(true) {
            Socket socket = null;
            BufferedReader in = null;
            PrintWriter out = null;
            try {
                // Wait for an incoming client request
                socket = serverSocket.accept();

                // At this point, a client connection has been made
                in = new BufferedReader(
                            new InputStreamReader(socket.getInputStream()));
```

```
                out = new PrintWriter(socket.getOutputStream());
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "SERVER: Error connecting to
    client",
                                              "Error", JOptionPane.ERROR_MESSAGE);
                System.exit(-1);
            }
            // Read in the client's request
            try {
                String request = in.readLine();
                System.out.println("SERVER: Client Message Received: " + request);
                if (request.equals("What Time is It ?")) {
                    out.println(new java.util.Date());
                    counter++;
                }
                else if (request.equals("How many requests have you handled ?"))
                    out.println(counter++);
                else
                    System.out.println("SERVER: Unknown request: " + request);

                // Now make sure that the response is sent
                out.flush();

                // We are done with the client's request
                socket.close();
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "SERVER: Error communicating with
    client",
                                              "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    public static void main (String args[]) {
        Server s = new Server();
        ServerSocket ss = s.goOnline();
        if (s != null) s.handleRequests(ss);
    }
}
```

Note, to run this using JCreator, we will have to execute two different JCreator applications, one for the server and one for the client.

9.5 Datagram Sockets

Recall that with datagrams there is no direct socket connection between the client and the server. That is, packets are received "in seemingly random order" from different clients. It is similar to the way email works. If the client requests or server responses are too big, they are broken up into multiple packets and sent one packet at a time. The server is not guaranteed to receive the packets all at once, nor in the same order, nor is it guaranteed to receive all the packets !!

Let us look at the same client-server application, but by now using **DatagramSockets** and **DatagramPackets**. Once again, the server will be in a infinite loop accepting messages, although there will be no direct socket connection to the client. We will be setting up a ***buffer*** (i.e., an array of bytes) which will be used to receive incoming requests. Each message is sent as a ***packet***. Each packet contains:

- the ***data*** of message (i.e., the message itself)
- the ***length*** of the message (i.e., the number of bytes)
- the ***address*** of the sender (as an InetAddress)
- the ***port*** of the sender

The code for packaging and sending an outgoing packet involves creating a **DatagramSocket** and then constructing a **DatagramPacket**. The packet requires an array of bytes, as well as the address and port in which to send to. The byte array can be obtained from most objects by sending a **getBytes()** message to the object. Finally, a **send()** message is used to send the packet:

```
    DatagramSocket  socket = new DatagramSocket();
    byte[]          sendBuffer = "This is the data (which does not have to be a
String)".getBytes();
    DatagramPacket  packetToSend = new DatagramPacket(sendBuffer, sendBuffer.length,
anInetAddress, aPort);
    socket.send(packetToSend);
```

The server code for receiving an incoming packet involves allocating space (i.e., a byte array) for the **DatagramPacket** and then receiving it. The code looks as follows:

```
byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
DatagramPacket receivePacket = new DatagramPacket(recieveBuffer, recieveBuffer.length);
socket.receive(receivePacket);
```

We then need to extract the data from the packet. We can get the address and port of the sender as well as the data itself from the packet as follows:

```
InetAddress  sendersAddress = receivePacket.getAddress();
int          sendersPort = receivePacket.getPort();
String       sendersData = new String(receivePacket.getData(), 0, receivePacket.getLength());
```

In this case the data sent was a String, although it may in general be any object.

By using the sender's address and port, whoever receives the packet can send back a reply.

Here is the modified client/server code using the DatagramPackets:

```
import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;
public class PacketServer {

     public static int SERVER_PORT = 6000;
     private static int INPUT_BUFFER_LIMIT = 500;

     private int counter = 0;

     // Handle all requests
     private void handleRequests() {
         System.out.println("SERVER online");

         // Create a socket for communication
         DatagramSocket socket = null;
         try {
             socket = new DatagramSocket(SERVER_PORT);
         } catch (SocketException e) {
             JOptionPane.showMessageDialog(null, "SERVER: Cannot connect to network",
                                           "Error", JOptionPane.ERROR_MESSAGE);
             System.exit(-1);
         }

         // Now handle incoming requests
         while(true) {
             try {
                // Wait for an incoming client request
                byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
                 DatagramPacket receivePacket = new DatagramPacket(
                                   recieveBuffer, recieveBuffer.length);
                 socket.receive(receivePacket);

                // Extract the packet data that contains the request
                 InetAddress address = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();
                 String request = new String(receivePacket.getData(), 0,
                                             receivePacket.getLength());
                 System.out.println("SERVER: Packet received: \"" + request +
                                   "\" from " + address + ":" + clientPort);

                // Decide what should be sent back to the client
                byte[] sendBuffer;
                if (request.equals("What Time is It ?")) {
                     System.out.println("SERVER: sending packet with time info");
                     sendResponse(socket, address, clientPort,
                                  new java.util.Date().toString().getBytes());
                     counter++;
                 }
                else if (request.equals("How many requests have you handled ?")) {
                     System.out.println("SERVER: sending packet with num requests");
                     sendResponse(socket, address, clientPort,
                                  ("" + ++counter).getBytes());
                 }
                else
                     System.out.println("SERVER: Unknown request: " + request);
             } catch(IOException e) {
```

```
                JOptionPane.showMessageDialog(null, "SERVER: Error receiving client
    requests",
                                              "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    // This helper method sends a given response back to the client
    private void sendResponse(DatagramSocket socket, InetAddress address,
                              int clientPort, byte[] response) {
        try {
            // Now create a packet to contain the response and send it
            DatagramPacket sendPacket = new DatagramPacket(response,
                                response.length, address, clientPort);
            socket.send(sendPacket);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error sending response to client" +
                                          address + ":" + clientPort,
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    public static void main (String args[]) {
        new PacketServer().handleRequests();
    }

}
```

Notice that only one **DatagramSocket** is used, but that a new **DatagramPacket** object is created for each incoming message.

Now lets look at the client:

```
    import java.net.*;
    import java.io.*;
    import javax.swing.JOptionPane;
    public class PacketClient  {

        private static int INPUT_BUFFER_LIMIT = 500;
        private InetAddress localHost;

        public PacketClient() {
            try {
                localHost = InetAddress.getLocalHost();
            } catch(UnknownHostException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Error connecting to
        network",
                                              "Error", JOptionPane.ERROR_MESSAGE);
                System.exit(-1);
            }
        }

        // Ask the server for the current time
        private void askForTime() {
            DatagramSocket socket = null;
            try {
                socket = new DatagramSocket();
                byte[] sendBuffer = "What Time is It ?".getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                            sendBuffer.length, localHost,
        Server.SERVER_PORT);
                System.out.println("CLIENT: Sending time request to server");
                socket.send(sendPacket);
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Error sending time
        request to server",
                                              "Error", JOptionPane.ERROR_MESSAGE);
            }

            try {
                byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
                DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
        receiveBuffer.length);
                socket.receive(receivePacket);
                System.out.println("CLIENT: The time is " +
                                   new String(receivePacket.getData(), 0,
        receivePacket.getLength()));
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive time from
        server",
```

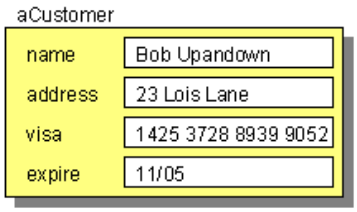
```
                                        "Error", JOptionPane.ERROR_MESSAGE);
            }
            socket.close();
        }

        // Ask the server for the number of requests obtained
        private void askForNumberOfRequests() {
            DatagramSocket socket = null;
            try {
                socket = new DatagramSocket();
                byte[] sendBuffer = "How many requests have you handled ?".getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                    sendBuffer.length, localHost, Server.SERVER_PORT);
                System.out.println("CLIENT: Sending request count request to server");
                socket.send(sendPacket);
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Error sending request to
        server",
                                        "Error", JOptionPane.ERROR_MESSAGE);
            }

            try {
                byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
                DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
        receiveBuffer.length);
                socket.receive(receivePacket);
                System.out.println("CLIENT: The number of requests are " +
                                new String(receivePacket.getData(), 0,
        receivePacket.getLength()));
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive num
        requests from server",
                                        "Error", JOptionPane.ERROR_MESSAGE);
            }
            socket.close();
        }

        // Ask the server to order a pizza
        private void askForAPizza() {
            try {
                byte[] sendBuffer = "Give me a pizza".getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                sendBuffer.length, localHost, Server.SERVER_PORT);
                DatagramSocket socket = new DatagramSocket();
                System.out.println("CLIENT: Sending pizza request to server");
                socket.send(sendPacket);
                socket.close();
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "CLIENT: Error sending request to
        server",
                                        "Error", JOptionPane.ERROR_MESSAGE);
            }
        }

        public static void main (String args[]) {
            PacketClient c = new PacketClient();
            c.askForTime();
            c.askForNumberOfRequests();
            c.askForAPizza();
            c.askForTime();
            c.askForNumberOfRequests();
        }

    }
```

9.6 Auction Example

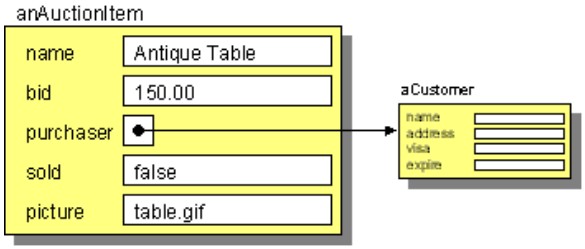
Let us look at a large example now in which a server application represents an auction. Items are put up for auction and clients bid over the network on the items. Clients must first register with the auction server and then they may make bids on the items as they are placed up for auction. The user at the server end, decides when the item is no longer up for auctioning.

The example involves 11 classes:

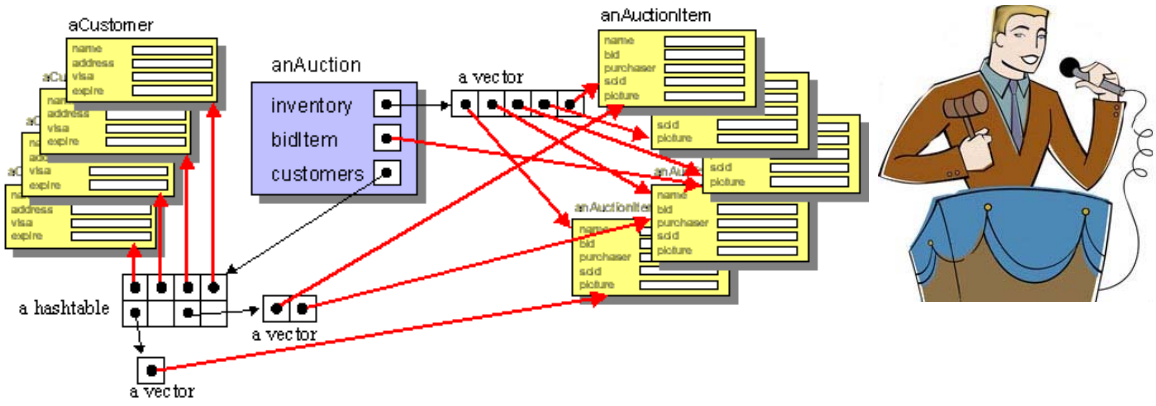
- Model classes:
  - **Customer** - a customer who will be bidding at the auction:



- **AuctionItem** - an item which is up for Auctioning:

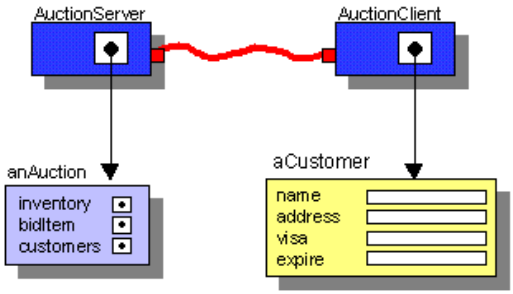


- **Auction** - the auction itself, keeping track of inventory of items to be auctioned, the current item up for auction and the customers (and their purchases) and the auctioning business logic:



Communication classes:

- **AuctionServer** - allows communication between the auction and the outside world.
- **AuctionClient** - allows communication (i.e., bidding/responses) between the client and the server.



User Interface classes:

- **AuctionServerApp** - the server GUI (only 1 server runs this).
- **AuctionClientApp** - the client GUI (each client runs this).
- **RegistrationDialog** - the dialog box used for getting registration information.
- **AuctionCatalogDialog** - the dialog box used to display the auction's catalog of items.
- **AuctionItemDialog** - the dialog box that allows a new AuctionItem to be specified (at the server).
- **DialogClientInterface** - the interface that allows dialog boxes to inform their owners upon closing.

Let us consider the basic model classes. First, we will examine the **Customer** ... it is quite simple:

```
public class Customer implements java.io.Serializable {
    private String   name;
```

```
    private String   address;
    private String   visa;
    private String   expire;

    public Customer() { this("","","",""); }
    public Customer(String n, String a, String v, String e) {
        name = n; address = a; visa = v; expire = e;
    }
    public String getName() { return name; }
    public String getAddress() { return address; }
    public String getVisa() { return visa; }
    public String getExpire() { return expire; }

    public void setName(String n) { name = n; }
    public void setAddress(String a) { address = a; }
    public void setVisa(String v) { visa = v; }
    public void setExpire(String e) { expire = e; }

    public boolean hasMissingInformation() {
        return ((name == null) || (name.length()==0) ||
                (visa == null) || (visa.length()==0) ||
                (address == null) || (address.length()==0) ||
                (expire == null) || (expire.length()==0));
    }
}
```

The class is straight forward. The **hasMissingInformation()** method will be useful later when we ask the user for the customer information. It checks to make sure that there are non-null, non-zero-length strings for all the data. Of course, we are not validating the visa number anywhere.

The **AuctionItem** class is similarily simple:

```
public class AuctionItem implements java.io.Serializable {
    private String     name;
    private float      bid;
    private Customer   purchaser;
    private boolean    sold;
    private String     picture;

    public AuctionItem() { this("", 0, ""); }
    public AuctionItem(String n, float startingBid, String fileName) {
        name = n;
        bid = startingBid;
        purchaser = null;
        picture = fileName;
        sold = false;
    }
    public String getName() { return name; }
    public float getBid() { return bid; }
    public Customer getPurchaser() { return purchaser; }
    public String getPicture() { return picture; }
    public boolean isSold() { return sold; }

    public void setName(String n) { name = n; }
    public void setBid(float amount) { bid = amount; }
    public void setPicture(String imageName) { picture = imageName; }
    public void setPurchaser(Customer c) { purchaser = c; }
    public void setSold() { sold = true; }

    public String toString() {
        if (sold) return "[SOLD " + name + " ]";
        else return name;
    }
}
```

Notice that the picture is actually just a filename. We use the following images for our auction items:

Now what about the **Auction** itself ? It is more complicated since it must deal with all auction-type behaviour such as placing items up for bid, handling/validating bids, registering clients, handling purchases etc... So what kind of methods do we need to write ? Consider first the Auction's state, and think about how it changes hen the users interact with it.

The auction will maintain an inventory of **AuctionItem** objects:

```
private ArrayList<AuctionItem>   inventory;
```

We will need to know which item is currently up for bid (initially **null**):

```
private AuctionItem   bidItem;
```

Finally, we will want to keep track of the **Customers** and their purchases. We can use a **HashMap** where the keys are the **Customer** objects themselves and the corresponding value will be an **ArrayList** of all **AuctionItems** purchased so far by that **Customer**:

```
private HashMap<Customer,ArrayList<AuctionItem>>  customers;
```

Customers should interact this way with the auction:

- Register with the auction - note that we keep an **ArrayList** of purchases for each **Customer**.

```
public void registerCustomer(Customer c) {
    customers.put(c, new ArrayList<AuctionItem>());
}
```

- Ask for a catalog (i.e., a list) of items that will be bidded on

```
public ArrayList<AuctionItem> getInventory() { return inventory; }
```

- Ask if an item is currently up for bidding

```
public boolean hasItemUpForBid() { return bidItem != null; }
```

- Ask what item is currently up for bidding

```
public AuctionItem getBidItem() { return bidItem; }
```

- Ask what the latest bid is - return 0 if no item is up for bid

```
public float latestBid() {
    if (bidItem == null) return 0;
    return bidItem.getBid();
}
```

- Make a bid for the latest item

```
public boolean acceptBidFrom(String name, float amount) {
    // If nothing is up for bidding, don't accept the bid
    if (bidItem == null) return false;

    // First make sure the bid is actually valid
    if (amount <= latestBid()) return false;

    // Now make sure the customer is valid
    Customer c = customerWithName(name);
    if (c != null) {
        // Store the bid amount AND the bidder
        bidItem.setPurchaser(c);
        bidItem.setBid(amount);
        return true;
    }
    return false;
}
```

- Ask who made the latest bid (as confirmation when the customer makes a bid)

```
public Customer latestBidder() {
    if (bidItem == null) return null;
    return bidItem.getPurchaser();
}
```

What about the person running the auction ? What kind of behaviour does he/she need ?

- Add to the inventory of items to be auctioned

```
public void add(AuctionItem item) {
    inventory.add(item);
}
```

- Get a list of all customers and their purchases

```
public HashMap<Customer, ArrayList<AuctionItem>> getCustomers() { return customers; }
```

- Place an item up for bidding - make sure it was not already sold :)

```
public void placeUpForBid(AuctionItem item) {
    if (item.isSold()) return;
    bidItem = item;
}
```

- Stop the bidding process for an item - call when the item is considered sold

```
public void stopBidding() {
    if (latestBidder() != null) {
        ArrayList<AuctionItem> purchases = customers.get(latestBidder());
        purchases.add(bidItem);
        bidItem.setSold();
        inventory.remove(bidItem);
    }
    bidItem = null;
}
```

So, you can see that the methods are all quite simple. They are all public so that the **AuctionServer** can communicate with this model class when either customer requests come in, or when the server user interacts with his/her GUI.

Here is the combined code:

```
import java.util.ArrayList;
import java.util.HashMap;
public class  Auction {

    private ArrayList<AuctionItem>                    inventory;
    private AuctionItem                               bidItem;
    private HashMap<Customer,ArrayList<AuctionItem>>  customers;

    public Auction() { this(new ArrayList<AuctionItem>()); }
    public Auction(ArrayList<AuctionItem> initInventory) {
        inventory = initInventory;
        bidItem = null;
        customers = new HashMap<Customer,ArrayList<AuctionItem>>();
    }
    public ArrayList<AuctionItem> getInventory() { return inventory; }
    public AuctionItem getBidItem() { return bidItem; }
    public HashMap<Customer,ArrayList<AuctionItem>> getCustomers() { return
    customers; }

    // Return the latest bid
    public float latestBid() {
        if (bidItem == null) return 0;
        return bidItem.getBid();
    }

    // Return the latest bidder
    public Customer latestBidder() {
        if (bidItem == null) return null;
        return bidItem.getPurchaser();
    }

    // Return the name of the latest bidder
    public String latestBidderName() {
        if ((bidItem == null) || (bidItem.getPurchaser() == null))
            return "";
        return bidItem.getPurchaser().getName();
    }

    // Add the given item to the inventory
    public void add(AuctionItem item) {
        inventory.add(item);
    }

    // Register the Customer with the given information to Auction
    public void registerCustomer(String name, String address,
        String visa, String expire) {
        registerCustomer(new Customer(name, address, visa, expire));
```

```
    }

    // Register the given Customer with the Auction
    public void registerCustomer(Customer c) {
        customers.put(c, new ArrayList<AuctionItem>());
    }

    // Place the given item up for bidding, customers can now bid on it
    public void placeUpForBid(AuctionItem item) {
        if (item.isSold()) return;
        bidItem = item;
    }

    // Return whether or not there is currently an item up for bidding
    public boolean hasItemUpForBid() { return bidItem != null; }

    // Find the Customer with the given name
    public Customer customerWithName(String name) {
        for (Customer c: customers.keySet())
          if (c.getName().equals(name)) return c;
        return null;
    }

    // Accept an incoming bid for the item up for bid.
    // If it is a valid bid, remember who made the bid
    // and increase the latest bid amount.
    public boolean acceptBidFrom(String name, float amount) {
        // If nothing is up for bidding, don't accept the bid
        if (bidItem == null) return false;

        // First make sure the bid is actually valid
        if (amount <= latestBid()) return false;

        // Now make sure the customer is valid
        Customer c = customerWithName(name);
        if (c != null) {
            bidItem.setPurchaser(c);
            bidItem.setBid(amount);
            return true;
        }
        return false;
    }

    // Once the bidding has stopped, the item is considered
    // to be sold to the last bidder, if there was one.
    public void stopBidding() {
        if (latestBidder() != null) {
            ArrayList<AuctionItem> purchases = customers.get(latestBidder());
            purchases.add(bidItem);
            bidItem.setSold();
            inventory.remove(bidItem);
        }
        bidItem = null;
    }

    public static Auction example1() {
        Auction      a = new Auction();
        AuctionItem  first = new AuctionItem("Antique Table",150.0f,"table.jpg");
        a.add(first);
        a.add(new AuctionItem("JVC VCR",65.0f,"vcr.jpg"));
        a.add(new AuctionItem("Antique Cabinet",400.0f,"cabinet.jpg"));
        a.add(new AuctionItem("5-piece Drumset",190.0f,"drumset.jpg"));
        a.add(new AuctionItem("Violin & Case",100.0f,"violin.jpg"));
        a.add(new AuctionItem("13\" TV/VCR Combo",100.0f,"tvvcr.jpg"));
        a.add(new AuctionItem("486Dx2-66 Laptop",125.0f, "486laptop.jpg"));
        a.add(new AuctionItem("Rocking Chair",80.0f, "rockingchair.jpg"));
        a.add(new AuctionItem("1996 Mazda Miata",6500.0f,"miata.jpg"));
        a.placeItemUpForBid(first);
        return a;
    }

}
```

Now, what about the server itself ? Well, we have seen earlier how to make a simple server that can **accept()** incoming messages from a client via a **ServerSocket** object. We use the same approach. We will simply get the server started, and then dispatch any incoming messages to an appropriate helper method.

What kinds of client messages should the server accept ?

- 'r': register a client
- 'b': handle an incoming bid;
- 'c': handle a catalog request
- 'u': handle an update request (i.e., latest bid info)

So, the server should wait in an infinite loop, accepting client messages forever. Here is the basic framework, we will add the helper methods later:

```
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;
public class  AuctionServer  extends Thread {

     // These variables are required for communication with clients
     public static int SERVER_PORT = 6000;
     private ServerSocket       serverSocket;
     private ObjectInputStream  inputStreamFromClient;
     private ObjectOutputStream outputStreamToClient;
     private boolean            online;

     // This is the model on which we are auctioning
     private Auction     auction;

     // Keep the appl. too so we can update it when we change info
     private AuctionServerApp serverApplication;

     public Auction getAuction() { return auction; }
     public AuctionServer(Auction a) {
         auction = a;  online = false;
     }

     // Allow a server application to register for updates to the model
     public void registerForUpdates(AuctionServerApp app) {
         serverApplication = app;
     }

     // Attempt to bring the server online
     public boolean goOnline() {
         online = false;
         try {
             serverSocket = new ServerSocket(SERVER_PORT);
             online = true;
             System.out.println("SERVER Auction Server Online");
             start(); // Starts the server by calling run()
         } catch(IOException e) {
             JOptionPane.showMessageDialog(null, "SERVER: Error Getting Server
     Online",
                                           "Error", JOptionPane.ERROR_MESSAGE);
             System.exit(-1);
         }
         return online;
     }

     // Do what is necessary to shut down the server connection
     public boolean goOffline() {
         try {
             if (online){
                 serverSocket.close();
                 online = false;
                 System.out.println("SERVER Auction Server Offline");
             }
         } catch(IOException e) {
             JOptionPane.showMessageDialog(null, "SERVER: Error Going Offline",
                                           "Error", JOptionPane.ERROR_MESSAGE);
         }
         return online;
     }

     // Try disconnecting from the client
     private boolean closeClientConnection(Socket s) {
         try {
             if (s != null) s.close();
             return true;
         } catch(IOException e) {
             JOptionPane.showMessageDialog(null, "SERVER: Error During Client
     Disconnect",
                                           "Error", JOptionPane.ERROR_MESSAGE);
             return false;
         }
     }
```

```
        // Accept incoming messages from clients forever
        public void run() {
            // Accept messages forever
            while (online) {
                Socket socket = null;
                try {
                   // Wait for an incoming message
                    socket = serverSocket.accept();
                } catch(IOException e) {
                    JOptionPane.showMessageDialog(null, "SERVER: Error Contacting
        Client",
                                                  "Error", JOptionPane.ERROR_MESSAGE);
                }
                try {
                    // Make object streams for the socket
                    inputStreamFromClient =
                        new ObjectInputStream(socket.getInputStream());
                    outputStreamToClient =
                        new ObjectOutputStream(socket.getOutputStream());

                    // Now handle the message
                    try {
                        String aLine; // Will hold initial command message
                        aLine = (String)inputStreamFromClient.readObject();
                        System.out.println("SERVER Received: " + aLine);

                      if (aLine != null) {
                          // Dispatch to the helper methods
                          char command = aLine.charAt(0);
                          switch(command) {
                              case 'r': registerClient(); break;
                              case 'b': handleIncomingBid(); break;
                              case 'c': handleCatalogRequest(); break;
                              case 'u': handleUpdateRequest(); break;
                              default: System.out.println("SERVER Error:
                                            Invalid Message " + command);
                          }
                      }
                      else
                            System.out.println("SERVER Error: Invalid
                                            Client Message Command");
                    } catch(ClassNotFoundException e) {
                        System.out.println("SERVER Error in Client
                                            Message Data");
                    }
                } catch(IOException e) {
                    JOptionPane.showMessageDialog(null, "SERVER: Error Receiving
        Client Message",
                                                  "Error", JOptionPane.ERROR_MESSAGE);
                } finally {
                    // Now close the connection to this client
                    System.out.println("SERVER Closing Client Connection");
                    closeClientConnection(socket);
                }
            }
        }
        // Test a simple AuctionServer
        public static void main(String args[]) {
            AuctionServer server = new AuctionServer(Auction.example1());
            server.goOnline();
        }

    }
```

Notice that the code looks huge ... but its actually mostly error checking. For example, here is the run() method without the clutter of the error checking and comments (although the code below won't compile):

```
public void run() {
    while (online) {
        Socket socket = serverSocket.accept();
        inputStreamFromClient = new ObjectInputStream(socket.getInputStream());
        outputStreamToClient = new ObjectOutputStream(socket.getOutputStream());

        String aLine = (String)inputStreamFromClient.readObject();
        if (aLine != null) {
            char command = aLine.charAt(0);
            switch(command) {
              case 'r': registerClient(); break;
              case 'b': handleIncomingBid(); break;
              case 'c': handleCatalogRequest(); break;
              case 'u': handleUpdateRequest(); break;
```

```
                default: System.out.println("SERVER Error: Invalid Message " +
                                            command);
            }
        }
        closeClientConnection(socket);
    }
}
```

So what about handling the different messages ? Below are the helper methods.

When a registration message is received, we need to read in a **Customer** object (the client will have to make a **Customer** object with his/her name, address, visa & expiry date. We will take this object, make sure that no information is missing and then send a reply back to the client customer. If the registration is successful, we will send back `"Registration received"` otherwise we will send back `"Registration Error: information is missing"`.

```
    // Handle an incoming request for a client to be registered
    private void registerClient() {
        Customer c = null;
        try {
            c = (Customer)inputStreamFromClient.readObject();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Receiving Client Registration
    Information",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        } catch(ClassNotFoundException e) {
            System.out.println("SERVER Error In Client Registration Data");
        }
        try {
            if (c != null) {
                if (c.hasMissingInformation())
                    outputStreamToClient.writeObject("Registration Error: information is
    missing");
                else {
                    auction.registerCustomer(c);
                    outputStreamToClient.writeObject("Registration received");
                }
            }
            outputStreamToClient.flush();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Sending Registration
    Response",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Now what about when the customer wants to make a bid ? In this case, we will need to know the name of the **Customer** as well as the amount being bid. We will need to verify that this customer is indeed registered and also that his/her bid is valid. If he/she is not registered, we will send back: `"Error: You MUST register first"`. If the bid is invalid, we will send back: `"Error: Your bid is invalid"`. Otherwise we will record the bid and send back: `"Bid received"`.

```
    // Handle an incoming request for a client to make a bid
    private void handleIncomingBid() {
        String name = "";
        float bid = 0;
        try {
            // Expect the client's name and bid
            name = (String)inputStreamFromClient.readObject();
            bid = ((Float)inputStreamFromClient.readObject()).floatValue();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Receiving Client Bid
    Information",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        } catch(ClassNotFoundException e) {
            System.out.println("SERVER Error In Client's Bid Data");
        }
        try {
            if (auction.customerWithName(name) == null)
                outputStreamToClient.writeObject("Error: You MUST register first");
            else if (auction.acceptBidFrom(name, bid))
                outputStreamToClient.writeObject("Bid received");
            else
                outputStreamToClient.writeObject("Error: Your bid is invalid");
            outputStreamToClient.flush();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Sending Bid Response",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
        // Update the server application to reflect the new bid information
        if (serverApplication != null)
```

```
            serverApplication.update();
    }
```

Now what about requests for a client to have a catalog ? In this case, we simply send back the inventory **ArrayList**:

```
    // Handle an incoming request for a client to have a catalog
    private void handleCatalogRequest() {
        try {
            outputStreamToClient.writeObject(auction.getInventory());
            outputStreamToClient.flush();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Sending Catalog to Client",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

That one was easy :). Our last message that we need to handle is an **update()**. The client application will repeatedly ask for the latest bid information so that it's screen can be refreshed. Each time the server receives this message, it should simply send back the **AuctionItem** that is currently up for bid. Note that this item contains all necessary update information including the name of the item, its latest bid value and its latest bidder. Note that if nothing is currently up for bid when this message is received, we simply send back **null**.

```
    // Handle an incoming request for the latest bid information
    private void handleUpdateRequest() {
        try {
            outputStreamToClient.writeObject(auction.getBidItem());
            outputStreamToClient.flush();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Sending Update to Client",
                                          "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Well, that is it for the **Server**. Now what about the **Client** ? Here is the framework

```
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;
import java.util.ArrayList;
public class  AuctionClient  {

     // These variables hold the necessary socket information
     // for communincating with the Auction Server
     private Socket                socket;
     private ObjectInputStream     inputStreamFromServer;
     private ObjectOutputStream    outputStreamToServer;
     private Object                serverReply;

     // This is the Customer who is attached to this client as its model
     private Customer     customer;
     private AuctionItem  latestAuctionItem;

     public AuctionClient(Customer c) { customer = c; }

     // Return the server's latest reply, useful for GUI status pane
     public Object getServerReply() { return serverReply; }

     // Return the Customer information pertaining to this client
     public Customer getCustomer() { return customer; }

     // Return the AuctionItem being bid on
     public AuctionItem getLatestAuctionItem() {
         return latestAuctionItem;
     }

     // Try connecting to the auction server
     private boolean establishServerConnection() {
         try {
             socket = new
     Socket(InetAddress.getLocalHost(),AuctionServer.SERVER_PORT);
             outputStreamToServer = new
     ObjectOutputStream(socket.getOutputStream());
             inputStreamFromServer = new ObjectInputStream(socket.getInputStream());
             return true;
         } catch(IOException e) {
             handleError("CLIENT: Error Connecting to Server");
             return false;
         }
```

```
        }

        // Try disconnecting from the auction server
        private boolean closeServerConnection() {
            try {
                socket.close();
                return true;
            } catch(IOException e) {
                handleError("CLIENT: Error Disconnecting from Server");
                return false;
            }
        }

        // Use this to display errors and also to record them for the GUI
        private void handleError(String s) {
            serverReply = s;
            JOptionPane.showMessageDialog(null, s, "Error",
        JOptionPane.ERROR_MESSAGE);
        }

        // Test by registering, sending bid and requesting catalog
        public static void main(String args[]) {
            AuctionServer server = new AuctionServer(Auction.example1());
            server.goOnline();
            AuctionClient client = new AuctionClient(
                   new Customer("Mark", "23 Oak St.", "4256 4878 0439 2387", "12/04"));
            System.out.println(client.register());
            System.out.println(client.getServerReply());
            System.out.println(client.sendBid("Mark", 120.23f));
            System.out.println(client.getServerReply());
            System.out.println(client.sendBid("Bob", 300));
            System.out.println(client.getServerReply());
            System.out.println(client.sendBid("Mark", 5000));
            System.out.println(client.getServerReply());
            System.out.println(client.sendForCatalog());
            System.out.println("\"" + client.sendForUpdate() + "\"");
            server.goOffline();
        }

    }
```

Notice in the main method that the interesting methods are missing. Let us create those now.

To register, the client must connect to the server, send a `"register request"` message and then send the **Customer** information for this client. We will wait for the server's reply and store it in the serverReply variable (which will be shown in the status pane of the GUI). Then, we close the server connection.

```
    // Send a request to be registered for the auction by the server
    public boolean register() {
        if (!establishServerConnection()) return false;

        // Output the appropriate registration information
        try {
            outputStreamToServer.writeObject("register request");
            outputStreamToServer.writeObject(customer);
            outputStreamToServer.flush();
        } catch(IOException e) {
            handleError("CLIENT: Error sending Registration");
        }

        // Now wait to see if the registration went through
        boolean result = false;
        try {
            serverReply = (String)inputStreamFromServer.readObject();
            if (serverReply != null)
                result = serverReply.equals("Registration received");
        } catch(IOException e) {
            handleError("CLIENT: Error receiving registration response");
        } catch(ClassNotFoundException e) {
            handleError("CLIENT: Error in Server reply data");
        } finally {
            closeServerConnection();
            return result;
        }
    }
```

Now to bid on an item, we need to send a `"bid request"` message as well as the **Customer** name and the bid value. We then need to wait for the reply to see if it worked and simply store this reply in the serverReply for the GUI to display.

```
    // Send a bid to the server
```

```
public boolean sendBid(String name, float bid) {
    if (!establishServerConnection()) return false;

    // Output the appropriate bid information
    try {
        outputStreamToServer.writeObject("bid request");
        outputStreamToServer.writeObject(name);
        outputStreamToServer.writeObject(new Float(bid));
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending bid");
    }

    // Now wait to see if the bid went through
    boolean result = false;
    try {
        serverReply = (String)inputStreamFromServer.readObject();
        if (serverReply != null)
            result = serverReply.equals("Bid received");
    } catch(IOException e) {
        handleError("CLIENT: Error receiving bid response");
    } catch(ClassNotFoundException e) {
        handleError("CLIENT: Error in Server reply data");
    } finally {
        closeServerConnection();
        return result;
    }
 }
```

When a catalog is required, we simply send a simple `"catalog request"` message to the server and get back the result (stored in serverReply). The method will also return the catalog **ArrayList** ... but we will let the GUI figure out what to do with it :).

```
// Send a request for a catalog of auction items to the server
public ArrayList<AuctionItem> sendForCatalog() {
    if (!establishServerConnection())
   return new ArrayList<AuctionItem>();

    // Send the request
    try {
        outputStreamToServer.writeObject("catalog request");
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending catalog request");
    }

    // Now wait to see if the request went through
    serverReply = new ArrayList<AuctionItem>();
    try {
        serverReply = (ArrayList<AuctionItem>)inputStreamFromServer.readObject();
        if (serverReply == null)
            serverReply = new ArrayList<AuctionItem>();
    } catch(IOException e) {
        handleError("CLIENT: Error receiving catalog");
    } catch(ClassNotFoundException e) {
        handleError("CLIENT: Error in Server reply data");
    } finally {
        closeServerConnection();
        return (ArrayList<AuctionItem>)serverReply;
    }
}
```

OK ... Now for the update request. We will send `"update request"` to the server and then wait for the latest **AuctionItem** to be returned. We will return it from this method.

```
// Send a request for the latest bid information to the server
public AuctionItem sendForUpdate() {
    if (!establishServerConnection()) return null;

    // Send the request
    try {
        outputStreamToServer.writeObject("update request");
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending update request");
        return null;
    }
    // Now wait to see if the request went through
    serverReply = null;
    latestAuctionItem = null;
```

```
        try {
            serverReply = (AuctionItem)inputStreamFromServer.readObject();
            latestAuctionItem = (AuctionItem)serverReply;
        } catch(IOException e) {
            handleError("CLIENT: Error receiving update");
        } catch(ClassNotFoundException e) {
            handleError("CLIENT: Error in Server reply data");
        } finally {
            closeServerConnection();
            return latestAuctionItem;
        }
    }
```

Wow! That sure was fun. Actually, we can run the main method shown above since it runs the server and a client as well. It is a good idea to run these to make sure that everything works fine. Here is our test case:

```
AuctionServer server = new AuctionServer(Auction.example1());
server.goOnline();
AuctionClient client = new AuctionClient(
       new Customer("Mark", "23 Oak St.", "4256 4878 0439 2387", "12/04"));
System.out.println(client.register());
System.out.println(client.getServerReply());
System.out.println(client.sendBid("Mark", 120.23f));
System.out.println(client.getServerReply());
System.out.println(client.sendBid("Bob", 300));
System.out.println(client.getServerReply());
System.out.println(client.sendBid("Mark", 5000));
System.out.println(client.getServerReply());
System.out.println(client.sendForCatalog());
System.out.println("\"" + client.sendForUpdate() + "\"");
server.goOffline();
```

Here is the output which server output in blue, client output in red:

```
SERVER Auction Server Online
SERVER Received a Command: register request
SERVER Closing Client Connection
true
Registration received
SERVER Received a Command: bid request
SERVER Closing Client Connection
false
Error: Your bid is invalid
SERVER Received a Command: bid request
SERVER Closing Client Connection
false
Error: You MUST register first
SERVER Received a Command: bid request
SERVER Closing Client Connection
true
Bid received
SERVER Received a Command: catalog request
SERVER Closing Client Connection
[Antique Table, JVC VCR, Antique Cabinet, 5-piece Drumset, Violin & Case, 13" TV/VCR
Combo, 486Dx2-66 Laptop, Rocking Chair, 1996 Mazda Miata]
SERVER Received a Command: update request
SERVER Closing Client Connection
"Antique Table"
SERVER Auction Server Offline
```

Notice that there are some errors when closing because the sockets are still set up when the client quits :).

Now what about the GUI's ? Let us examine the dialog box that requests for a new item to be added to the auction inventory. It should allow the user to specify the item name, starting bid and picture file. Here is what it will look like, and its code is shown below:


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

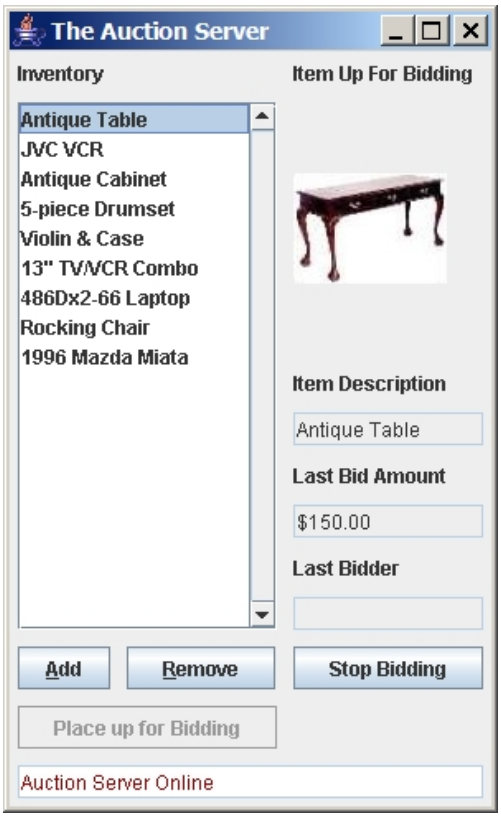
```
import javax.swing.*;
public class  AuctionItemDialog  extends JDialog {

     // Store a pointer to the model for changes later
     private AuctionItem item;

     private JTextField nameTextField;
     private JTextField bidTextField;
     private JTextField pictureField;

     // The buttons and main panel
     private JButton okButton;
     private JButton cancelButton;

     public AuctionItemDialog(Frame owner, AuctionItem ai){

         super(owner,"New Auction Item", true);

         item = ai;  // Store the model

         // Make a panel with two buttons (placed side by side)
         JPanel buttonPanel = new JPanel();
         buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
         buttonPanel.add(okButton = new JButton("OK"));
         buttonPanel.add(cancelButton = new JButton("CANCEL"));

         // Make a panel with auction item information
         JPanel itemPanel  = new JPanel();

         nameTextField = new JTextField(item.getName(), 15);
         bidTextField = new JTextField(""+item.getBid(), 15);
         bidTextField.setHorizontalAlignment(JTextField.RIGHT);
         pictureField = new JTextField(item.getPicture(), 15);

         // Set the layoutManager and add the components
         itemPanel.setLayout(new GridLayout(3,2,5,5));
         itemPanel.add(new JLabel("Item Name:"));
         itemPanel.add(nameTextField);
         itemPanel.add(new JLabel("Starting Bid ($):"));
         itemPanel.add(bidTextField);
         itemPanel.add(new JLabel("Picture file (gif/jpg):"));
         itemPanel.add(pictureField);

         // Make the dialog box by adding the two panels
         setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
         add(itemPanel);
         add(buttonPanel);

         // Prevent the window from being resized
         setSize(365, 150);
         setResizable(false);

         // Listen for ok button click
         okButton.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent event){
                 okButtonClicked();
         }});

         // Listen for cancel button click
         cancelButton.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent event){
                 cancelButtonClicked();
         }});

         // Listen for window closing: treat like cancel button
         addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent event) {
                 cancelButtonClicked();
         }});
     }

     public AuctionItem getAuctionItem() { return item; }

     private void okButtonClicked(){
         // Update model to show changed owner name
         item.setName(nameTextField.getText());
         item.setBid(Float.parseFloat(bidTextField.getText()));
         item.setPicture(pictureField.getText());

         // Tell the client that ok was clicked
```

```
            if (getOwner() != null)
                ((DialogClientInterface)getOwner()).dialogFinished(this);
            dispose();
        }

        private void cancelButtonClicked(){
            // Tell the client that cancel was clicked
            if (getOwner() != null)
                ((DialogClientInterface)getOwner()).dialogCancelled(this);
            dispose();
        }

    }
```

Notice how the item variable (i.e., the model) is updated when the OK is clicked. Other than that, there is nothing new here.

OK, now let us look at the Server's GUI code. The **AuctionServerApp** class represents the application that is used by the person who is running the auction. It should be connected to an **Auction** object and have the ability to list/add/remove **AuctionItems** as well as place one up for bidding. In addition, the user will want to be able to stop the bidding when no clients have responded to the latest bid for a while. The image to the right is what the GUI will look like.

Below is the basic framework for the code. To keep things simple, the update methods and the event handlers are discussed afterwards.




```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.*;
import java.net.*;

public class  AuctionServerApp extends JFrame implements DialogClientInterface {

    // This image gets shown when there is nothing up for bid
    private static ImageIcon BLANK_IMAGE =  new ImageIcon("blankItem.jpg");

    private JTextField   statusField;
    private JTextField   bidItemField;
    private JTextField   bidAmountField;
    private JTextField   bidderField;
    private JList        itemsList;
    private JLabel       pictureLabel;
    private JButton      bidButton, stopButton;
    private JButton      addButton, removeButton;

    // This interface connects to an AuctionServer that handles all the work
    private AuctionServer     auctionServer;
    private int               selectedItemIndex;
    private AuctionItem       newAuctionItem; // item being added to auction

    private ListSelectionListener        itemsListListener;
```

```
public AuctionServerApp() { this(new AuctionServer(new Auction())); }
public AuctionServerApp(AuctionServer a) {
    super("The Auction Server");
    auctionServer = a;
    auctionServer.registerForUpdates(this);
    if (auctionServer.getAuction().getInventory().size() > 0)
        selectedItemIndex = 0;
    else
        selectedItemIndex = -1;
    initializeComponents();
    addListeners();
    update();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(300,490);
    setVisible(true);
}

private void addListeners() {
    // When the window is first OPENED, go online
    addWindowListener(new WindowAdapter() {
        public void windowOpened(WindowEvent event) {
           goOnline(); }});

    // When the window is CLOSED, go offline
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent event) {
           goOffline(); }});

    // Add a listener for the ADD button
    addButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
           handleAddAuctionItem(); }});

    // Add a listener for the REMOVE button
    removeButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
           handleRemoveAuctionItem(); }});

    // Add a listener for the PLACE UP FOR BIDDING button
    bidButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
           handlePlaceForBid(); }});

    // Add a listener for the STOP BIDDING button
    stopButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
           handleStopBid(); }});

    // Add a selection listener for the inventory list
    itemsList.addListSelectionListener(
    itemsListListener = new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent theEvent) {
           handleSelectAuctionItem(); }});
}

// Cause the Auction Server to go online
private void goOnline() {
    if (auctionServer.goOnline()) {
        updateStatus("Auction Server Online");
        auctionServer.start(); // start the thread
    }
    else
        updateStatus("Error: Problem Getting Auction Server Online");
}

// Cause the Auction Server to go offline
private void goOffline() {
    if (auctionServer.goOffline())
        updateStatus("Auction Server Offline");
    else
        updateStatus("Error: Problem Going Offline");
}

// Build the frame by adding all the components
private void initializeComponents() {
    GridBagLayout layout = new GridBagLayout();
    GridBagConstraints layoutConstraints = new GridBagConstraints();

    JPanel panel = new JPanel();
    panel.setLayout(layout);
    setContentPane(panel);
```

```
JLabel aLabel = new JLabel("Inventory");
layoutConstraints.gridx = 0;
layoutConstraints.gridy = 0;
layoutConstraints.gridwidth = 2;
layoutConstraints.gridheight = 1;
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.insets = new Insets(5,5,5,5);
layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
layoutConstraints.weightx = 0.0;
layoutConstraints.weighty = 0.0;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

aLabel = new JLabel("Item Up For Bidding");
layoutConstraints.gridx = 2;
layoutConstraints.gridwidth = 1;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

pictureLabel = new JLabel();
layoutConstraints.gridy = 1;
layoutConstraints.fill = GridBagConstraints.BOTH;
layoutConstraints.weightx = 1.0;
layoutConstraints.weighty = 1.0;
layout.setConstraints(pictureLabel, layoutConstraints);
panel.add(pictureLabel);

aLabel = new JLabel("Item Description");
layoutConstraints.gridy = 2;
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.weightx = 0.0;
layoutConstraints.weighty = 0.0;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

aLabel = new JLabel("Last Bid Amount");
layoutConstraints.gridy = 4;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

aLabel = new JLabel("Last Bidder");
layoutConstraints.gridy = 6;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

bidItemField = new JTextField();
bidItemField.setEditable(false);
layoutConstraints.gridx = 2;
layoutConstraints.gridy = 3;
layoutConstraints.gridwidth = 1;
layoutConstraints.gridheight = 1;
layoutConstraints.fill = GridBagConstraints.HORIZONTAL;
layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
layoutConstraints.weightx = 1.0;
layoutConstraints.weighty = 0.0;
layout.setConstraints(bidItemField, layoutConstraints);
panel.add(bidItemField);

bidAmountField = new JTextField();
bidAmountField.setEditable(false);
layoutConstraints.gridy = 5;
layout.setConstraints(bidAmountField, layoutConstraints);
panel.add(bidAmountField);

bidderField = new JTextField();
bidderField.setEditable(false);
layoutConstraints.gridx = 2;
layoutConstraints.gridy = 7;
layout.setConstraints(bidderField, layoutConstraints);
panel.add(bidderField);

addButton = new JButton("Add");
addButton.setMnemonic('A');
layoutConstraints.gridx = 0;
layoutConstraints.gridy = 8;
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.weightx = 0.0;
layout.setConstraints(addButton, layoutConstraints);
panel.add(addButton);

itemsList = new JList();
JScrollPane scrollPane = new JScrollPane(itemsList,
```

```
                ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
            layoutConstraints.gridx = 0;
            layoutConstraints.gridy = 1;
            layoutConstraints.gridwidth = 2;
            layoutConstraints.gridheight = 7;
            layoutConstraints.fill = GridBagConstraints.BOTH;
            layoutConstraints.anchor = GridBagConstraints.CENTER;
            layoutConstraints.weighty = 1.0;
            layout.setConstraints(scrollPane, layoutConstraints);
            panel.add(scrollPane);

            removeButton = new JButton("Remove");
            removeButton.setMnemonic('R');
            layoutConstraints.gridx = 1;
            layoutConstraints.gridy = 8;
            layoutConstraints.gridwidth = 1;
            layoutConstraints.gridheight = 1;
            layoutConstraints.fill = GridBagConstraints.NONE;
            layoutConstraints.ipadx = 10;
            layoutConstraints.ipady = 0;
            layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
            layoutConstraints.weighty = 0.0;
            layout.setConstraints(removeButton, layoutConstraints);
            panel.add(removeButton);

            bidButton = new JButton("Place up for Bidding");
            layoutConstraints.gridx = 0;
            layoutConstraints.gridy = 9;
            layoutConstraints.gridwidth = 2;
            layout.setConstraints(bidButton, layoutConstraints);
            panel.add(bidButton);

            stopButton = new JButton("Stop Bidding");
            layoutConstraints.gridx = 2;
            layoutConstraints.gridy = 8;
            layoutConstraints.gridwidth = 1;
            layout.setConstraints(stopButton, layoutConstraints);
            panel.add(stopButton);

            statusField = new JTextField();
            statusField.setEditable(false);
            statusField.setBackground(new Color(255,255,255));
            statusField.setForeground(new Color(160,0,0));
            layoutConstraints.gridx = 0;
            layoutConstraints.gridy = 10;
            layoutConstraints.gridwidth = 4;
            layoutConstraints.fill = GridBagConstraints.HORIZONTAL;
            layoutConstraints.weightx = 1.0;
            layout.setConstraints(statusField, layoutConstraints);
            panel.add(statusField);
        }

        // Update all the components
        public void update() {
            itemsList.removeListSelectionListener(itemsListListener);
            updateItemsList();
            updateRemoveButton();
            updatePlaceButton();
            updateStopButton();
            updateBidItemField();
            updateBidAmountField();
            updateBidderField();
            updatePictureLabel();
            itemsList.addListSelectionListener(itemsListListener);
        }

        // Test it by bringing up the server and some clients
        public static void main(String[] args) {
            new AuctionServerApp(new AuctionServer(Auction.example1()));
            new AuctionClientApp();
            new AuctionClientApp();
            new AuctionClientApp();
        }

    }
```

What about writing all the update methods ?
The inventory list is updated simply by obtaining the inventory **ArrayList** from the auction model:

```
    private void updateItemsList() {
        itemsList.setListData(new Vector(auctionServer.getAuction().getInventory()));
```

```
        itemsList.setSelectedIndex(selectedItemIndex);
    }
```

The REMOVE button is disabled when nothing is selected from the list. The PLACE UP FOR BIDDING button and the STOP BIDDING buttons are disabled when the auction does not have anything up for bidding.

```
    private void  updateRemoveButton() {
        removeButton.setEnabled(selectedItemIndex != -1);
    }
    private void  updatePlaceButton() {
        bidButton.setEnabled(!auctionServer.getAuction().hasItemUpForBid());
    }
    private void  updateStopButton() {
        stopButton.setEnabled(auctionServer.getAuction().hasItemUpForBid());
    }
```

The text fields are all updated according to the latest bid item information:

```
    private void  updateBidItemField() {
        if (auctionServer.getAuction().hasItemUpForBid())
            bidItemField.setText(auctionServer.getAuction().getBidItem().getName());
        else bidItemField.setText("");
    }

    private void  updateBidAmountField() {
        if (auctionServer.getAuction().hasItemUpForBid()) {
            java.text.DecimalFormat formatter = new java.text.DecimalFormat("$0.00");
            bidAmountField.setText(formatter.format(
                        auctionServer.getAuction().latestBid()));
        }
        else bidAmountField.setText("");
    }

    private void  updateBidderField() {
        if (auctionServer.getAuction().hasItemUpForBid())
            bidderField.setText(auctionServer.getAuction().latestBidderName());
        else bidderField.setText("");
    }

    private void  updatePictureLabel() {
        if (auctionServer.getAuction().hasItemUpForBid())
            pictureLabel.setIcon(new ImageIcon(
               auctionServer.getAuction().getBidItem().getPicture()));
        else
            pictureLabel.setIcon(BLANK_IMAGE);
    }
```

Finally, the status field shows whatever we pass it. Notice that this is not called from the **update()** method. Instead, whenever there is an important message, we call it:

```
    private void  updateStatus(String s) {
        statusField.setText(s);
    }
```

Now we look at the event handlers for the buttons.

When the ADD button is pressed, create a new **AuctionItem** and open the **AuctionItemDialog** box to edit it. We will set up **dialogFinished()** and **dialogCancelled()** methods to add or ignore the item as necessary.

```
    private void  handleAddAuctionItem() {
        newAuctionItem = new AuctionItem();
        new AuctionItemDialog(this, newAuctionItem).setVisible(true);
    }

    public void dialogFinished() {
        auctionServer.getAuction().add(newAuctionItem);
        selectedItemIndex = auctionServer.getAuction().getInventory().size()-1;
        update();
    }
    public void dialogCancelled() {} //do nothing
```

When the REMOVE button is pressed, remove the currently selected item:

```
    private void  handleRemoveAuctionItem() {
        if (selectedItemIndex != -1) {
            auctionServer.getAuction().getInventory().remove(selectedItemIndex);
            selectedItemIndex--;
            update();
        }
```

```
}
```

When the PLACE UP FOR BIDDING button is pressed, make the currently selected item to be the one that is placed up for bidding:

```
private void  handlePlaceForBid() {
    if (selectedItemIndex != -1) {
        auctionServer.getAuction().placeUpForBid(
                  (AuctionItem)auctionServer.getAuction().
                   getInventory().get(selectedItemIndex));
        update();
    }
}
```

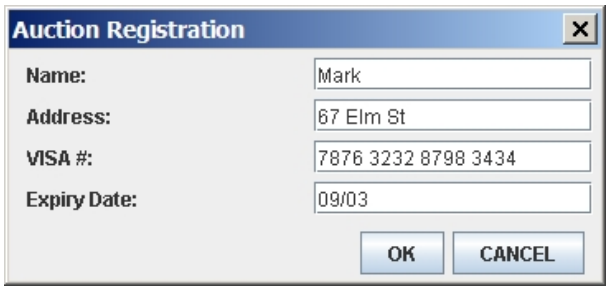
When the STOP BIDDING button is pressed, stop the current bid item from being bid on:

```
private void  handleStopBid() {
    auctionServer.getAuction().stopBidding();
    update();
}
```

Lastly, when the item is selected from the list, simply store its index in a local variable:

```
private void  handleSelectAuctionItem() {
    selectedItemIndex = itemsList.getSelectedIndex();
    update();
}
```

Well that is it for the server app!! Quite a lot of code, isn't it ? Now let us look at the Client-side GUI. First, we will consider the registration dialog. Notice that this dialog box is fairly straight forward. We pass in the **Customer** in the constructor and use this Customer's information to fill in the initial textFields. Since in our application, customers will only register once, this initial **Customer** object is probably filled with empty information. Nevertheless, we may want to use this dialog box in the future for editing purposes and in this case, our code will work fine. Notice also that when the OK button is clicked, the most recent data in the text fields is used to fill in the **Customer** object.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class  RegistrationDialog extends JDialog {

     // Store a pointer to the model for changes later
     private Customer customer;

     private JTextField nameTextField;
     private JTextField addressTextField;
     private JTextField visaTextField;
     private JTextField expireTextField;

     // The buttons and main panel
     private JButton okButton;
     private JButton cancelButton;

     // A constructor that takes the model and client as parameters
     public RegistrationDialog(Frame owner, Customer c){

         // Call the super constructor that does all the work of setting up the
     dialog
         super(owner,"Auction Registration",true);

         customer = c;  // Store the model

         // Make a panel with two buttons (placed side by side)
         JPanel buttonPanel = new JPanel();
         buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
         buttonPanel.add(okButton = new JButton("OK"));
```

```
            buttonPanel.add(cancelButton = new JButton("CANCEL"));

            // Make a panel with auction item information
            JPanel itemPanel  = new JPanel();

            // Create the textfields initially with the model's contents
            nameTextField = new JTextField(c.getName(), 15);
            addressTextField = new JTextField(c.getAddress(),15);
            visaTextField = new JTextField(c.getVisa(),15);
            expireTextField = new JTextField(c.getExpire(),15);

            // Set the layoutManager and add the components
            itemPanel.setLayout(new GridLayout(4,2,5,5));
            itemPanel.add(new JLabel("Name:"));
            itemPanel.add(nameTextField);
            itemPanel.add(new JLabel("Address:"));
            itemPanel.add(addressTextField);
            itemPanel.add(new JLabel("VISA #:"));
            itemPanel.add(visaTextField);
            itemPanel.add(new JLabel("Expiry Date:"));
            itemPanel.add(expireTextField);

            // Make the dialog box by adding bank account panel and button panel
            setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
            add(itemPanel);
            add(buttonPanel);

            // Prevent the window from being resized
            setSize(365, 170);
            setResizable(false);

            // Listen for ok button click
            okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event){
                    okButtonClicked(); }});

            // Listen for cancel button click
            cancelButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event){
                    cancelButtonClicked(); }});

            // Listen for window closing: treat like cancel button
            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent event) {
                    cancelButtonClicked(); }});
        }

        private void  okButtonClicked(){
            // Update model to show changed owner name
            customer.setName(nameTextField.getText());
            customer.setAddress(addressTextField.getText());
            customer.setVisa(visaTextField.getText());
            customer.setExpire(expireTextField.getText());

            // Tell the client that ok was clicked
            if (getOwner() != null)
                ((DialogClientInterface)getOwner()).dialogFinished();
            dispose();
        }

        private void  cancelButtonClicked(){
            // Tell the client that cancel was clicked
            if (getOwner() != null)
                ((DialogClientInterface)getOwner()).dialogCancelled();
            dispose();
        }

    }
```

Now the **AuctionCatalogDialog**, which displays the catalog returned from the server, should display a list of items and their pictures. The user should be able to browse around the list and see the pictures.

Notice that there is no client being passed in. In fact, there is no "response" that needs to be returned to the application. The user simply opens this window and does some browsing. So, there is no OK/CANCEL button combinations, simply a CLOSE button to close the window. Notice as well that the dialog box is non-modal, so the user can open a bunch of them.

A main method is provided to test out the code as well.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class  AuctionCatalogDialog  extends JDialog {

     // Store a pointer to the model for changes later
     private ArrayList<AuctionItem>  inventory;

     private JList   itemsList;
     private JLabel  picture;
     private JButton okButton;

     // A constructor that takes the model and client as parameters
     public AuctionCatalogDialog(Frame owner, ArrayList<AuctionItem> items){

         // Call the super constructor that does all the work of setting up the
     dialog
         super(owner,"Auction Inventory Catalog",false);

         // Store the model and client into instance variables
         inventory = items;

         itemsList = new JList(new Vector<AuctionItem>(items));
         itemsList.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxx");
         JScrollPane scrollPane = new JScrollPane(itemsList,
             ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
             ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
         picture = new JLabel("");

         // Set the layoutManager and add the components
         setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
         itemsList.setAlignmentX(FlowLayout.LEFT);
         add(new JLabel("Inventory"));
         add(itemsList);
         add(picture);
         add(okButton = new JButton("CLOSE"));

         // Prevent the window from being resized
         setSize(200, 450);
         setResizable(false);

         // Listen for CLOSE button click
         okButton.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent event){
                 dispose(); }});

         // Add a selection listener for the inventory list
         itemsList.addListSelectionListener(new ListSelectionListener() {
```

```
                public void valueChanged(ListSelectionEvent theEvent) {
                   if (itemsList.getSelectedValue() != null)
                        picture.setIcon(new ImageIcon(((AuctionItem)itemsList.
                                        getSelectedValue()).getPicture())); }});

            // Listen for window closing: treat like CLOSE button
            addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent event) {
                        dispose(); }});

            // Now make the dialog box appear
            setVisible(true);
        }

        // This is used for testing only
        public static void main(String args[]) {
            ArrayList<AuctionItem> v = new ArrayList<AuctionItem>();
            v.add(new AuctionItem("Antique Table",150.0f, "table.jpg"));
            v.add(new AuctionItem("JVC VCR",65.0f,"vcr.jpg"));
            v.add(new AuctionItem("Antique Cabinet",400.0f,"cabinet.jpg"));
            v.add(new AuctionItem("5-piece Drumset",190.0f,"drumset.jpg"));
            v.add(new AuctionItem("Violin & Case",100.0f,"violin.jpg"));
            v.add(new AuctionItem("13\" TV/VCR Combo",100.0f,"tvvcr.jpg"));
            v.add(new AuctionItem("486Dx2-66 Laptop",125.0f,"486laptop.jpg"));
            v.add(new AuctionItem("Rocking Chair",80.0f,"rockingchair.jpg"));
            v.add(new AuctionItem("1996 Mazda Miata",6500.0f,"miata.jpg"));
            JDialog dialog = new AuctionCatalogDialog(null, v);
        }

    }
```

Finally, we will examine the **AuctionClientApp** GUI application. Client users will want to have the ability to register for an auction. This should bring up the **RegistrationDialog**, and then use the **AuctionClient** to send this information to the server. Once registered, the client can then make bids.

The CATALOG button can be pressed at any time, and it should get the catalog from the sever, then bring up the **AuctionCatalogDialog** box.

The frame itself should display the information for the latest item which is being bid on. The user should be able to make a bid and press the MAKE BID button to send the bid to the server.

Notice as well that a **Timer** event is set up in the code. Every second, this timer event sends a request to the server for the latest **AuctionItem** information. This information is then returned to this client application and is shown in the window through an update call.

Once again, the code framework is given first, and then the update/event handler methods are shown afterwards.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.*;
import java.net.*;

public class  AuctionClientApp  extends JFrame implements DialogClientInterface {

    // This image gets shown when there is nothing up for bid
    private static ImageIcon BLANK_IMAGE =  new ImageIcon("blankItem.jpg");

    private JTextField   statusField;
```

```
private JTextField   bidItemField;
private JTextField   bidAmountField;
private JTextField   bidderField;
private JTextField   newBidField;
private JList        itemsList;
private JLabel       pictureLabel;
private JButton      bidButton;
private JButton      registerButton, catalogButton;

// This interface connects to an AuctionServer that handles all the work
private AuctionClient       auctionClient;
private float               bidToMake;
private javax.swing.Timer   updateTimer;

public AuctionClientApp() { this(new AuctionClient(new Customer())); }
public AuctionClientApp(AuctionClient c) {
    super("UNREGISTERED Client");
    auctionClient = c;
    bidToMake = 0.0f;

    initializeComponents();
    addListeners();
    update();
    updateTimer.start(); // Start requesting for updates
    setSize(200,460);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
}

private void addListeners() {
    // Add a listener for the REGISTER button
    registerButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
            handleRegistration();  }});

    // Add a listener for the REMOVE button
    catalogButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
            handleCatalogRequest(); }});

    // Add a listener for the MAKE BID button
    bidButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
            handleMakeNewBid(); }});

    // Add a listener for key presses in the MAKE BID textfield
    newBidField.getDocument().addDocumentListener(new DocumentListener() {
        private void handleBidBeingMade() {
           if (newBidField.getText().length() == 0)
                bidToMake = 0;
           else try {
                bidToMake = Float.parseFloat(newBidField.getText());
            } catch(NumberFormatException e) {
                bidToMake = 0;
            }
            updateMakeBidButton();
        }
        public void changedUpdate(DocumentEvent e) { handleBidBeingMade(); }
        public void insertUpdate(DocumentEvent e) { handleBidBeingMade(); }
        public void removeUpdate(DocumentEvent e) { handleBidBeingMade(); }});

    //Add a Timer event handler for updates
    updateTimer =  new javax.swing.Timer(1000, new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
            handleRequestUpdate(); }});
}

// Build the frame by adding all the components
private void initializeComponents() {
    GridBagLayout layout = new GridBagLayout();
    GridBagConstraints layoutConstraints = new GridBagConstraints();

    JPanel panel = new JPanel();
    panel.setLayout(layout);
    setContentPane(panel);

    registerButton = new JButton("Register");
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 0;
    layoutConstraints.gridwidth = 1;
    layoutConstraints.gridheight = 1;
```

```
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
layoutConstraints.weightx = 0.0;
layoutConstraints.weighty = 0.0;
layout.setConstraints(registerButton, layoutConstraints);
panel.add(registerButton);

catalogButton = new JButton("Catalog");
layoutConstraints.gridx = 1;
layoutConstraints.ipadx = 10;
layoutConstraints.ipady = 0;
layoutConstraints.anchor = GridBagConstraints.NORTHEAST;
layout.setConstraints(catalogButton, layoutConstraints);
panel.add(catalogButton);

JLabel aLabel = new JLabel("Item Up For Bidding");
layoutConstraints.gridx = 0;
layoutConstraints.gridy = 1;
layoutConstraints.gridwidth = 2;
layoutConstraints.fill = GridBagConstraints.BOTH;
layoutConstraints.insets = new Insets(5,5,5,5);
layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

pictureLabel = new JLabel();
layoutConstraints.gridy = 2;
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.weightx = 1.0;
layoutConstraints.weighty = 1.0;
layout.setConstraints(pictureLabel, layoutConstraints);
panel.add(pictureLabel);

aLabel = new JLabel("Item Description");
layoutConstraints.gridy = 3;
layoutConstraints.weightx = 0.0;
layoutConstraints.weighty = 0.0;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

aLabel = new JLabel("Last Bid Amount");
layoutConstraints.gridy = 5;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

aLabel = new JLabel("Last Bidder");
layoutConstraints.gridy = 7;
layout.setConstraints(aLabel, layoutConstraints);
panel.add(aLabel);

bidItemField = new JTextField();
bidItemField.setEditable(false);
layoutConstraints.gridy = 4;
layoutConstraints.fill = GridBagConstraints.HORIZONTAL;
layoutConstraints.weightx = 1.0;
layout.setConstraints(bidItemField, layoutConstraints);
panel.add(bidItemField);

bidAmountField = new JTextField();
bidAmountField.setEditable(false);
layoutConstraints.gridy = 6;
layout.setConstraints(bidAmountField, layoutConstraints);
panel.add(bidAmountField);

bidderField = new JTextField();
bidderField.setEditable(false);
layoutConstraints.gridy = 8;
layout.setConstraints(bidderField, layoutConstraints);
panel.add(bidderField);

bidButton = new JButton("Make Bid");
layoutConstraints.gridx = 1;
layoutConstraints.gridy = 9;
layoutConstraints.gridwidth = 1;
layoutConstraints.fill = GridBagConstraints.NONE;
layoutConstraints.weightx = 0.0;
layout.setConstraints(bidButton, layoutConstraints);
panel.add(bidButton);

newBidField = new JTextField();
layoutConstraints.gridx = 0;
layoutConstraints.gridwidth = 2;
```

```
            layoutConstraints.fill = GridBagConstraints.HORIZONTAL;
            layoutConstraints.anchor = GridBagConstraints.NORTHEAST;
            layoutConstraints.weightx = 1.0;
            layout.setConstraints(newBidField, layoutConstraints);
            panel.add(newBidField);

            statusField = new JTextField();
            statusField.setEditable(false);
            statusField.setBackground(new Color(255,255,255));
            statusField.setForeground(new Color(160,0,0));
            layoutConstraints.gridy = 10;
            layoutConstraints.gridwidth = 2;
            layoutConstraints.weightx = 10.0;
            layout.setConstraints(statusField, layoutConstraints);
            panel.add(statusField);
        }

        // Update all the components
        private void update() {
            updateMakeBidButton();
            updateBidItemField();
            updateBidAmountField();
            updateBidderField();
            updateNewBidField();
            updatePictureLabel();
        }

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

    }
```

Let us look now at the update methods. The MAKE BID button should be disabled when there is nothing to bid on. We will see that the timer updates will eventually get something returned which is the item that is being bid on:

```
    private void  updateMakeBidButton() {
        bidButton.setEnabled((auctionClient.getLatestAuctionItem() != null) &&
                             (bidToMake != 0));
    }
```

Each of the text fields are then updated according to the information from the latest bid item:

```
    private void  updateBidItemField() {
        AuctionItem item = auctionClient.getLatestAuctionItem();
        if (item == null) bidItemField.setText("");
        else bidItemField.setText(item.getName());
    }

    private void  updateBidAmountField() {
        AuctionItem item = auctionClient.getLatestAuctionItem();
        if (item == null) bidAmountField.setText("");
        else bidAmountField.setText(String.valueOf(item.getBid()));
    }

    private void  updateBidderField() {
        AuctionItem item = auctionClient.getLatestAuctionItem();
        if ((item == null) || (item.getPurchaser() == null))
            bidderField.setText("");
        else
            bidderField.setText(item.getPurchaser().getName());
    }
```

When there is nothing to bid on, we erase the text inside the text field that is being used to make a bid. This is not so important, but it allows us to clear the field in between bids, so that no weird bid amounts will be accidentally submitted to the server:

```
    private void  updateNewBidField() {
        AuctionItem item = auctionClient.getLatestAuctionItem();
        if (item == null) newBidField.setText("");
    }
```

The status field allows us to see what is going on. It displays error messages as well as server replies. We allow any string to be passed in here and we display whatever is passed in:

```
    private void  updateStatus(String s) {
        statusField.setText(s);
    }
```

Lastly, the picture for the item is displayed. The picture to be displayed depends on the item which is currently up for bid. However, the **AuctionItems** only store the picture filename, not the picture itself. In fact, we are "faking" something here. The server actually has all the images on its machine, not the client. So, when **AuctionItem** objects are sent to the client, the client only has the filename, not the files. So it is impossible to be able to display the image !!! However, since our test program has everything running in the same directory, we simply read the .gif files from there based on the name that was given to us by the server :). So ... we are cheating. To implemet things properly in a real system, we would have to transfer the image from the server to the client. However, in java, **Images** are not **Serializable**. Its a real pain! We could however, read the .gif file, send its bytes one at a time to the client and have the client save the bytes back to the file and then create an **ImageIcon** from it. That would work :).

```
private void  updatePictureLabel() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if (item != null)
        pictureLabel.setIcon(new ImageIcon(item.getPicture()));
    else
        pictureLabel.setIcon(BLANK_IMAGE);
}
```

Now for the event handlers. When the REGISTER button is pressed, we need to create a new **Customer** object and bring up the **RegistrationDialog** box to get its information. Then send a registration request to the **AuctionServer**:

```
private void  handleRegistration() {
    new RegistrationDialog(this, auctionClient.getCustomer()).setVisible(true);
}

public void dialogFinished() {
    if (auctionClient.register())
        setTitle("Client: " + auctionClient.getCustomer().getName());
    updateStatus(auctionClient.getServerReply().toString());
}
public void dialogCancelled() {} //do nothing
```

When the CATALOG button is pressed, send a request to the server for a catalog of items:

```
private void  handleCatalogRequest() {
    auctionClient.sendForCatalog();
    new AuctionCatalogDialog(this,
(ArrayList<AuctionItem>)auctionClient.getServerReply());
}
```

When the MAKE BID button is pressed, send the bid to the Server:

```
private void  handleMakeNewBid() {
    auctionClient.sendBid(auctionClient.getCustomer().getName(), bidToMake);
    updateStatus(auctionClient.getServerReply().toString());
}
```

When the timer ticks, send an update request to the server, then update the screen. We also detect changes in the **AuctionItem** so that we can display a nice message. For example, if there was nothing being bid on, then suddenly a new **AuctionItem** comes up for bidding, we display the message `"New Item Up For Bidding"`. If the item was already being bid on, then suddenly becomes **null**, we display a message stating `"Item No Longer Up For Bidding"`. Of course if this client made the last bid, then we should inform him/her that he/she now owns the item with a message such as: `"Antique Table SOLD to you for $100.00"`.

```
private void  handleRequestUpdate() {
    AuctionItem prevItem = auctionClient.getLatestAuctionItem();
    AuctionItem newItem = auctionClient.sendForUpdate();
    if (prevItem == null) {
        if (newItem != null)
            updateStatus("New Item Up For Bidding");
    }
    else if (newItem == null) {
           if (prevItem.getPurchaser().getName().equals(
                    auctionClient.getCustomer().getName()))
                updateStatus(prevItem.getName() + " SOLD to you for $" +
                             prevItem.getBid());
           else
                updateStatus("Item No Longer Up For Bidding");
        }
    else if (!prevItem.getName().equals(newItem.getName())) {
           if (prevItem.getPurchaser().getName().equals(
                    auctionClient.getCustomer().getName()))
                updateStatus(prevItem.getName() + " SOLD to you for $" +
                             prevItem.getBid());
           else
```

```
                    updateStatus("New Item Up For Bidding");
                }
            update();
        }
```

Well that is it! There was a LOT of code for this AuctionSystem. You can always add to it if you want to make a nice application.

---

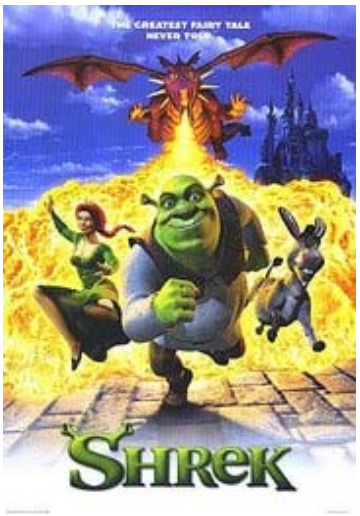
10 Animation

What's in This Set of Notes ?

Besides stationary graphics, we may also need to display graphics that change over time. Sometimes the graphics must change on regular intervals (such as with a timer). We therefore must learn a little about animation. We will look here at some very basic 2D animation concepts and what is involved with implementing some simple "motion" aspects of graphics programming.

Here are the individual topics found in this set of notes (click on one to go there):

10.1 Animation Concepts



Simply put, computer animation is the term used when successive images are displayed consecutively to cause the appearance of "motion" in the image. Unfortunately for us computer science guys n' gals, all of the hard work in animation is in the drawing of the images, not displaying. So:

- BAD NEWS: computer animation is intended for the artistically inclined.

Nevertheless, a person that is "lame" at drawing can still do animation.

Computer animation has become quite popular. Almost all computer animation that we see these days deals with 3D objects. The objects are modeled in the computer and then manipulated as wireframes using many techniques and various physics models. The movements can be quite complex and much of this requires a knowledge of 3D transformations (i.e., translations and rotations) in 3D. After the movement is completed, then coloring, lighting, shading and texture mapping is all applied. To complete a single fully rendered frame it can take many minutes depending on the machine. To make a 5 minute high quality computer animated video can take many hours to render with a single machine.

We will not deal with 3D animation at all here. We will simply animate 2D images (a.k.a. ***sprites***) and move them around on the screen.

The simplest form of 2D animation in JAVA is the use of animated gif files. We have seen this type of file with "brain.gif". Unfortunately, animated gifs are "stuck" in endless loops that do not allow user interaction. That is, we cannot control the sequence of images and hence the animation is very basic. We will therefore consider making our own animated sequences that are under user control.

Frames:

The first step in animation is to draw a set of pictures that are called ***frames***. These frames represent the different "movements" of the object to be animated. By displaying these frames in sequence, we achieve animation!!

For example, consider a stick person walking. We can do this with only two frames and just swap between them:

Frame 1	Frame 2	The Animation
		

It is not too nice since it is a little "jumpy". The problem is that there is no smooth transition between the frames. We can make a big improvement just by introducing one more picture and duplicating the 2nd frame twice to produce a 4 frame sequence:

Frame 1	Frame 2	Frame 3	Frame 4	The Animation
				

Notice that it looks much smoother. But hey, it seems slower! This is because in both cases we have introduced a 1/4 second delay between frames. In the 2-frame situation it takes 1/2 of a second to complete a cycle while in the 4-frame case, it takes a full second. We can reduce the inter-frame delay to 1/8 of a second and we will be fine.

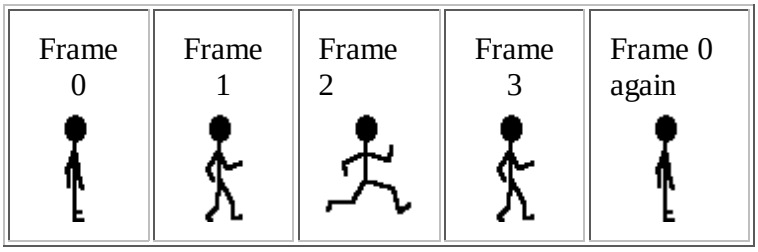
This change in time between frame displaying is known as the ***frame rate***. Here are some varying frame rates:

5/20 sec.	4/20 sec.	3/20 sec.	2/20 sec.	1/20 sec.
				

There isn't too much more to say about animation right now.

10.2 Simple Animation and Threads

Our first example is to use the **CardLayout** manager to display consecutive images one after another. We will place an **ImageIcon** on each of 4 buttons and when the button is pressed, we will cause consecutive images to be displayed by just showing the next card (i.e., button) in the layout. As long as the pictures represent consecutive images, we are fine. Here are the images that we will use, shown in order from 0 to 3:




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutAnimation  extends JFrame implements ActionListener {
    private CardLayout  cardLayoutManager;

    public CardLayoutAnimation (String title) {
        super(title);
        cardLayoutManager = new CardLayout(0,0);
        setLayout(cardLayoutManager);

        // Add the 8 buttons, each with a different picture as the icon
        for (int i=0; i<4; i++) {
            JButton aButton = new JButton(new ImageIcon("Stick" + i + ".gif"));
            add(String.valueOf(i), aButton);
            aButton.addActionListener(this);
        }
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(80, 120);
    }

    // Cause a 1/10 second delay when called
    private void delay() {
        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e){ /* do nothing */ }
    }

    // Implements the listener behavior, e.g. go to next button in the stack
    public void actionPerformed(ActionEvent theEvent) {
        for (int i=0; i<4; i++) {
            cardLayoutManager.next(getContentPane());
            // Now redraw the window by updating its appearance
            update(getGraphics()); // a standard JComponent method
            delay();
        }
    }

    // Create main method to execute the application
    public static void main(String args[]) {
        new CardLayoutAnimation("CardLayout Animation").setVisible(true);
    }
}
```

Notice the use of **Thread.sleep(100)** to provide a 1/10th second delay between image flips. Also notice the use of **update(getGraphics())** to ensure that the frame is updated after every image change. This is a standard method available for all JComponents and it usually simply repaints the component immediately. Why do we need to do this redraw update for each image ? Remember, while executing code within an event handler, no other events can be handled, including events responsible for redrawing components. So we have to explicitly call this update method in order to see screen changes while we are still in our event handling code.

What happens if we increase the delay between images ? If the delay is too long, no events are handled and so the interface seems to lock up. Let us add another button that prints out a message when pressed.

We will increase the delay of the drawing so that this button becomes unresponsive while the animation is taking place. A similar kind of delay may occur when we need to do a lot of animating, or perhaps when the animation steps themselves require a lot of computation.

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

public class CardLayoutAnimation2  extends JFrame implements
ActionListener {

    private CardLayout  cardLayoutManager;
    private JPanel  panel;

    public CardLayoutAnimation2 (String title) {
        super(title);
        setLayout(new FlowLayout());

        // Make a panel to hold the buttons using a card layout manager
        cardLayoutManager = new CardLayout(0,0);
        panel = new JPanel();
        panel.setLayout(cardLayoutManager);
        for (int i=0; i<4; i++) {
            JButton aButton = new JButton(new ImageIcon("Stick" + i +
".gif"));
            panel.add(String.valueOf(i), aButton);
            aButton.addActionListener(this);
        }

        // Make another button, then add the panel and this button to
the frame
        JButton b = new JButton("Press Me");
        add(panel);
        add(b);

        // Print a simple message when the button is pressed
        b.addActionListener( new ActionListener() {
             public void actionPerformed(ActionEvent theEvent) {
                System.out.println("Hello");
        }});

        setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
        setSize(80,160);
    }

    // Cause a 1 second delay when called
    private void delay() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e){}
    }

    // implements the listener behavior, e.g. go to next button in the
stack
    public void actionPerformed(ActionEvent theEvent) {
        for (int i=0; i<4; i++) {
            cardLayoutManager.next(panel);
            update(getGraphics());
            delay(); // Now a long delay
        }
    }

    public static void main(String args[]) {
        new CardLayoutAnimation2("Animated Title").setVisible(true);
    }
}
```

When the code runs, try pressing the "Press Me" button. It prints a message. Now try clicking the image button. Notice that the animation is now slower, since we lengthened the frame rate to 1 second. Try pressing the "Press Me" button a few times during the animating process. Nothing happens. That is because the events are being queued, but not handled. Once the animation stops, you will see that the queued messages are handled all at once. Try it again, this time try to close the window during the animation.

So ... clearly this is a problem. How can we fix it ? We need to create a separate thread (i.e., a separate process) to handle the animation, while the main process handles all other application events. Generally, whenever you need to have an event handler that is computationally intensive (i.e., it is slow at what it needs to do), you should make a separate thread to do it. Let us look at making a separate thread in our example.

One way to make a separate process for doing our animation is to make a separate class, which will be a subclass of the **Thread** class. We will make one called **AnimationThread** that will do the animation on the panel which we specify:

```
import java.awt.*;
import javax.swing.*;
public class AnimationThread  extends Thread {
    private JPanel aPanel;
    private CardLayout aLayoutManager;

    public AnimationThread(JPanel p, CardLayout c) {
        aPanel = p;
        aLayoutManager = c;
    }

    // Make a brief delay
    private void delay() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e){}
    }

    public void run() {
        for (int i=0; i<4; i++) {
            aLayoutManager.next(aPanel);
            aPanel.update(aPanel.getGraphics());
            delay();
        }
    }
```

```
}
```

All **Threads** MUST have a **run()** method which contains all the code to be done in the thread. It is kind of like the "main" method of a **JFrame.** When it is done executing, the **Thread** is done too.

Now, we can change the listener in our **CardAnimationExample2** to be:

```
public void actionPerformed(ActionEvent theEvent) {
    new AnimationThread(panel, cardLayoutManager).start();
}
```

Try the code. You will notice now that as the animation is working, the application still responds to button clicks and window closing events.

A second way to do all of this would have been NOT to make a separate class of **Thread**, but in fact make our application implement the **Runnable** interface. We would still write a **run()** method, but everything goes into one class:

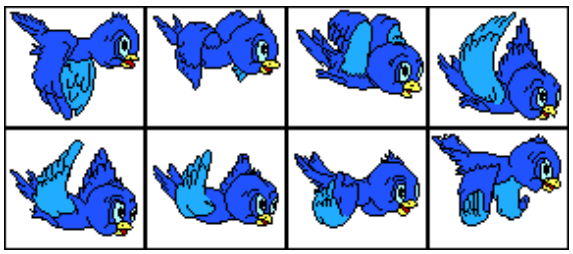
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutAnimation4  extends JFrame
                                   implements ActionListener, Runnable {

    private CardLayout  cardLayoutManager;
    private JPanel  panel;

    public CardLayoutAnimation2 (String title) {
        super(title);
        setLayout(new FlowLayout());

        // Make a panel to hold the buttons using a card layout manager
        cardLayoutManager = new CardLayout(0,0);
        panel = new JPanel();
        panel.setLayout(cardLayoutManager);
        for (int i=0; i<4; i++) {
            JButton aButton = new JButton(new ImageIcon("Stick" + i +
".gif"));
            panel.add(String.valueOf(i), aButton);
            aButton.addActionListener(this);
        }

        // Make another button, then add the panel and this button to
the frame
        JButton b = new JButton("Press Me");
        add(panel);
        add(b);

        // Print a simple message when the button is pressed
        b.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent theEvent) {
                System.out.println("Hello");
        }});

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(80,160);
    }

    // Cause a 1 second delay when called
    private void delay() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e){}
    }

    // Handle the animation
    public void run() {
```

```
            for (int i=0; i<4; i++) {
                cardLayoutManager.next(panel);
                panel.update(panel.getGraphics());
                delay();
            }
        }

        // implements the listener behavior, e.g. go to next button in the
    stack
        public void actionPerformed(ActionEvent theEvent) {
            // Start a new thread using the run() method from this class
            new Thread(this).start();
        }

        public static void main(String args[]) {
            new CardLayoutAnimation2("Animated Title").setVisible(true);
        }
    }
```

Often, this is the simplest way to do it :).

10.3 Kinetic Animation

We will now look at kinetic animation, that is ... animation that moves as opposed to staying in one location. Our task will be to move a bird around in a window. Here are the frames that we will use:



These frames are numbered 0 to 7 starting at the top left and numbering across first.

Getting this bird to fly in a single location on the screen is now easy. We can use the **CardLayout** manager if we want to ... however, we will want to make the bird move around. Our choice will be to display it on a **JFrame** or **JPanel**. We will write the code such that the user gets to make the bird fly by causing it to flap its wings whenever he/she clicks the mouse. Notice that our bird will only fly from left to right.

To start, let us consider making a **FlyingBird** class to represent the bird.
What information (i.e., instance variables) should we keep for the bird ?

- `Point currentLocation;  // the bird's coordinate on the screen`
- `Image[] images;  // the frames of the bird (we will have 8)`
- `int currentFrame;  // the frame currently being displayed`

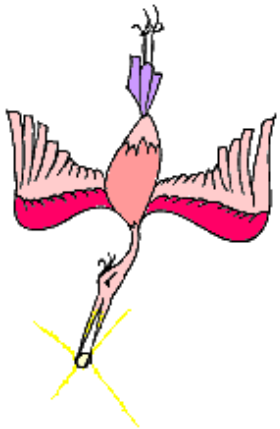
Here is the start of our code. We will make two class variables to keep track of how many frames the bird will have and the width of each frame (in pixels):

```
public static int NUM_FRAMES = 8;
public static int WIDTH = 78;
```

When making a bird, we must load up the images from the file and store them into the **images** array. We probably also want to choose some starting frame (in our case #4) as well as a starting location. We will not supply a "get" method for the image array. Instead, we will write a method called **appearance()** that will return the image corresponding to the current frame of the bird. Here is a start to our code:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FlyingBird  {
    public static int NUM_FRAMES = 8;
    public static int WIDTH = 78;

    // Instance variables
    private Point   currentLocation;  // the bird's coordinate on the
screen
    private Image[] images;           // the frames of the bird (we will
have 8)
    private int     currentFrame;     // the frame currently being
displayed

    // Default constructor
    public FlyingBird() {
        currentFrame = 4;
        currentLocation = new Point(100,100);
        images = new Image[FlyingBird.NUM_FRAMES];
        for (int i=0; i<NUM_FRAMES; i++) {
            images[i] = Toolkit.getDefaultToolkit().getImage(
                                    "BIRDX" + (i+1) + ".gif");
        }
    }

    public int getCurrentFrame() { return currentFrame; }
    public Point getCurrentLocation() { return currentLocation; }

    // Return the image representing the bird's current appearance
    public Image appearance() {
        return (images[currentFrame]);
    }
}
```

So, each bird will keep track of its current frame, location and appearance. We will just need to supply some methods that we can call from the application that will tell the bird to ***advance*** its frame to the next one and move forward as necessary. For some added realism, we will have the bird "fall" when its wings are not flapping. That brings up some good questions:



- When is the bird flying and when is it falling ?
- What does it look like when its falling ?
- How do we make it fall ?

These are easily answered.

- The bird is flying when its wings are flapping down and falling otherwise.
- We will choose frame 3 to represent the "falling" frame.
- We make it fall by increasing the y value of the location.

Here is the **advance()** method that moves the bird forward while advancing the frame and also takes

into accout gravity:

```
// This method allows the frames to advance as well as move the bird
public void advance() {
    // Move the bird forward 10 pixels
    currentLocation.translate(10,0);

    // Make gravity pull the bird down, unless its wings are flapping
    if (currentFrame > 3)
        currentLocation.translate(0,-5);
    else
        currentLocation.translate(0,5);

    if (currentFrame != 3)
        currentFrame = (currentFrame + 1) % 8;
}
```

We will also want to make the bird flap its wings. To do this, we can just "jump" to the frame that starts the flapping:

```
// Set the frame to show the bird starting to flap its wings
public void flapWings() {
    currentFrame = 4;
}
```

The "follow-through" from the flapping (i.e., the continuation and completion of the flapping motion) will be handled by successive calls to **advance()**.

Now that we have the bird working, let us get the interface going. We will make a **JFrame** and color the background white. We will set up a **Timer** that will cause the bird to advance every 1/5 of a second.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FlyingBirdApp  extends JFrame implements ActionListener {
    private static int WIDTH = 600;
    private static int HEIGHT = 400;

    private FlyingBird aBird; // model
    private Timer  aTimer;
    private Image  background;

    public FlyingBirdApp (String title, FlyingBird theBird) {
        super(title);
        aBird = theBird;

        // We can use nice scenery for the background
        background = Toolkit.getDefaultToolkit().getImage("beach.jpg");

        // Start the timer so that the bird comes to life
        aTimer = new Timer(100, this);
        aTimer.start();

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(540, 210);
    }

    // This is the timer event handler
    public void actionPerformed(ActionEvent e) {
        aBird.advance();
        if (aBird.getCurrentLocation().x > WIDTH)
            aBird.getCurrentLocation().x = -1 * FlyingBird.WIDTH;
        repaint();
    }
```

```
    public void paint(Graphics g) {
        g.drawImage(background, 0, 0, null);
        g.drawImage(aBird.appearance(), aBird.getCurrentLocation().x,
                    aBird.getCurrentLocation().y, this);
    }

    // Create main method to execute the application
    public static void main(String args[]) {
        new FlyingBirdApp("Flying Bird", new
FlyingBird()).setVisible(true);
    }
}
```

Notice that the **Timer** event causes the bird to advance and then repaints the frame. It also checks to see if the bird goes off the end of the frame and brings it back around to the left side again. The **paint()** method merely gets the appearance of the bird (i.e., the image) and displays it at the bird's current location. Note that we use **paint()** instead of **paintComponent()** since this is a **JFrame**, not a **JPanel.**

Now we just need to add the **MousePressed** event handler to make the bird flap its wings:

```
// Add a MousePressed event handler
addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        aBird.flapWings();
}});
```

Here is the end result of our hard work: