Chapter 23

Graphical Applications

Now that we've covered many of the fundamentals of Java programming, we can consider the question of creating graphical programs. There are two kinds of graphical programs, applications that you can start at the command line or by clicking on an icon, and applets that can run within a browser. We'll start with applications and move to applets later.

Throughout this chapter, key ideas as labelled as **Graphics ideas**. There are a number of new techniques, and lots of little details, involved in creating graphical programs.

Graphics Idea 1 *Graphical applications are event-driven.*

There are two aspects to a graphical application, how it appears on the screen and how it responds to *events* such as mouse clicks and keyboard inputs. The structure of a graphical application is strikingly different than that of an ordinary application. In many ordinary applications, there is a main loop in which input is received and processed. Most graphical applications work differently and simply provide a collection of methods for handling different events as they occur. In effect, the user of an *event-driven* program provides the loop when he or she uses the mouse or keyboard to generate a sequence of events.

Graphics Idea 2 *A graphical application is based on a collection of components.*

Consider an application that creates and manages the window shown in Figure 23.1. The goal of this application is simple: to show how many times the button has been clicked. This program is available as an applet at

`http://www.cs.amherst.edu/lam/applets/Counter/counter.html`

The Java files are available in the directory

```
http://www.cs.amherst.edu/lam/applets/Counter
```

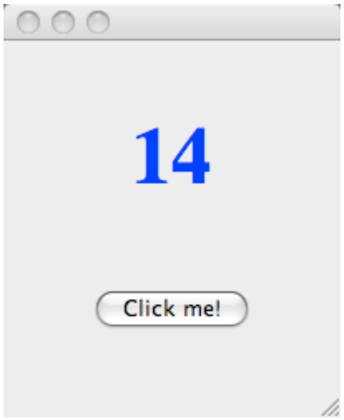

Figure 23.1: A simple graphical window

There are four graphical elements in the window, each of which is represented by a different kind of graphical component:

- A `JButton` represents the button.
- A `JLabel` represents the text that is displayed.
- A `JPanel` represents the overall "contents" of the window, i.e., the button, the text, and the space around them.
- A `JFrame` represents the entire window, including the close, minimize, and maximize buttons.

(Don't worry yet about the full distinction between the `JPanel` and the `JFrame`.) The application needs to handle several kinds of events: clicking of the four different buttons ("click me", minimize, maximize, and close) and resizing of the frame.

These four classes used here, `JButton`, `JLabel`, `JPanel`, and `JFrame`, are part of the Java Swing library. Swing works with the Java AWT (Abstract Window Toolkit) and provides a powerful mechanism for creating graphical programs. In order to access the Swing and AWT libraries, you should generally include these lines in the classes that you create:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
```

You should consult the API documentation extensively for details on the classes that we discuss in this chapter.

Figure 23.1 is, by the way, a screenshot from a Macintosh. On other systems there would be minor differences in the appearance of the frame, buttons, and fonts. One of the interesting features of Swing is the ability to change the look-and-feel so that the appearance of windows follows the rules of other systems.

Graphics Idea 3 *Most graphical programs are written by creating subclasses of classes in the Swing and AWT packages.*

Figure 23.2 displays the code for class `CounterPanel`, which is a subclass of `JPanel` and which does most of the work in the counter program.

Graphics Idea 4 *A* `JPanel` *is a graphical component that can contain other components.*

A `JPanel` can "contain" other components, in the sense that other components can be incorporated graphically and operationally into it. In Figure 23.2, a `JLabel` is created and added on line 16, and a `JButton` is created and added on line 23. (Recall that it's possible to do an assignment within an expression; that's how variables `label` and `button` are set.)

The subcomponents in a `JPanel` are arranged by a *layout manager*. Line 14 specifies that we want to use a `BoxLayout` in which the subcomponents appear in a single column. (If the second argument were `BoxLayout.X_AXIS`, it would be a single row.) Lines 19 and 25 ensure that the center lines of the two components are aligned (as opposed to their left or right edges). Line 21 places a 50-pixel gap between the subcomponents.

Lines 17 and 18 set the font size and color for the label, superseding the smaller black font that would otherwise be used.

Line 27 places a 50-pixel empty border on all four sides of the panel. Without this line, the panel would appear very cramped.

Graphics Idea 5 *A button click is an* `ActionEvent` *and is handled by an* `ActionListener`*.*

A class that implements `ActionListener` provides an `actionPerformed` method for processing `ActionEvents`. In Figure 23.2, class `CounterPanel` provides such a method.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class CounterPanel extends JPanel implements ActionListener {

    private int        counter = 0;
    private JLabel     label;
    private JButton    button;

    public CounterPanel() {

        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        add (label  = new JLabel(counter + ""));
        label.setFont(new Font ("Serif", Font.BOLD, 48));
        label.setForeground(Color.BLUE);
        label.setAlignmentX(0.5f);

        add (Box.createVerticalStrut(50));

        add (button = new JButton("Click me!"));
        button.addActionListener(this);
        button.setAlignmentX(0.5f);
        button.setFocusable(false);

        setBorder(new EmptyBorder(50, 50, 50, 50));
    }

    public void actionPerformed (ActionEvent e) {

        if (e.getSource().equals(button)) {
            counter++;
            label.setText(counter + "");
        }
    }
}
```

Figure 23.2: CounterPanel.java

```
1   import javax.swing.*;
2
3   public class CounterApp extends JFrame {
4
5       private CounterApp() {
6           super("Counter");
7           setDefaultCloseOperation(EXIT_ON_CLOSE);
8           setContentPane(new CounterPanel());
9           pack();
10          setVisible(true);
11      }
12
13      public static void main (String[] args) {
14          new CounterApp();
15      }
16  }
```

Figure 23.3: CounterApp.java

On line 24, the call `button.addActionListener(this)` specifies that `ActionEvents` generated from clicking `button` should be referred to the `CounterPanel` itself. The `actionPerformed` method checks that the source of the event was `button` and then increments `counter`, the count of clicks. It resets the text for `label`, which causes the window to be updated.

Graphics Idea 6 *A* `JFrame` *is top-level graphical component.*

Figure 23.3 shows the main class for the counter program. `CounterApp` is a subclass of `JFrame`, in other words, it is our version of the overall window. The main method simply creates a `CounterApp` object and exits. The constructor for `CounterApp` makes a series of calls to methods inherited from `JFrame`. The `super` call on line 6 invokes the constructor in `JFrame` and sets the title of the frame. The call to `setDefaultCloseOperation` ensures that clicking the close button will cause the program to terminate. The call to `setContentPane` specifies that the contents of the window will be a `CounterPanel`. The `pack()` call uses the specifications of the various graphical subcomponents and builds the frame and its contents. The `setVisible(true)` call ensures that the window is visible, not invisible. Each of these calls, except the one that sets the title, is critical and should appear whenever you create a subclass of `JFrame`.

The minimize and maximize buttons and resizing of the top-level window are all handled automatically by Swing. Resizing is a delicate issue that we'll discuss more soon.

In many cases, you can use the code in Figure 23.3 almost verbatim, changing only the names, in your `JFrame` class. Structuring your `JFrame` class in this way will make it easy to turn your application into an applet.

23.1 Threads

All of the non-graphical programs that we discussed in previous chapters used a single *thread of execution*. When execution begins, a main method is called and begins its work. Execution may involve branching, loops, and method calls, but there is always a single "current location" in the program.

Graphical programs are often organized differently. There are multiple threads of execution when the program is running. Essentially this means that there are multiple programs running simultaneously that share the same memory but handle different tasks.

The counter program involves two threads:

- The *main thread* is the one that starts first and begins by running the main method. It creates a `CounterApp` object and terminates. The creation of the first graphical object causes the other thread to start running.

- The *event thread* responds to events triggered by user actions, such a mouse clicks, mouse movements, and keystrokes. The event thread is also responsible for the display of components. It redisplays them as needed, for example, when the window is first created, when it is resized, or when text of a label changes.

The significance of the use of multiple threads will become more evident as we consider more complex graphical programs. In particular, animation requires the use of additional threads.

23.2 Drawing in a Graphics Window

Figure 23.4 shows the window that will be maintained by our second graphical program. Clicking the buttons will add and remove colored balls from the window. The balls will be stationary for now, but we'll add motion later.

An applet for this program is available on the web at:

`http://www.cs.amherst.edu/lam/applets/Balls/balls.html`

The Java files are available in the directory

`http://www.cs.amherst.edu/lam/applets/Balls`

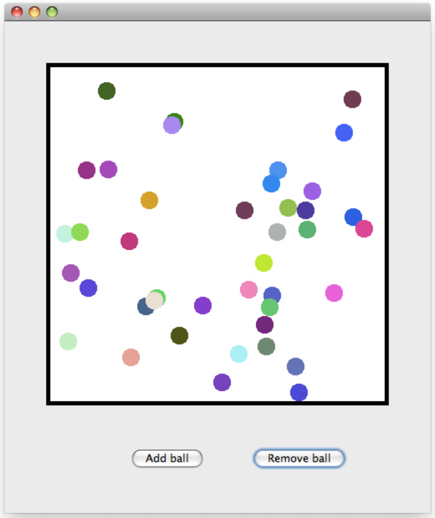

Figure 23.4: A graphical window containing colored balls.

Figure 23.5 shows the code for class `BallPanel`. A `BallPanel` object is associated with the entire contents of the window. Class `BallApp`, similar to `CounterApp` (Figure 23.3), creates a `BallPanel` object and calls `setContentPane` on it.

A `BallPanel` contains several subcomponents:

- A `BallCanvas`, the drawable area in which the balls appear. We'll discuss this subcomponent in detail.
- Two `JButtons`.
- A `ButtonPanel` that holds the two buttons. This subpanel is implemented via an inner class.

The `ButtonPanel` uses a `BoxLayout` in `Y_AXIS` mode to place the two buttons in a horizontal row. The main panel uses a `BoxLayout` in `X_AXIS` mode to place the `ButtonPanel` below the `BallCanvas`.

Graphics Idea 7 *In order to have a panel appear the way you want, it is sometimes useful to define one or more subpanels.*

Method `actionPerformed` runs when either button is clicked. This method calls the correct method in `BallCanvas`, either `addBall` or `removeBall`.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class BallPanel extends JPanel implements ActionListener {

    private BallCanvas canvas;
    private JButton addButton, removeButton;

    private class ButtonPanel extends JPanel {
        ButtonPanel() {
            setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
            addButton = new JButton("Add ball");
            addButton.setFocusable(false);
            add(addButton);
            add(Box.createHorizontalStrut(50));
            removeButton = new JButton("Remove ball");
            removeButton.setEnabled(false);
            removeButton.setFocusable(false);
            add(removeButton);
            setBorder(new EmptyBorder(50, 0, 0, 0));
        }
    }

    public BallPanel() {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        add(canvas = new BallCanvas(this));
        add(new ButtonPanel());
        addButton.addActionListener(this);
        removeButton.addActionListener(this);
        setBorder(new EmptyBorder(50,50,50,50));
    }

    public void actionPerformed (ActionEvent e) {
        if (e.getSource().equals(addButton))
            canvas.addBall();
        else
            canvas.removeBall();
    }

    public void enableRemove() {
        removeButton.setEnabled(true);
    }

    public void disableRemove() {
        removeButton.setEnabled(false);
    }
}
```

Figure 23.5: BallPanel.java

On lines 15 and 20 in Figure 23.5, we specify that the buttons in the ButtonPanel should not be *focusable*. *Keyboard focus* is a tricky topic that we'll discuss later. A button is normally focusable, meaning it might become the default button, the one that is highlit and that can be activated when the user types the enter key. In the ball example, the enter key will not be associated with either button.

Figure 23.6 shows the code for the class BallCanvas, which is responsible for maintaining and displaying the area inside the black square. BallCanvas is a subclass of JPanel.

On lines 21-23, size parameters are given for the component.

Graphics Idea 8 *The size and appearance of a* JPanel *(and of other graphical components) is determined in a complex way. Every component can have preferred size. The layout manager (e.g.,* BoxLayout*) attempts to place all the graphical elements so that they have their preferred sizes. If the overall window is resized by the user, the layout manager will run again and will redo the layout. Maximum and/or minimum sizes can also be given, setting limits on the extent of resizing. Each size is expressed via using a* Dimension *object, which incorporates a width and a height.*

In the constructor for BallCanvas, all three sizes are set to the same value, meaning that no resizing is possible. The border is considered part of the component, so setting the sizes to be SIZE + 2*BORDER_WIDTH leaves a drawable area with width and height SIZE. The constructor also sets a background color for the component and sets a border.

(It's important to avoid overusing maximum and minimum sizes, because the layout manager needs flexibility in order to do its work. Sometimes it is simply impossible to honor all size requests, for example if a large window of fixed size is placed inside a smaller window of fixed size. In that case, the smaller window would let users see only part of the larger window.)

BallCanvas uses a linked list of Ball objects to record the configuration that should be displayed in the window. Methods addBall and removeBall are called whenever the appropriate buttons are clicked and paintComponent is called whenever the component needs to be redrawn.

Graphics Idea 9 *Methods that respond to events should call method* repaint *when they want to request that a component be redrawn on the screen.*

Methods addBall and removeBall both work by first changing the list of balls and then calling repaint. They also call methods panel.enableRemove() and

```java
import javax.swing.*;
import java.awt.*;
import java.util.LinkedList;
import javax.swing.border.*;

public class BallCanvas extends JPanel {

    final private static int BALL_DIAMETER = 20;
    final private static int SIZE          = 400;
    final private static int BORDER_WIDTH  = 5;

    private BallPanel panel;
    private LinkedList<Ball> balls = new LinkedList<Ball>();

    public BallCanvas(BallPanel p) {

        panel = p;
        Ball.initialize(SIZE, BORDER_WIDTH);

        int fullSize = SIZE + 2*BORDER_WIDTH;
        setPreferredSize(new Dimension(fullSize, fullSize));
        setMaximumSize(new Dimension(fullSize, fullSize));
        setMinimumSize(new Dimension(fullSize, fullSize));

        setBackground(Color.WHITE);
        setBorder(new LineBorder(Color.BLACK, BORDER_WIDTH));
    }

    public void addBall() {
        balls.add(new Ball(BALL_DIAMETER));
        panel.enableRemove();
        repaint();
    }

    public void removeBall() {
        balls.removeLast();

        if (balls.size() == 0)
            panel.disableRemove();
        repaint();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (Ball b : balls)
            b.paint(g);
    }
}
```

Figure 23.6: BallCanvas.java

`panel.disableRemove()` so that the panel will know when to enable or disable the remove button.

Graphics Idea 10 *Method* `paintComponent` *is responsible for drawing on the screen.*

A subclass of `JPanel` can supply a `paintComponent` method to draw lines, shapes, and text in the graphics window. No `paintComponent` method is needed if the entire scene consists of a background color plus some collection of subcomponents.

If present, method `paintComponent` must be `public` and `void`, and it must take one parameter of type `Graphics`. The `Graphics` object, often named `g`, is used in the method calls that do drawing. The first line in `paintComponent` should always be

```
super.paintComponent(g);
```

This calls method `paintComponent` in `JFrame`, which paints the background.

In Figure 23.6, this required call appears on line 44. It is followed by a loop that makes calls requesting that each `Ball` object draw itself in the window.

Figure 23.7 shows the code for class `Ball`. Before examining the details of this code, it is useful to understand the coordinate system used for drawing.

Graphics Idea 11 *In any component, pixel locations are given by a pair* (x, y)*, with location* $(0, 0)$ *being the upper left corner. The* x *coordinate increases as one moves to the right, and the* y *coordinate increases as one moves down.*

The maximum x coordinate is one less than the width of the window, and the maximum y coordinate is one less than the height. To draw a line from point $(x0, y0)$ to point $(x1, y1)$, you can simply write

```
g.drawLine(x0, y0, x1, y1);
```

When a `Ball` is constructed, a diameter is specified. A random position $(x0, y0)$ is generated for the ball, with coordinates chosen in the range $[0, (arenaSize - diameter - 1)]$. In Java graphics, all ovals (of which a circle is a special case) are considered be inscribed within an imaginary rectangle. The upper left corner of the imaginary rectangle is considered to be the location of the oval, even though that location isn't even contained in the oval. By setting $(arenaSize - diameter - 1)$ as the upper limit on the coordinates, we ensure that each ball falls inside the desired area.

```java
import java.awt.*;

public class Ball {

    private static int arenaSize;
    private static int borderWidth;

    private int x0, y0;
    private Color color;
    private int diameter;

    public static void initialize(int a, int b) {
        arenaSize = a;
        borderWidth = b;
    }

    public Ball(int d) {
        diameter = d;
        x0 = (int)(Math.random() * (arenaSize - diameter));
        y0 = (int)(Math.random() * (arenaSize - diameter));
        color = new Color(rand255(), rand255(), rand255());
    }

    public void paint(Graphics g) {
        g.setColor(color);
        Point p = getPosition();

        g.fillOval(p.x+borderWidth, p.y+borderWidth, diameter, diameter);
    }

    private Point getPosition() {
        return new Point (x0, y0);
    }

    private int rand255() {
        return (int)(Math.random()*256);
    }
}
```

Figure 23.7: Ball.java

Each ball is assigned a random color that is created on line 21. The three arguments to the Color constructor are red, green, and blue intensities in the range $[0, 255]$. The code on line 19 uses random numbers for those intensities, and hence a random color is generated.

Method paint on line 24 is called from method paintComponent in BallCanvas. It sets the chosen color and draws the ball on the screen. (Note that the coordinates are adjusted by adding borderWidth to account for the fact that the border is drawn inside the component.) Class Point, used on line 26, is part of the graphics library.

23.3 The Relationship Between repaint and paintComponent

Suppose you have created a class for a graphical element, such as BallCanvas, that has a paintComponent method. Let's review what should happen in response to a button click or other event:

1. An event-handling method, such actionPerformed, begins running.

2. It, perhaps using other methods, updates data structures for the graphical component. (In the Ball example, the list of balls was updated.)

3. A call is made to repaint, requesting that the component be redrawn.

4. At some time in the near future, paintComponent runs and redisplays the component.

The distinction between repaint requests and actual painting is made because repaint requests can arise for many reasons. Suppose, for example, that one window is partially covering another and that the top window is closed. A repaint request would be generated automatically to repaint the part of the lower window that was exposed.

The possibility of a delay between a repaint request and the actual painting permits the Java graphics system to work more efficiently. If many events occur in quick succession, a single paintComponent call can be used to display all the updates.

Graphics Idea 12 *All methods that run in the event thread,* including paintComponent, *should do their work "quickly."*

Methods that run in the event thread should not do things that might cause indefinite delays. For example, they should not pause (via a sleep or wait operation), read from the keyboard, or try to make a connection to another computer.