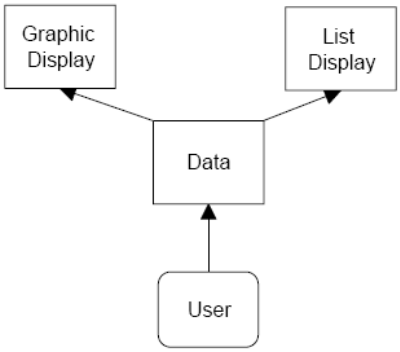
Observer Pattern

What is it? And an Example

- The observer pattern (sometimes known as publish/ subscribe) is a design pattern used to observe the state of an object.
- For example, we often 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.
  - In Java we can easily make use of the Observer Design Pattern to cause our program to behave in this way.

"The Observer Pattern Assumption"

- 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.



The Observer Philosophy

- 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.

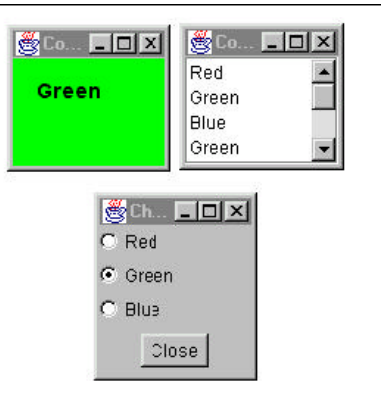
Observer and Subject Interfaces

- Lets define some abstract interfaces. They will allow us to write any sort of class objects we want as long as they implement these interfaces, and that we can declare these objects to be of type Subject and Observer no matter what else they do.

```
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);
}
```


An Example

- The user-controlled radio-button choice changes the display in two windows:
  - First, changes the background color and the text string in the first panel.
  - Second, adds the name of the new color to the list box.



Lets Create the Main Window!

- This main window is the Subject or data repository object.

```
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);
```


More Main Window Code

```
   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);
```

Wait, there's more!

- 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);
   }
```


The Observers

- 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);
```

Lets Create One of the Frames

- When creating the 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);
   }
```

Invoking sendNotify()

- Every time someone clicks on one of the radio buttons, the main program 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);
   }
}
```


sendNotify() Explained

- 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.

Notification Type

- In this carefully constructed 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.
- In more complicated systems, we might have observers that demand specific, but different, kinds of data.

Notification Type (contd…)

- 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.