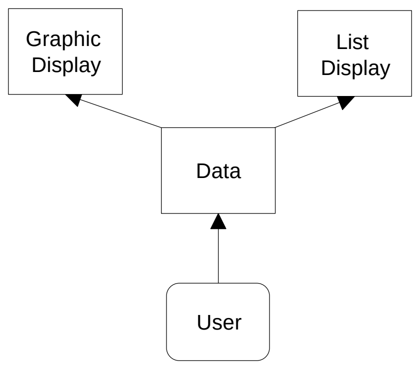
THE OBSERVER PATTERN

In our new, more sophisticated windowing world, we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. For example, you might represent stock price changes both as a graph and as a table or list box. Each time the price changes, we'd expect both representations to change at once without any action on our part.

We expect this sort of behavior because there are any number of Windows applications, like Excel, where we see that behavior. Now there is nothing inherent in Windows to allow this activity and, as you may know, programming directly in Windows in C or C++ is pretty complicated. In Java, however, we can easily make use of the Observer Design Pattern to cause our program to behave in this way.

The Observer pattern assumes that the object containing the data is separate from the objects that display the data, and that these display objects *observe* changes in that data. This is simple to illustrate as we see below.



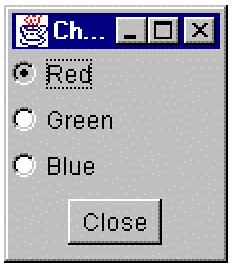
When we implement the Observer pattern, we usually refer to the data as the Subject and each of the displays as Observers. Each of these observers registers its interest in the data by calling a public method in the Subject. Then, each observer has a known interface that the subject calls when the data change. We could define these interfaces as follows:

```
abstract interface Observer {
//notify the Observers that a change has taken place
   public void sendNotify(String s);
}
//==============================================
abstract interface Subject {
//tell the Subject you are interested in  changes
   public void registerInterest(Observer obs);
}
```

The advantage of defining these abstract interfaces is that you can write any sort of class objects you want as long as they implement these interfaces, and that you can declare these objects to be of type Subject and Observer no matter what else they do.

Watching Colors Change

Let's write a simple program to illustrate how we can use this powerful concept. Our program shows a display frame containing 3 radio buttons named Red, Blue and Green as shown below:



This main window is the Subject or data repository object. We create this window using the JFC classes in the following simple code:

```
public class Watch2L extends JFrame
   implements ActionListener, ItemListener, Subject {
   Button Close;
   JRadioButton red, green, blue;
   Vector observers;
//------------------------------------------
   public Watch2L()    {
   super("Change 2 other frames");
//list of observing frames
   observers = new Vector();
//add panel to content pane
   JPanel p = new JPanel(true);
   p.setLayout(new BorderLayout());
   getContentPane().add("Center", p);

//vertical box layout
```

```
   Box box = new Box(BoxLayout.Y_AXIS);
   p.add("Center", box);
//add 3 radio buttons
   box.add(red = new JRadioButton("Red"));
   box.add(green = new JRadioButton("Green"));
   box.add(blue = new JRadioButton("Blue"));

 //listen for clicks on radio buttons
   blue.addItemListener(this);
   red.addItemListener(this);
   green.addItemListener(this);

 //make all part of same button group
   ButtonGroup bgr = new ButtonGroup();
   bgr.add(red);
   bgr.add(green);
   bgr.add(blue);
```

Note that our main frame class implements the Subject interface. That means that it must provide a public method for registering interest in the data in this class. This method is the *registerInterest* method, which just adds Observer objects to a Vector:

```
public void registerInterest(Observer obs)    {
      //adds observer to list in Vector
      observers.addElement(obs);
   }
```

Now we create two observers, once which displays the color (and its name) and another which adds the current color to a list box.

```
//---------create observers---------
   ColorFrame cframe = new ColorFrame(this);
   ListFrame lframe = new ListFrame(this);
```

When we create our ColorFrame window, we register our interest in the data in the main program:

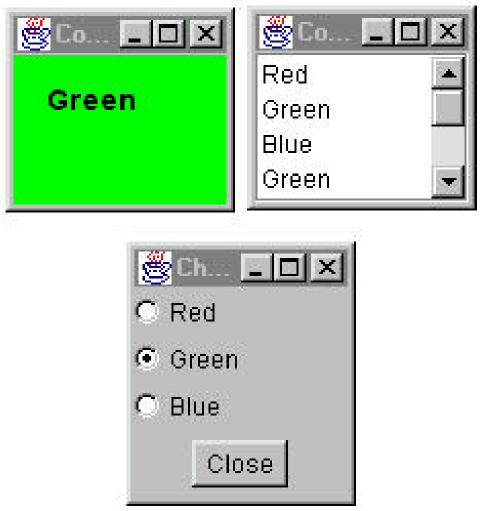
```
class ColorFrame extends Jframe
  implements Observer {
   Color color;
   String color_name="black";
   JPanel p = new JPanel(true);
//----------------------------------------
   public ColorFrame(Subject s)    {
    super("Colors");             //set frame caption
    getContentPane().add("Center",  p);
    s.registerInterest(this);  //register with Subject
    setBounds(100, 100, 100, 100);
    setVisible(true);
   }
//-------------------------------------
   public void sendNotify(String s)    {
       //Observer is notified of change here
      color_name = s;      //save color name
       //set background to that color
      if(s.toUpperCase().equals("RED"))
         color = Color.red;
      if(s.toUpperCase().equals("BLUE"))
        color =Color.blue;
      if(s.toUpperCase().equals("GREEN"))
        color = Color.green;
      setBackground(color);
   }
//-------------------------------------
   public void paint(Graphics g)    {
      g.drawString(color_name, 20, 50);
   }
```

Meanwhile in our main program, every time someone clicks on one of the radio buttons, it calls the *sendNotify* method of each Observer who has registered interest in these changes by simply running through the objects in the observers Vector:

```
   public void itemStateChanged(ItemEvent e)    {
      //responds to radio button clicks
      //if the button is selected
      if(e.getStateChange() == ItemEvent.SELECTED)
         notifyObservers((JRadioButton)e.getSource());
   }
   //---------------------------------------
   private void notifyObservers(JRadioButton rad)    {
      //sends text of selected button to all observers
      String color = rad.getText();
      for (int i=0; i< observers.size(); i++)   {

((Observer)(observers.elementAt(i))).sendNotify(color);
      }
   }
```

In the case of the ColorFrame observer, the sendNotify method changes the background color and the text string in the frame panel. In the case of the ListFrame observer, however, it just adds the name of the new color to the list box. We see the final program running below:



The Message to the Media

Now, what kind of notification should a subject send to its observers? In this carefully circumscribed example, the notification message is the string representing the color itself. When we click on one of the radio buttons, we can get the caption for that button and send it to the observers. This, of course, assumes that all the observers can handle that string representation. In more realistic situations, this might not always be the case, especially if the observers could also be used to observe other data objects. Here we undertake two simple data conversions:

1. we get the label from the radio button and send it to the observers, and

2. we convert the label to an actual color in the ColorFrame observer.

In more complicated systems, we might have observers that demand specific, but different, kinds of data. Rather than have each observer convert the message to the right data type, we could use an intermediate Adapter class to perform this conversion.

Another problem observers may have to deal with is the case where the data of the central subject class can change in several ways. We could delete points from a list of data, edit their values, or change the scale of the data we are viewing. In these cases we either need to send different change messages to the observers or send a single message and then have the observer ask which sort of change has occurred.

Th JList as an Observer

Now, what about that list box in our color changing example? We saved it for last because as we noted earlier in the Adapter class discussion, the JList is rather different in concept than the List object in the AWT. You can display a fixed list of data in the JList by simply putting the data into a Vector or String array. However, if you want to display a list of data that might grow or otherwise change, you need to put that data into a special data object derived from the AbstractListModel class, and then use that class in the constructor to the JList class. Our ListFrame class looks like this:

```
class ListFrame extends JFrame
   implements Observer {
   JList list;
   JPanel p;
   JScrollPane lsp;
   JListData listData;

   public ListFrame(Subject s)    {
    super("Color List");
    //put panel into the frmae
    p = new JPanel(true);
    getContentPane().add("Center",  p);
    p.setLayout(new BorderLayout());
    //Tell the Subject we are interested
    s.registerInterest(this);

    //Create the list
    listData = new JListData();  //the list model
    list = new JList(listData);  //the visual list
    lsp = new JScrollPane();     //the scroller
    lsp.getViewport().add(list);
    p.add("Center", lsp);
    lsp.setPreferredSize(new Dimension(100,100));
    setBounds(250, 100, 100, 100);
    setVisible(true);
   }
   //--------------------------------
   public void sendNotify(String s)    {
      listData.addElement(s);
   }
}
```

We name our ListModel class **JListData**. It holds the Vector that contains the growing list of color names.

```
class JListData extends AbstractListModel {
   private Vector data;   //the color name list
   public JListData()   {
      data = new Vector();
   }
   public int getSize()   {
      return data.size();
   }
   public Object getElementAt(int index)   {
      return data.elementAt(index);
   }
  //add string to list and tell the list about it
   public void addElement(String s)   {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1, data.size());
   }
}
```

Whenever the ColorList class is notified that the color has changed, it calls the *addElement* method of the JListData class. This method adds the string to the Vector, and then calls the *fireIntervalAdded* method. This base method of the AbstractListModel class connects to the JList class, telling that class that the data have changed. The JList class then redisplays the data as needed. There are also equivalent methods for two other kinds of changes: *fireIntervalRemoved* and *fireContentsChanged*. These represent the 3 kinds of changes that can occur in a list box: here each sends its own message to the JList display.

The MVC Architecture as an Observer

As we noted in Chapter 3, the JList, JTable, and JTree objects all operate as observers of a data model. In fact, all of the visual components derived from JComponent can have this same division of labor between the data and the visual representation. In JFC parlance, this is referred to as the Model-View-Controller (MVC) architecture, where the data are represented by the Model, and the View by the visual component. The Controller is the communication between the Model and View objects, and may be a separate class or it may be inherent in either the model or the view. This is the case for the JFC components, and they are all examples of the Observer pattern we've just been discussing.

Consequences of the Observer Pattern

Observers promote abstract coupling to Subjects. A subject doesn't know the details of any of its observers. However, this has the potential disadvantage of successive or repeated updates to the Observers when there are a series of incremental changes to the data. If the cost of these updates is high, it may be necessary to introduce some sort of change management, so that the Observers are not notified too soon or too frequently.

When one client makes a change in the underlying data, you need to decide which object will initiate the notification of the change to the other observers. If the Subject notifies all the observers when it is changed, each client is not responsible for remembering to initiate the notification. On the other hand, this can result in a number of small successive updates being triggered. If the clients tell the Subject when to notify the other clients, this cascading notification can be avoided, but the clients are left with the responsibility of telling the Subject when to send the notifications. If one client "forgets," the program simply won't work properly.

Finally, you can specify the kind of notification you choose to send by defining a number of update methods for the Observers to receive depending on the type or scope of change. In some cases, the clients will thus be able to ignore some of these notifications