Chapter 9 Buttons, Canvases, Controllers & Viewers

9.0 Introduction

This chapter is about the method of interactive program organization known as the *Controller*, *Model*, *Viewer* model. Most of the chapter introduces suitable standard Java classes with which to implement the model. The terms controller, model, and viewer appear in section 9.6.

The Java *Component* types discussed below: *Panel*, *Canvas*, *Label*, *TextField*, and *Button* are quite convenient to use for this type of organization. Most of this chapter is an introduction to these component types in Java and how to get them to work together.

NOTE:

The programs in previous chapters used *mousePressed* and *keyPressed* for all interaction. A considerable amount of setup required to use these was hidden (in the class *EventPanel*). From this point on *mousePressed*, *keyPressed* and *EventPanel* are rarely used. When *Buttons* and *TextFields* are used in Java programs, the alternate event handlers discussed in this chapter are ordinarily used. These event handlers require similar setup, which is included in the examples below.

9.1 *Components*

A Java program usually displays several visible *Components*. In the previous chapters of the text, the programs are created by customizing a single *Component*. The only exceptions being those programs that display *TextFields* — each *TextField* being a separate *Component*. Programs from this point on use several visible components.

Five types of *Component* appear in the diagram. These will be sufficient for our purposes.

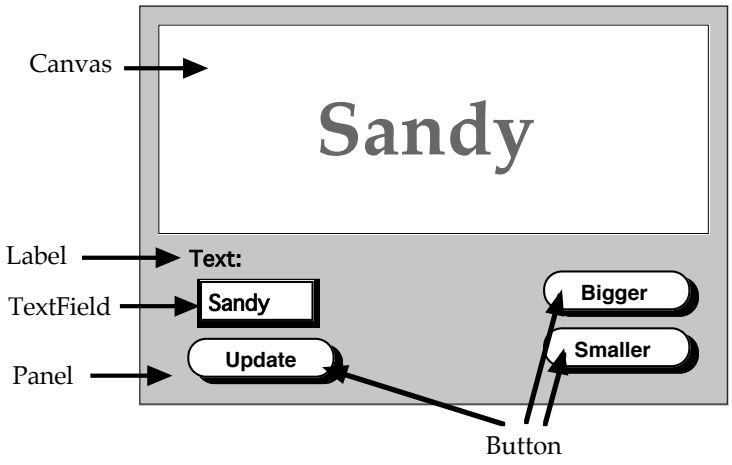
A *Panel* is generally used as a background onto which the other components are placed. Of the five types of component shown, only a *Panel* can have other components placed onto it.

A *Canvas* is a component whose graphics can be customized by extending *Canvas* and supplying a *paint* method. While the graphics on a *Panel* can also be customized in this way, custom graphics generally appear on a *Canvas*.

A *Label* displays text on a single line.

A *TextField* displays text on a single line contained in a box. Usually the user can edit this text. A text field comes with methods that make it easy for the program to react the user's editing.

A *Button* invites the user to click on it. A button comes with methods that make it easy for the program to react the user's clicking.



9.2 Placing *Components*

Each program in this text displays a *Panel*. All of the interactive programs start with either *PanelApplet* or *PanelApplication*, either of which places the *Panel* appropriately so that it will be displayed. Other *Components* are placed on this *Panel*.

The components should be created and placed in the constructor of the *Panel*. The class *TextPanel* below places the *Label*, the *TextField*, the *Canvas*, and one *Button* (of the program shown above) onto itself.

The use of the standard class *Canvas* below, makes the canvas blank. A real program would extend *Canvas* to provide a *Canvas* with a *paint* method that displayed the program's output.

```
public class TextPanel extends Panel
    {
    private static final Color BACK = new Color(200, 225, 255);

    private Label textLabel;
    private TextField textField;
    private Canvas canvas;
    private Button updateButton;

    public TextPanel()
        {
        setLayout(null);              // prevents Java from placing and sizing the components
        setBackground(BACK);          // sets the color of the Panel to light blue-green

        textLabel = new Label("Text:");
        textLabel.setSize(80, 25);
        textLabel.setLocation(30, 140);
        textLabel.setBackground(BACK);  // matches the background of the Label to the Panel
        add(textLabel));                // places the Label onto the Panel

        textField = new TextField("Sandy");
        textField.setSize(80, 25);
        textField.setLocation(20, 170);
        textField.setBackground(Color.white);  // makes the background of the Field white
        add(textField));                       // places the Field onto the Panel

        canvas = new Canvas();
        canvas.setSize(280, 120);
        canvas.setLocation(10, 10);
        canvas.setBackground(Color.white); // makes the background of the Canvas white
        add(canvas));                      // places the Canvas onto the Panel

        updateButton = new Button("Update");
        updateButton.setSize(80, 25);
        updateButton.setLocation(20, 200);
        add(updateButton));                // places the Button onto the Panel
    }
}
```

Each of the components has its size and location set. The colors of most of the components are also set. In some systems colors of components are inherited from the *Panel*, in others they are not. Set each color yourself so that the components will always be colored as you intend. *Buttons* seem to color appropriately, so I let buttons be an exception and don't attempt to color them.

NOTE:

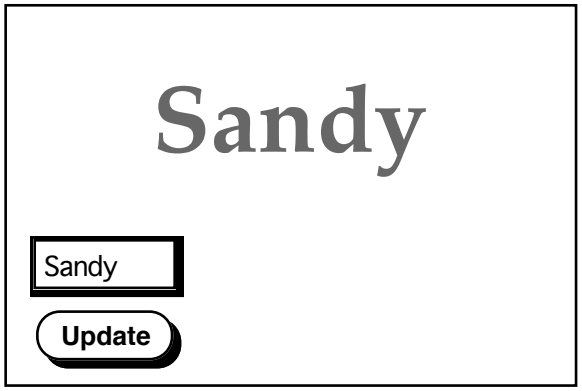
At this point we have a program that does not react to the user. How to set things up so that the components communicate with each other is the subject of the rest of the chapter.

9.3 *Buttons* and 'Actions'

There is no *Canvas* in this section. The example is written by customizing the *Panel*, as was done in previous chapters. The use of a *Canvas*, which is the 'right' way to do things, first appears in section 9.5.

The usual way to interact with a *Button*, is to use the event handler *actionPerformed*. Using *actionPerformed* requires two 'setup statements'. These statements are included in the sample program below:

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

public class ChangeableName extends Panel implements ActionListener ← note 1

private String theName;
private TextField theField;
private Button theButton;

public ChangeableName()
    {
    theName = "Sandy";

    setLayout(null);

    theField = new TextField();
    theField.setLocation(20, 170);
    theField.setSize(80, 25);
    theField.setText(theName);
    add(theField);

    theButton = new Button("Update");
    theButton.setLocation(20, 210);
    theButton.setSize(80, 25);
    add(theButton);

    theButton.addActionListener(this); ← note 2
    }

public void paint(Graphics g)
    {
    g.setColor(Color.red);
    g.setFont(new Font("Serif", Font.BOLD, 36));
    g.drawString(theName, 30, 200);
    }

public void actionPerformed(ActionEvent e)
    {
    Object source = e.getSource();
    if (source == theButton)
        {
        theName = theField.getText();
        repaint();
        }
    }
}
```



This statement informs *theButton* that the *actionPerformed* method is in this class.

Notes:

1. A class that contains an *actionPerformed* event handler declares that it *implements ActionListener*.
2. A *Button* can activate an *actionPerformed* event handler. Only after an *addActionListener* method has been called, will it do so.

9.4 *TextFields* and 'Actions'

TextFields can easily be made to act much like *Buttons* — pressing return or enter while typing in a *TextField* can call an *actionPerformed* method. To have the *TextField* in the program above call *actionPerformed*, add a line to the constructor:

```
theButton.addActionListener(this);
theField.addActionListener(this);          ← add this line
```

Also, check for the *TextField* in the *actionPerformed* method:

```
public void actionPerformed(ActionEvent e)
    {
    Object source = e.getSource();
    if (source == theButton || source == theField)     ← add a reference to theField
        {
        theName = theField.getText();
        repaint();
        }
    }
```

TextFields and *TextEvents*

There is an alternate event handler which can be used with *TextFields*. This handler, *textValueChanged*, is called whenever the user changes the contents of the *TextField*. If *textValueChanged* is used in the preceding example, the name in large print will change as the user types. To use *textValueChanged*:

1. Remove the button (and all references to the button).
   A button cannot call *textValueChanged* and the button will be useless anyway. Remove it.
2. Change *implements ActionListener* to *implements TextListener*.
3. Change the method name *actionPerformed* to *textValueChanged*.
4. Change the parameter type in *textValueChanged* to *TextEvent*.

Using a *TextField* to Input a Number

If you just want to display a number as typed, there is no particular difficulty. The way in which text is handled above will do. If you want to do arithmetic with the number, you will have to convert the string variable that you get from the text field to a numeric variable (*double* or *int*).

The code to convert a string variable to a numeric variable is:

double

```
String s = numberField.getText();
String error = "";
double d = -99999;
boolean ok = true;
s = s.trim();
try
    {
    d = (new Double(s)).doubleValue();
    }
catch (NumberFormatException err)
    {
    error = err;
    ok = false;
    }
```

int

```
String s = numberField.getText();
String error = "";
int i = -99999;
boolean ok = true;
s = s.trim();
try
    {
    i = (new Integer(s)).intValue();
    }
catch (NumberFormatException err)
    {
    error = err;
    ok = false;
    }
```

The method *trim* removes spaces from either end of the string. User are often unaware of such spaces, but conversion may fail if spaces are present.

After the above code has been executed:

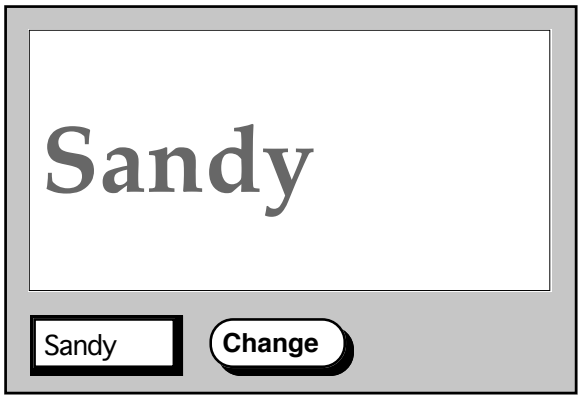
- the value of *d* or *i* will be the desired number.
- *error* will be the empty string.
- *ok* will be true.

or

- *d* or *i* will be –99999.
- *error* will describe what went wrong.
- *ok* will be false.

A simplified version of this number conversion code appears in the methods *convertToDouble* and *convertToInt* in section 3.8.

9.5 Canvases



Painting on a *Panel* that holds *Buttons* and *TextFields* is rarely done. The painting is usually done on a separate *Canvas*.

A modified version of the program above is described below. In this version there are four visible components:

1. A *Panel*: the shaded background.
2. A *TextField*: shows "Sandy".
3. A *Button*: labeled "Change".
4. A *Canvas*: shows "Sandy" in a large font.

The *Panel* is described by extending the class *Panel* to create a customized *Panel*: *ChangeableName*.

The version of *ChangeableName* for this program appears on the next page.

The *TextField* and the *Button* are created using the standard classes *TextField* and *Button*.

The *Canvas* is described by extending the class *Canvas* to create a customized *Canvas*: *NameCanvas*.

The class *NameCanvas* appears immediately below.

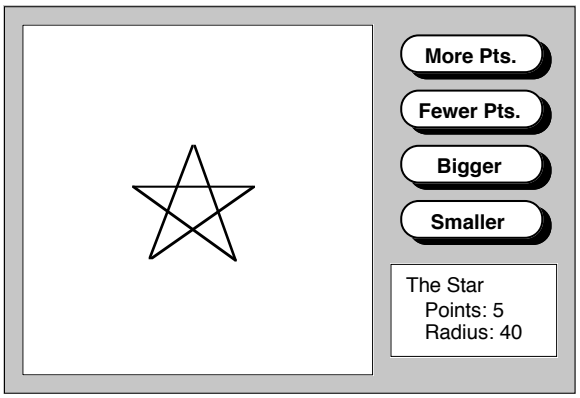
```
import java.awt.*;

public class NameCanvas extends Canvas
    {
    private String name;

    public NameCanvas(String n)
        {
        name = n;
        }

    public void paint(Graphics g)
        {
        g.setColor(Color.red);
        g.setFont(new Font("Serif", Font.BOLD, 36));
        g.drawString(name, 30, 90);
        }

    public void setName(String n)
        {
        name = n;
        }
    }
```

Note the background color setup for each *Component*. It is important to set the background colors since different systems will set the colors of the *TextField* and *Canvas* differently if you don't set them explicitly. I generally don't bother setting the color of a *Button*, and never seem to have a problem.

Note that the call to *repaint* (in the *actionPerformed* method) is *theCanvas.repaint()*. Just calling *repaint()* would *repaint* the *Panel* — which is not what you want, since the panel provides only the unchanging background.

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

public class ChangeableName extends Panel implements ActionListener
    {
    private TextField theField;
    private Button theButton;
    private NameCanvas theCanvas;

    public ChangeableName()
            {
        setLayout(null);
        setBackground(new Color(200, 225, 255));  ← sets the color of the Panel (light blue)

        theField = new TextField("Sandy");
        theField.setLocation(50, 180);
        theField.setSize(80, 25);
        theField.setBackground(Color.white);   ←  sets the color of the TextField (white)
        add(theField);                            Set the background of the field,
                                                  otherwise it may inherit from the Panel

        theButton = new Button("Change");
        theButton.setLocation(50, 180);
        theButton.setSize(80, 25);
        add(theButton);

        theCanvas = new NameCanvas("Sandy");
        theCanvas.setLocation(10, 10);
        theCanvas.setSize(380, 160);
        theCanvas.setBackground(Color.white);  ←  sets the color of the Canvas (white)
        add(theCanvas);

        theButton.addActionListener(this);
        theField.addActionListener(this);
        }

    public void actionPerformed(ActionEvent e)
        {
        Object source = e.getSource();
        if (source == theButton || source == theField)
            {
            theCanvas.setName(theField.getText());
            theCanvas.repaint();
            }
        }
    }
```

9.6 A Controller, a Model, and a Viewer



The general idea is that the data is kept in one object (the model), a second object (the viewer) is used to display the data, and a third object (the controller) is used to link user actions with data manipulations .

This example uses two additional objects.

One of these is a second viewer, which is included with the intent of clarifying the distinction between model and viewer.

The second is a 'set up' object that creates the model, the viewers, the controller, and the buttons that interact directly with the user. The setup object also sets up communication between the various objects.

The classes that define this program appear below in the following order:

The setup object:	described in the class *StarPanel*.
The controller:	described in the class *Manager*.
The two viewers:	described in the classes *StarCanvas* and *StatsCanvas*.
The model:	described in the class *Star* (a similar class appears in chapter 4).

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

public class StarPanel extends Panel
    {
    private Button moreButton, fewerButton, biggerButton, smallerButton;
    private Star theStar;
    private StarCanvas starCanvas;
    private StatsCanvas statsCanvas;
    private Manager manager;

    public StarPanel()
        {
        setLayout(null);
        setBackground(new Color(200, 225, 255));

        moreButton = new Button("More Points");
        moreButton.setLocation(305, 20);
        moreButton.setSize(80, 25);
        add(moreButton);

        fewerButton = new Button("Fewer Points");
        fewerButton.setLocation(305, 60);
        fewerButton.setSize(80, 25);
        add(fewerButton);

        biggerButton = new Button("Bigger");
        biggerButton.setLocation(305, 100);
        biggerButton.setSize(80, 25);
        add(biggerButton);

        smallerButton = new Button("Smaller");
        smallerButton.setLocation(305, 140);
        smallerButton.setSize(80, 25);
        add(smallerButton);
```

```
        theStar = new Star(140, 140);

        starCanvas = new StarCanvas(theStar); ← starCanvas displays theStar.
        starCanvas.setLocation(10, 10);
        starCanvas.setSize(280, 280);
        starCanvas.setBackground(Color.white);
        add(starCanvas);
```

Since *starCanvas* displays *theStar,* it must know of *theStar*'s existence. *StarCanvas* is constructed after *theStar* — it cannot be informed of a *Star* that is not yet created.

```
        statsCanvas = new StatsCanvas(theStar); ← statsCanvas displays data from theStar.
        statsCanvas.setLocation(300, 200);
        statsCanvas.setSize(90, 70);
        statsCanvas.setBackground(Color.white);
        add(statsCanvas);
```

StatsCanvas displays information about *theStar,* and thus must know of *theStar*'s existence. Like the *starCanvas, statsCanvas* is constructed after *theStar*.

```
        manager = new Manager(theStar, starCanvas, statsCanvas,
                          moreButton, fewerButton, biggerButton, smallerButton);
```

Since the manager will be checking the buttons, changing the star, and repainting the canvases, it must know of their existence. The manager is constructed after these other objects, since it is to be informed of their existence.

```
        moreButton.addActionListener(manager); ← manager contains the actionPerformed method
        fewerButton.addActionListener(manager);
        biggerButton.addActionListener(manager);
        smallerButton.addActionListener(manager);
        }
    }

import java.awt.*;
import java.awt.event.*;

public class Manager implements ActionListener ← this class contains an actionPerformed method
    {
    private Star theStar;
    private StarCanvas starCanvas;
    private TextCanvas textCanvas;
    private Button moreButton, fewerButton, biggerButton, smallerButton;

    public Manager(Star s, StarCanvas sCanvas, TextCanvas tCanvas, Button mButton,
                                        Button fButton, Button bButton, Button sButton)
        {
        theStar = s; starCanvas = sCanvas; textCanvas = tCanvas; moreButton = mButton;
        fewerButton = fButton; biggerButton = bButton; smallerButton = sButton;
        }

    public void actionPerformed(ActionEvent e)
        {
        Object source = e.getSource();
        if (source == moreButton)
            theStar.morePoints();
        else if (source == fewerButton)
            theStar.fewerPoints();
        else if (source == biggerButton)
            theStar.bigger();
        else if (source == smallerButton)
            theStar.smaller();
        starCanvas.repaint(); textCanvas.repaint();
        }
    }
```

```
import java.awt.*;

public class StarCanvas extends Canvas
   {
   private Star theStar;

   public StarCanvas(Star s)
      {
      theStar = s;
      }

      public void paint(Graphics g)
      {
      theStar.paint(g);
      }
   }
```

```
import java.awt.*;

public class StatsCanvas extends Canvas
   {
   private Star theStar;

   public StatsCanvas(Star s)
      {
      theStar = s;
      }

   public void paint(Graphics g)
      {
      g.setColor(Color.black);
      g.drawString("The Star:", 10, 15);
      g.drawString("Points: " + theStar.getPoints(), 20, 40);
      g.drawString("Radius: " + theStar.getRadius(), 20, 55);
      }
   }
```

← Coordinates on a *Canvas* are measured from the upper left corner of the *Canvas*. Hence the small coordinates.

The *StarCanvas* class above takes advantage of the fact that the *Star* class already had a paint method. It would be equally effective to move the paint method to the *StarCanvas* and remove the paint method from the *Star* class. This would make the separation between the roles of the model and the viewer more explicit.

```
public void paint(Graphics g)
   {
   g.setColor(Color.black);
   int points = theStar.getPoints()
   int radius = theStar.getRadius();
   int x = theStar.getX();
   int y = theStar.getY();
   int almostHalf = points/2;
   double angle = almostHalf*2*Math.PI/points;
   for (int p = 0; p < points; p++)
      {
      g.drawLine(x + (int)(radius * Math.sin(angle * p)),
         y - (int)(radius * Math.cos(angle * p)),
         x + (int)(radius * Math.sin(angle * (p+1) )),
         y - (int)(radius * Math.cos(angle * (p+1) )) );
      }
   }
```

← alternate *paint* method for *StarCanvas*

```
import java.awt.*;

public class Star
   {
   private int x, y, points, radius;

   public Star(int centerHorizontal, int centerVertical)
      {
      x = centerHorizontal;
      y = centerVertical;
      points = 5;
      radius = 50;
      }

   public void paint(Graphics g)
      {
      paintStar(g);
      }

   private void paintStar(Graphics g)
      {
      g.setColor(Color.black);
      int almostHalf = points/2;
      double angle = almostHalf*2*Math.PI/points;
      for (int p = 0; p < points; p++)
         {
         g.drawLine(x + (int)(radius * Math.sin(angle * p)),
            y - (int)(radius * Math.cos(angle * p)),
            x + (int)(radius * Math.sin(angle * (p+1) )),
            y - (int)(radius * Math.cos(angle * (p+1) )) );
         }
      }

   public void bigger()
      {
      radius += 10;
      }

   public void smaller()
      {
      if (radius > 10)
         radius -= 10;
      }

   public void morePoints()
      {
      points += 2;
      }

   public void fewerPoints()
      {
      if (points > 3)
         points -= 2;
      }

   public int getRadius()
      { return radius; }

   public int getPoints()
      { return points; }

   public int getX()
      { return x; }

   public int getY()
      { return y; }

   }
```

Exercise — 9.6

1. Reorganize one or more of the programs from chapter 6 that manipulate a list of names. Put the list on a *Canvas* and use the Controller, Model, Viewer model.