More Frameworks and a Case Study

CS151

Chris Pollett

Oct. 31, 2005.

Outline

- Layouts
- Handling Events
- Input/Output Framework
- Decorator Pattern

Layouts

- Last Day, we briefly described the setLayout method to set the layout of a Container.
- We also briefly described using add to add a component to a container.
- We now briefly described some different layout classes.
  - FlowLayout. Can be constructed with FlowLayout(align, hGap, vGap), FlowLayout(align), or FlowLayout(). align can be one of FlowLayout.LEFT, FlowLayout.CENTER, or FlowLayout.RIGHT. In the no arg case, align is centered and hGap and vGap are 5px. Idea to to place as many components as possible on the same row according to the alignment, then move down a row.
  - Gridlayout. Can be constructed with GridLayout(r, c, hGap, vGap), or GridLayout(r,c) or GridLayout(). Here r is the numbers of rows, c is the number of columns, hGap and vGap are as before. add() then fills row by row.
  - BorderLayout. Can be constructed with BorderLayout(hGap, vGap), or BorderLayout(). When add component specify one of BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER.

Example

```
JFrame myFrame = new JFrame(“hello”);
myFrame.setLayout(new FlowLayout());
JPanel myPanel = new JPanel();
myPanel.setLayout(new BorderLayout());
myPanel.add(new JButton(“hi”),
   BorderLayout.CENTER);
myFrame.add(myPanel);
```

Handling Events

- GUI components communicate with the rest of the application through *events*. These represent inputs or actions.
- The *source* of an event is the component from which the event originated
- A *listener* of an event is an object that receives events and processes the event.
- Event Handling in Java consists of:
  - Determining the type of events to be handled and their associated listeners. For example, button clicks generate ActionEvents which are handled by an ActionListener. (This is an interface)
  - Define listener classes that implement the listener interface:

    class MyButtonHandler implements ActionListener

    { public void actionPerformed(ActionEvent e) { /*code*/}

    }
  - Create instances of the component which is the source of the event and add an instance of the listener we created.

    Button b = new Button("hello"), b2= new Button("goodbye");

    MyButtonHandler h = new MyButtonHandler();

    b.addActionListener(h);

    b2.addActionListener(h);

The Event Handling Process

What roughly happens when processing an event is:

- When an event is triggered, the Java run time determines its source and type.
- If a listener for this type is registered with the source, then an event object is created.
- For each listener that listens to this type of event, the Java run time invokes the appropriate event-handling method of the listener and passes the event objects as a parameter.

Example 1

```
public class MyFrame extends Frame implements ActionListener
{
    public MyFrame()
    {
        super();
        Button b = new Button();
        b.addListener(this);
        l =new Label("old text");
        add(b); add(l);
    }
    public void actionPerformed(ActionEvent e)
    {
        l.setText("Button has been pushed");
    }
  public Label l;
}
```

Example Using Inner Class

```
public class MyFrame extends Frame
{
      public MyFrame()
      {
              super();
              Button b = new Button();
              b.addListener(this);
              l =new Label("old text");
              add(b); add(l);
      }
      public class MyListener implements ActionListener
      {
            public void actionPerformed(ActionEvent e)
            {
              l.setText("Button has been pushed");
            }
      }
     public Label l;
}
```

Example Using Anonymous Inner Class

```
public class MyFrame extends Frame
{
	public MyFrame()
	{
		super();
		Button b = new Button();
		l =new Label(“old text”);
		b.addListener(new ActionListener() // if listener has multiple methods use an adapter class
		{
		    public void actionPerformed(ActionEvent e)
		     {
		        l.setText(“Button has been pushed”);
		});
	}
	Label l;
}
```

Input/Output Framework

- Java supports two types of I/O (input/output)
  - Stream I/O: a stream is a sequence of bytes. Stream based I/O supports reading or writing data sequentially
  - Random Access I/O supports reading and writing data at any position in a file.

Byte Streams

- The most basic and primitive stream I/O capabilities are declared in two abstract class InputStream and OutputStream.
- InputStream supports: read(), read(ba), read(ba,off,len), skip(n), close()
- OutputStream supports: write(b), write(ba), write(ba, off, len), skip(n), close()
- The class FileInputStream and FilterInputStream extends InputStream and FileOutputStream and FilterOutputStream extends OutputStream.
- They have one parameter constructors which take a String filename.

Data Input and Output Streams

- Reading and writing on a byte by byte level can be quite awkward.
- The two interfaces DataInput and DataOutput are designed to give a more useable way of doing things.
- DataInput has methods: readBoolean(), readByte(), readChar(), readDouble(), readFloat(), readInt(), readLong(), readShort(), readUTF().
- DataOutput has methods: writeBoolean(b), writeByte(b), writeChar(c), writeDouble(d), writeFloat(f), writeInt(i), writeLong(l), writeShort(s), writeUTF(s).
- FilterInputStream and FilterOutputStream are subclassed by DataInputStream and DataOutputStream which respectively implement DataInput and DataOutput.
- The constructor of DataInputStream takes an InputStream; the constructor of DataOutputStream takes an OutputStream.
- Objects of these classes then store this InputStream/OutputStream in a has-a way. This is an example of the decorator pattern.

Buffered Streams

- Another example of using filter streams is BufferedInputStream and BufferedOutputStream.
- Again, their constructor take InputStream/OutputStream and stores it in a has a relation.
- Commands issued to the buffer stream are again filtered to the low-level stream in another example of the decorator pattern.

Object Stream

- Another useful pair of interfaces is ObjectInput which has readObject() and ObjectOutput which has writeObject().
- ObjectInputStream and ObjectOutputStream are two filter streams which implement these interfaces.

Example

```
try
{
    DataInputStream d = new DataInputStream( new
       FileInputStream(“my.txt”));
    d.readBoolean();
    d.close();
}
catch(IOException ie)
{
    ie.printStackTrace();
}
```

Decorator Pattern