CHAPTER 17: GRAPHICAL USER INTERFACES AND APPLETS

17.1 GUI VERSUS COMMAND LINE INTERFACE

All the programs we have written so far use **terminal input/output** (also called the **command line interface**). With this approach, input is via lines entered by the user on the keyboard, and output is via lines displayed on the display monitor. An alternative approach is a **graphical user interface** (GUI). Programs that use a GUI create one or more windows on the display screen that hold a variety of graphical components, such as menus that list choices and buttons that trigger actions on a mouse click.

The principal advantages of GUIs are

- A GUI requires less keyboard input. For example, with a GUI, a user can input a file name by selecting it from a menu with a single mouse click. In contrast, with terminal I/O, the user has to type in the file name.
- A GUI can easily inform the users of all the choices that are available using the various components on the window. For example, by displaying four buttons, a GUI communicates to the user that four choices are available.
- By giving users more choices during the execution of a program, it gives users more control a program.
- A GUI can provide a standard user interface across a large variety of programs.

17.2 OUR FIRST GUI PROGRAM

To create a GUI program from scratch would require an enormous amount of complex programming. Fortunately, we do not have to do this. We can simply use predefined classes that support GUI. These classes are in two packages: `java.awt` (the **Abstract Windows Toolkit**) and `javax.swing`.

Before we examine our first GUI program, let's introduce some important terminology. A **frame** is a window that is displayed on the screen. A **container** is an object to which we can add components. The container associated with a frame is called the **content pane**.

To create and display a window, we

1) Create a frame object from the `JFrame` class.
2) Configure the frame. Specifically, we specify its title, size, and the action that occurs when its close button is clicked.
3) Add components to the frame's content pane.
4) Make the frame visible.

Consider the window in Fig. 17.1a. It contains two components: a button and a label:

a)

b)

```
 1 import javax.swing.*;
 2 import java.awt.*;           // needed for Color class
 3 class GUI1 extends JFrame
 4 {
 5    private Container contentPane;
 6    private JButton button1;
 7    private JLabel label1;
 8    //---------------------------------
 9    public GUI1()
10    {
12       setTitle("GUI1");              // sets title on frame
13       setSize(400, 100);             // sets frame width, height
14       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//set close action
15
16       // get content pane of frame
17       contentPane = getContentPane();
18
19       // configure content pane
20       contentPane.setLayout(new FlowLayout());
21       contentPane.setBackground(Color.GREEN);
22
23       // add button and label to the content pane of the frame
24       button1 = new JButton("This is a button");
25       contentPane.add(button1);
26       label1 = new JLabel("This is a label");
27       contentPane.add(label1);
28
29       setVisible(true);                  // make frame visible
30    }
31    //---------------------------------
32    public static void main(String[] args)
33    {
34       GUI1 window = new GUI1();          // create window
35    }
36 }
```

Figure 17.1

The program that creates this window is in Fig. 17.1b. The `GUI1` class in this program extends the `JFrame` class (line 3). Thus, it inherits everything in `JFrame`. When `main` creates an object from `GUI1` (line 34), it creates a frame object that represents the window that is displayed on the screen.

The `GUI1` class contains a constructor (see lines 9 to 30) that sets up the frame. The constructor does this by calling methods inherited from the `JFrame` class. These methods are `setTitle` (line 12), `setSize` (line 13), `setDefaultCloseOperation` (line 14), `getContentPane` (line 17), and `setVisible` (line 29).

The title on the blue bar at the top of the window is set with

```
12       setTitle("GUI1");              // sets title on frame
```

The size of the frame is specified in units of pixels. For example, on line 13

```
13       setSize(400, 100);             // sets frame width, height
```

we are setting the width and height of the frame to 400 and 100 pixels, respectively. A **pixel** is the smallest possible dot a display monitor can create on the screen. Pixel size depends on the size and resolution of the monitor. For example, a monitor with a resolution of 1024 by 768 pixels would have 1024 pixels in each row and 768 pixels in each column. Thus, a width specification of 1024 pixels would reach from the left side all the way to the right side of the monitor. However, on a monitor of the same size but with a 2048 by 768 resolution, a 1024 width would be only half of the horizontal screen size.

The default close operation is set with

```
14       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //set close action
```

The `JFrame.EXIT_ON_CLOSE` is a constant in the `JFrame` class. It indicates that the window should exit (i.e., terminate) when the close button (the button marked with × in the upper right corner of the frame) is clicked. If, instead, we specify the constant `JFrame.HIDE_ON_CLOSE` in place of `JFrame.EXIT_ON_CLOSE`, the window disappears from view but does not terminate when its close button is clicked.

The constructor calls `getContentPane` to get the content pane of the frame:

```
17       contentPane = getContentPane();
```

Remember that the content pane is the container associated with the frame.

On line 20, the constructor calls the `setLayout` method in the content pane to specify how components should be laid out on the content pane:

```
20       contentPane.setLayout(new FlowLayout());
```

The `FlowLayout` object that is passed to `setLayout` indicates that components should be placed on the content pane in rows left to right. When one row is full, placement moves to the next row.

The constructor calls `setBackground` in the content pane to set the background color of the frame:

```
21       contentPane.setBackground(Color.GREEN);
```

`Color.GREEN` is a constant that represents the color green. The constants that are available for colors are given in Fig. 17.2.

```
Color.BLACK
Color.DARK_GRAY
Color.GRAY
Color.LIGHT_GRAY
Color.WHITE
Color.CYAN
Color.MAGENTA
Color.PINK
Color.RED
Color.ORANGE
Color.YELLOW
Color.GREEN
Color.BLUE
```

Figure 17.2

The constructor calls the `add` method in the content pane to add each component to the content pane. For example, the constructor creates a button with

```
24        button1 = new JButton("This is a button");
```

It then adds the button to the content pane one line 25 with

```
25        contentPane.add(button1);
```

For simplicity, we give each component a name consisting of its kind ("button", "label", etc.) suffixed with a sequence number. For example, we name the reference variable to the button component `button1`. In a GUI with multiple buttons, we would name them `button1`, `button2`, `button3`, and so on. If a GUI has many components of the same kind, a better approach to naming them is to give each component a name that describes its function. With such names, it is easier to remember what each component does.

The last step is to make the frame visible:

```
29        setVisible(true);                    // make frame visible
```

The program in Fig. 17.1b is in a file named `GUI1.java`. To compile it, enter

```
javac GUI1.java
```

To run it, enter

```
java GUI1
```

The window it creates will then appear on the display screen (see Fig. 17.1a). If you click on the button labeled with `This is a button`, nothing happens. This lack of response occurs because our program does not associate any action with this button. To terminate the window, click on the close button in the upper right corner of the frame.

17.3 USING PANELS

The program in Fig. 17.1b places the button and label directly on the content pane of the frame. A more typical approach, however, is to place components on a panel and then place the panel on the content pane of the frame. The program in Fig. 17.3b is like the program in Fig. 17.1b except that it uses panels.

a)

b)

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 class GUI2 extends JFrame
 4 {
 5    private Container contentPane;
 6    private JPanel panel1, panel2;
 7    private JButton button1;
 8    private JLabel label1;
 9    //---------------------------------
10    public GUI2()
11    {
12       setTitle("GUI2");
13       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15       panel1 = new JPanel();              // create panel 1
16       panel1.setLayout(new FlowLayout()); // configure panel 1
17       panel1.setBackground(Color.GREEN);
18       panel1.setPreferredSize(new Dimension(100, 100));
19       button1 = new JButton("This is a button");
20       panel1.add(button1);                // add button to panel 1
21
22       panel2 = new JPanel();              // create panel 2
23       panel1.setLayout(new FlowLayout()); // configure panel 2
24       panel2.setBackground(Color.YELLOW);
25       panel2.setPreferredSize(new Dimension(300, 100));
26       label1 = new JLabel("This is a label");
27       panel2.add(label1);                 // add label to panel 2
28
29       // get content pane of frame
30       contentPane = getContentPane();
31
32       contentPane.setLayout(new FlowLayout());
33       contentPane.add(panel1);            // add panels to content pane
34       contentPane.add(panel2);
35
36       pack();     // adjust frame size to accommodate panels
37       setVisible(true);
38    }
39    //---------------------------------
40    public static void main(String[] args)
41    {
42       GUI2 window = new GUI2();
43
44    }
45 }
```

Figure 17.3

It creates two panels with

```
15       panel1 = new JPanel();              // create panel 1
```

and

```
22       panel2 = new JPanel();              // create panel 2
```

It creates and places a button on panel 1:

```
19       button1 = new JButton("This is a button");
20       panel1.add(button1);               // add button to panel 1
```

It creates and places the label on the panel 2:

```
26       label1 = new JLabel("This is a label");
27       panel2.add(label1);                // add label to panel 2
```

It places both panels on the content pane of the frame:with

```
33       contentPane.add(panel1);           // add panels to content pane
34       contentPane.add(panel2);
```

The preferred size of each panel is specified by a call to the `setPreferredSize` method in each panel object. For example, to set the width and height of `panel1` to 100 and 100 pixels, respectively, we use

```
18       panel1.setPreferredSize(new Dimension(100, 100));
```

`Panel1` is sized accordingly unless there is not enough room on the screen .

Unlike the program in Fig. 17.1b, the program in Fig. 17.3b does not call the `setSize` method to set the size of the frame. Instead, it calls `pack` (line 36) which causes the frame size to adjust so that it is just big enough to accommodate the panels it holds:

```
36       pack();     // adjust frame size to accommodate panels
```

Using panels allows for more flexibility. Each panel is a separate container. Thus, each panel can be configured differently. For example, the two panels created by the program in Fig. 17.3b have different sizes and colors (compare lines 17 and 18 in Fig. 17.3b with lines 24 and 25).

17.4 EVENTS AND ACTION LISTENERS

An **event** in a GUI is a action taken by the user on one of the GUI components. For example, clicking on a button is an event.

When an event occurs on a component in a GUI, the following sequence occurs (see Fig. 17.4):

1) An `ActionEvent` object is created that represents that event.

2) If the component is associated with a listener object, the `ActionEvent` object is passed to the `actionPerformed` method in the listener object. The code in the `actionPerformed` method is then executed.

To handle events, a program should import the `java.awt.event` package (see line 3 in Fig. 17.5b).

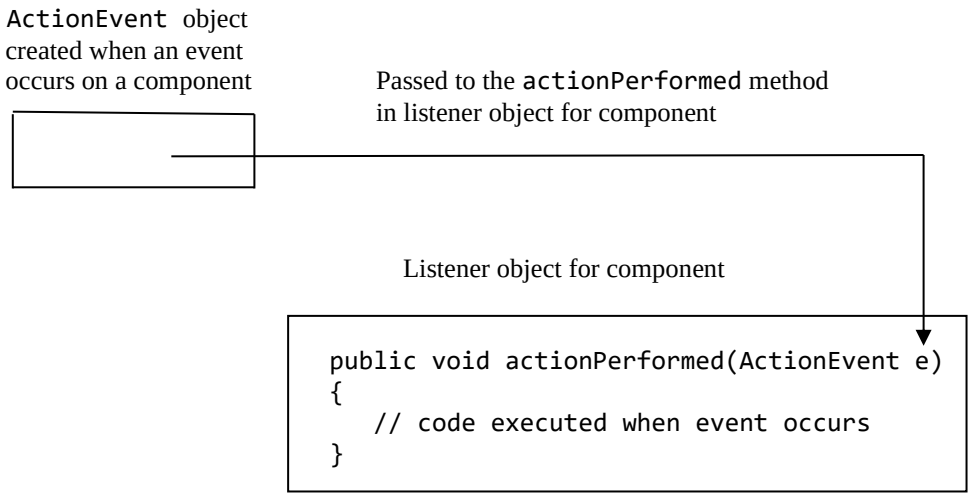
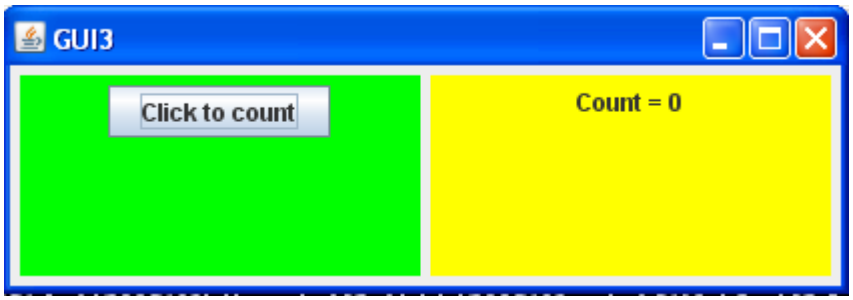


Figure 17.4

Let's examine the program in Fig. 17.5b to see how to create a listener object and associate it with a button component. The window that this program creates contains a button and a label (see Fig. 17.5a). Each time the button is clicked, the count in the label is incremented.

a)



b)

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4 class GUI3 extends JFrame
 5 {
 6    private Container contentPane;
 7    private JPanel panel1, panel2;
 8    private JButton button1;
 9    private JLabel label1;
10    private int count;
11    //---------------------------------
12    public GUI3()
13    {
14       setTitle("GUI3");
15       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17       panel1 = new JPanel();     // create panel 1
18       panel1.setLayout(new FlowLayout());
19       panel1.setBackground(Color.GREEN);
20       panel1.setPreferredSize(new Dimension(200, 100));
21       button1 = new JButton("Click to count");
22       button1.addActionListener(new Listener());  // add listener
23       panel1.add(button1);
24
25       panel2 = new JPanel();     // create panel 2
26       panel2.setLayout(new FlowLayout());
27       panel2.setBackground(Color.YELLOW);
28       panel2.setPreferredSize(new Dimension(200, 100));
29       count = 0;
30       label1 = new JLabel("Count = " + count);
31       panel2.add(label1);
32
33       // get content pane of frame
34       contentPane = getContentPane();
35
36       contentPane.setLayout(new FlowLayout());
37       contentPane.add(panel1);    // add panels to content pane
38       contentPane.add(panel2);
39
40       pack();
41       setVisible(true);                   // make window visible
42    }
43    //---------------------------------
44    public static void main(String[] args)
45    {
46       GUI3 window = new GUI3();          // create window
47
48    }
49    //---------------------------------
50    private class Listener implements ActionListener
51    {
52       public void actionPerformed(ActionEvent e)
53       {
54          label1.setText("Count = " +  ++count);  // set label
55       }
56    }
57 }
```

Figure 17.5

Line 22 creates a `Listener` object and associates it with the `button1` component by calling the `addActionListener` method in the `button1` component:

```
22       button1.addActionListener(new Listener());  // add listener
                                   --------------
                                                 creates listener object
```

The `Listener` object is created from the inner class on lines 50 to 56. This class must implement the `ActionListener` interface. This interface contains only the `actionPerformed` method. Thus, only this method has to be implemented in the listener class. You can use whatever name you like for the listener class—it does not have to be `Listener`. The code in the `actionPerformed` method is whatever you want. In the example in Fig. 17.5b, it increments `count` and then sets the text in the `label1` component to this new count:

```
54          label1.setText("Count = " +  ++count);  // set label
```

17.5 Determining Which Component Triggers an Event

If a GUI has more than one component that can trigger an event, we can either create a listener object for each component or we can create a single listener object that handles all the events. The program in Fig. 17.6b does the latter. It has two buttons one labeled `Click to count` and one labeled `Click to count down`. Clicking the former increments the count displayed; clicking of the latter decrements the count displayed.

a)

b)

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4 class GUI4 extends JFrame
 5 {
 6    private Container contentPane;
 7    private JPanel panel1, panel2;
 8    private JButton button1, button2;
 9    private JLabel label1;
10    private int count;
11    private ActionListener listener;
12    //---------------------------------
13    public GUI4()
14    {
15       setTitle("GUI4");
16       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17
18       panel1 = new JPanel();    // create panel 1
19       panel1.setBackground(Color.GREEN);
20       panel1.setPreferredSize(new Dimension(300, 100));
21       listener = new Listener();
22       button1 = new JButton("Click to count");
23       button1.addActionListener(listener);
24       panel1.add(button1);
25       button2 = new JButton("Click to count down");
26       button2.addActionListener(listener);
27       panel1.add(button2);
28
29       panel2 = new JPanel();            // create panel 2
30       panel2.setBackground(Color.YELLOW);
31       panel2.setPreferredSize(new Dimension(100, 100));
32       count = 0;
33       label1 = new JLabel("Count = " + count);
34       panel2.add(label1);
35
36       // get content pane of frame
37       contentPane = getContentPane();
38
39       contentPane.setLayout(new FlowLayout());
40       contentPane.add(panel1);          // add panels to content pane
41       contentPane.add(panel2);
42
43       pack();
44       setVisible(true);
45    }
46    //---------------------------------
47    public static void main(String[] args)
48    {
49       GUI4 window = new GUI4();         // create window
50
51    }
52    //---------------------------------
53    private class Listener implements ActionListener
54    {
55       public void actionPerformed(ActionEvent e)
56       {
57          // e.getSource() returns ref to triggering component
58          if (e.getSource() == button1)
59             label1.setText("Count = " +  ++count);
60          else
61          if (e.getSource() == button2)
62             label1.setText("Count = " +  --count);
63       }
64    }
65 }
```

Figure 17.6

On line 21, the constructor creates a single listener object:

```
21          listener = new Listener();
```

It adds this listener object to `button1`:

```
23          button1.addActionListener(listener);
```

and to `button2`:

```
26          button2.addActionListener(listener);
```

Because `button1` and `button2` share the same listener, the code in this listener must determine which button triggered the event so it can take the appropriate action. If `button1` triggers the event, the listener should increment the count, but if `button2` triggers the event, the listener should decrement the count.

The component that triggers an event can be determined from the `ActionEvent` object passed to the `actionPerformed` method when the event occurs. The `getSource` method in this object returns the reference to the component that triggers the event. Lines 58 and 61 compare this reference with the `button1` and `button2` references to determine which button triggered the event:

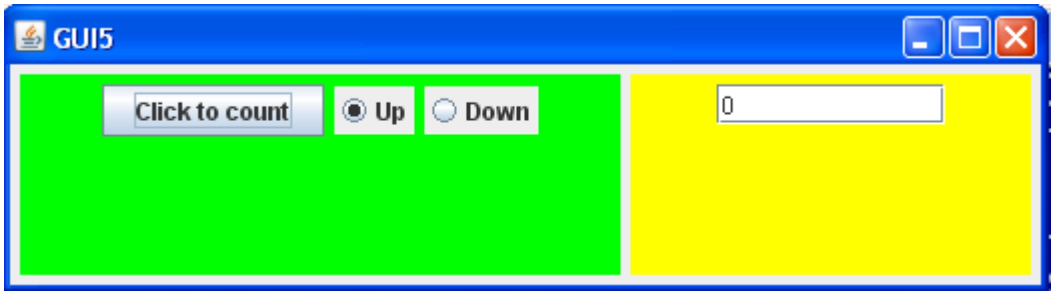
```
58          if (e.getSource() == button1)
59             label1.setText("Count = " +  ++count);
60          else
61          if (e.getSource() == button2)
62             label1.setText("Count = " +  --count);
```

17.6 USING RADIO BUTTONS

Radio buttons are organized into groups. Within each group, only one radio button can be selected at any given time. If the user clicks on an unselected button, it becomes the selected button, and the previously selected button is automatically unselected.

Like the program in Fig. 17.6b, the program in Fig. 17.7b can count up or down. However, unlike the program in Fig. 17.6b, it uses only one count button. Clicking on this button causes the count to increase if the **`Up`** radio button is selected, and decrease if the **`Down`** radio button is selected (see Fig. 17.7a). Another difference between the program in Fig. 17.6b and the program in Fig. 17.7b is the latter uses a text field object rather than a label object to display the count. The text field displays the current count. However, it can also be used to reset the count to any value. To do this, the user highlights the count with the mouse, enters the new count, and then hits the Enter key on the keyboard. Thus, a text field can be used to input data as well as display data. A label component, on the other hand, can be used only to display data.

a)



b)

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4 class GUI5 extends JFrame
 5 {
 6    private Container contentPane;
 7    private JPanel panel1, panel2;
 8    private JButton button;
 9    private JTextField text1;
10    private JRadioButton radio1;
11    private JRadioButton radio2;
12    private ButtonGroup group;
13    private ActionListener listener;
14    private int count;
15    //---------------------------------
16    public GUI5()
17    {
18       setTitle("GUI5");
19       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20
21       panel1 = new JPanel();           // create panel 1
22       panel1.setBackground(Color.GREEN);
23       panel1.setPreferredSize(new Dimension(300, 100));
24       listener = new Listener();
25       button = new JButton("Click to count");
26       button.addActionListener(listener);
27       panel1.add(button);
28
29       radio1 = new JRadioButton("Up", true);  // create radio button 1
30       radio2 = new JRadioButton("Down");      // create radio button 2
31       group = new ButtonGroup();      // create button group
32       group.add(radio1);              // add radio buttons to group
33       group.add(radio2);
34       panel1.add(radio1);             // add radio button 1 to panel 1
35       panel1.add(radio2);             // add radio button 2 to panel 2
36
37       panel2 = new JPanel();          // create panel 2
38       panel2.setBackground(Color.YELLOW);
39       panel2.setPreferredSize(new Dimension(200, 100));
40       text1 = new JTextField(10);
41       count = 0;
42       text1.setText(""+ count);
43       text1.addActionListener(listener);
44       panel2.add(text1);
45
46       // get content pane of frame
47       contentPane = getContentPane();
48       contentPane.setLayout(new FlowLayout());
49       contentPane.add(panel1);    // add panels to content pane
50       contentPane.add(panel2);
51
52       pack();
53       setVisible(true);
54    }
55    //---------------------------------
56    public static void main(String[] args)
57    {
58       GUI5 window = new GUI5();
59    }
60    //---------------------------------
61    private class Listener implements ActionListener
62    {
63       public void actionPerformed(ActionEvent e)
64       {
65          if (e.getSource() == text1)
66             count = Integer.parseInt(text1.getText());
67          else
68          if (e.getSource() == button)
69             if (radio1.isSelected())  // radio button 1 selected?
70                text1.setText("" + ++count);  // increment count
71             else
72             if (radio2.isSelected())  // radio button 2 selected?
73                text1.setText("" + --count);  // decrement count
74       }
75    }
76 }
```

Figure 17.7

To create the radio button group, the GUI5 constructor first creates the two radio buttons:

```
29       radio1 = new JRadioButton("Up", true);  // create radio button 1
30       radio2 = new JRadioButton("Down");      // create radio button 2
```

The `true` argument used on line 29 makes the `radio1` button the button that is initially selected. The `GUI5` constructor then creates a button group:

```
31       group = new ButtonGroup();      // create button group
```

It then adds the radio buttons to this group:

```
32       group.add(radio1);              // add radio buttons to group
33       group.add(radio2);
```

Finally, it adds the radio buttons to `panel1`:

```
34       panel1.add(radio1);             // add radio button 1 to panel 1
35       panel1.add(radio2);             // add radio button 2 to panel 2
```

Note that the individual radio buttons are added to the panel—not the button group.

The constructor creates the text field with

```
40       text1 = new JTextField(10);
```

The parameter 10 specifies the width of the text field. The initial count displayed in the text field is set with

```
41       count = 0;
42       text1.setText(""+ count);
```

The `setText` method requires a `String` parameter. The concatenation of `""` (the null string) with `count` (which is type `int`) yields the string required by `setText`.

If an event is triggered by the text field, line 66 in the `actionPerformed` method is executed:

```
66             count = Integer.parseInt(text1.getText());
```

The `getText` method returns the text in the text field as a string. The `parseInt` method then converts this string to type `int`, and assigns it to `count`. Thus, if the user then clicks on the count button, this new count will be incremented or decremented.

If the count button triggers the event, then the count is either incremented or decremented, depending on which radio button is selected:

```
69             if (radio1.isSelected())  // radio button 1 selected?
70                text1.setText("" + ++count);  // increment count
71             else
72             if (radio2.isSelected())  // radio button 2 selected?
73                text1.setText("" + --count);  // decrement count
```

17.7 LAYOUT MANAGERS

All the containers in the GUI programs we have seen use `FlowLayout` (`FlowLayout` is the default layout if a specific layout for a container is not specified). With this layout, components are added left to right in rows. If a component cannot fit on the current row, a new row is started below the current row.

A left/center/right justification of components can be specified when a `FlowLayout` is created. For example, in

```
c.setLayOut(new FlowLayout(FlowLayout.LEFT));
```

we are specifying `FlowLayout` with left justification for the container `c`. With left justification, any extra room in a row will appear on the right side of the row. For center and right justification, use `FlowLayout.CENTER` and `FlowLayout.RIGHT`, respectively. We can also specify the horizontal a vertical gaps between components. These gaps are passed as the second and third parameters to the `FlowLayout` constructor. For example, to specify center justification, a 20 pixel horizontal gap, and a 10 pixel vertical gap for the `c` container, use

```
c.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 10));
```

Remember that every container is controlled by a layout. Thus, if we place two panels on a frame, the two panels and the frame can each have their own layout.

Two other layouts available are `BorderLayout` and `GridLayout`. In `BorderLayOut`, the screen is divided into five regions: north (top), south (bottom), east (right), west (left), and center. Components are placed in one of these regions. For example, suppose we set `BorderLayout` for a container named `c` with

```
c.setLayout(new BorderLayout());
```

To place `button1`, `button2`, `button3`, `button4`, and `button5` in the north, south, east, west, and center regions, respectively, use

```
c.add(button1, BorderLayout.NORTH);
c.add(button2, BorderLayout.SOUTH);
c.add(button3, BorderLayout.EAST);
c.add(button4, BorderLayout.WEST);
c.add(button5, BorderLayout.CENTER);
```

In `GridLayout`, the screen is divided into rows and columns. For example, suppose we set `GridLayout` for the `c` container with

```
c.setLayout(new GridLayout(3, 2));
```

This creates a grid of 3 rows and 2 columns. Thus, each of the three rows can accommodate at most two components. To place `button1` and `button2` in row 1, `button3` and `button4` in row 2, and `button5` and `button6` in row 3, use

```
c.add(button1);
c.add(button2);
c.add(button3);
c.add(button4);
c.add(button5);
c.add(button6);
```

The components are added left to right in each row. Each row is filled before the next one is used.

17.8 APPLETS

All the programs we have seen so far are Java applications. A **Java application** is a standalone program written in Java that runs on a computer. An **applet**, on the other hand, is a Java program embedded in a web page. When a web browser views a web page that has an embedded applet, it executes that applet in addition to displaying that web page.

Because applets have the power and flexibility of programs, they can greatly enhance the capabilities of a web page. For example, you could embed a tax computation applet in a web page. Then anyone can use that program simply by visiting that page with a browser.

The power and flexibility of applets, however, presents a potential danger. An applet perhaps could erase your hard disk or make confidential information on your computer available to others. To avoid these problems, the capabilities of applets are intentionally restricted. For example, an applet cannot delete, read, or create files on the system on which it is running.

A web page is a text file that typically contains **hypertext markup language** (HTML). HTML contains **tags** that tell the browser how to display or handle what follows the tag. Most tags come in opening/closing pairs. For example, in the HMTL in Fig. 17.8a, `<I>` is an opening tag, and `</I>` is the corresponding closing tag. `<I>` tells the browser to display text in italic type. `</I>` tells the browser to revert back to the original type. `<BR>` is the break tag. It has no corresponding closing tag. It tells the browser to break the current line (i.e., go to the next line). The `<APPLET>` and `</APPLET>` tags embed an applet within a web page. **Attributes** within a tag provide specific information on that tag, and are specified with keywords such as `code`, `width`, and `height`. For example, the code attribute in the `<APPLET>` tag in Fig. 17.8a is

```
code="Applet1.class"
```

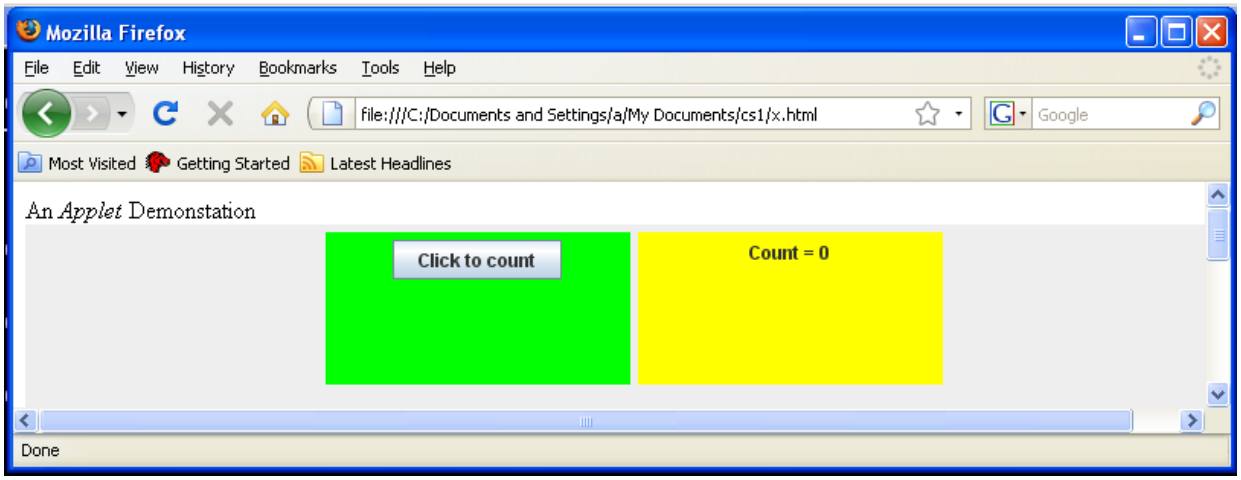
It specifies the name of the class file of the applet to be executed. The **width** and **height attributes** specify the size in pixels of the window in which the applet executes

a)

index.html

```
An <I> Applet </I> Demonstration
<BR>
<APPLET code="Applet1.class" width=800 height=300>
</APPLET>
```

b)



c)

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4 public class Applet1 extends JApplet // must be public, extend JApplet
 5 {
 6    private Container contentPane;
 7    private JPanel panel1, panel2;
 8    private JButton button1;
 9    private JLabel label1;
10    private int count;
11    //---------------------------------
12    public void init()     // use init method in place of constructor
13    {
14       panel1 = new JPanel();              // create panel 1
15       panel1.setLayout(new FlowLayout());
16       panel1.setBackground(Color.GREEN);
17       panel1.setPreferredSize(new Dimension(300, 50));
18       button1 = new JButton("Click to count");
19       button1.addActionListener(new Listener());
20       panel1.add(button1);
21
22       panel2 = new JPanel();              // create panel 2
23       panel2.setLayout(new FlowLayout());
24       panel2.setBackground(Color.YELLOW);
25       panel2.setPreferredSize(new Dimension(300, 50));
26       count = 0;
27       label1 = new JLabel("Count = " + count);
28       panel2.add(label1);
29
30       // get content pane of frame
31       contentPane = getContentPane();
32
33       contentPane.setLayout(new FlowLayout());
34       contentPane.add(panel1);      // add panels to content pane
35       contentPane.add(panel2);
37    }
38    //---------------------------------
39    private class Listener implements ActionListener
40    {
41       public void actionPerformed(ActionEvent e)
42       {
43          label1.setText("Count = " +  ++count);
44          System.out.println("Orange = " + Color.ORANGE);
45       }
46    }
47 }
```

Figure 17.8

Applets require a slightly different setup than Java applications. For example, Fig. 17.8c shows a Java applet that provides the same function as the GUI3 application in Fig. 17.5. It differs from the GUI3 application program in the following ways:

1) The class for the applet must be public (see line 4).

2) The class for the applet must extend `JApplet`, not `JFrame` (see line 4).

3) It does not have the calls of the `setTitle`, `setSize`, `setDefaultCloseOperation`, `pack`, and `setVisible` methods. These methods come from the `JFrame` class. But the `Applet1` class in Fig. 17.8c is not a subclass of `JFrame`. Thus, these methods are not available to the `Applet1` class. A Java application uses these methods to configure the window that it displays. However, an applet does not need these methods because it does not have its own window (an applet uses the browser's window).

4) Because the class of the applet is public, the base name of the file that contains it must match the class name. Thus, the file for the applet in Fig. 17.8c must be `Applet1.java`.

5) The `init` method that starts on line 12 in the applet takes on the function of the constructor in a Java application. The browser initiates the execution of the applet by calling the `init` method—not a `main` method. An applet does not contain a `main` method.

After compiling the applet in Fig. 17.8c, we can run it and see the web page in which it is embedded (see Fig. 17.8b) by opening the file in Fig. 17.8a with a Java-enabled web browser. Alternatively, we can enter

```
appletviewer index.html
```

where `index.html` is the name of the file in Fig. 17.8a. `appletviewer` displays only the applet—not the rest of the web page in which it is embedded. `appletviewer` is a utility program that comes with the Java compiler and interpreter.

To make the web page in Fig. 17.8 available on the World Wide Web, you must place it in the appropriate directory on your web server. Check with your system administrator for specific directions on how to do this.