Chapter 14 Animation and Threads

14.1 Flicker free screen updates

Not clearing the screen

Clearing the screen to the background color causes much of the flicker seen in most Java programs. The screen is cleared by a method called *update*. The default *update* method clears the screen and then calls *paint*. To avoid clearing the screen, we replace the *default* update method with one that just calls *paint* (without clearing the screen).

The following program expands an oval on the screen.

```
public class growingDisk extends EventPanel
    {
    private int size;

    public growingDisk()
        {
        size = 5;
        }

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

    public void paint(Graphics g)
        {
        g.setColor(Color.red);
        g.fillOval(200-size/2, 150-size/3, size, size*2/3);
        g.setColor(Color.black);
        g.drawString("Even clicking very fast produces no flicker", 50, 270);
        }

    public void mousePressed(MouseEvent e)
        {
        if (size < 300)
            {
            size += 3;
            repaint();
            }
        }
    }
```

As long as each new image covers up the last, this program does 'the right thing'. However, if a new image uncovers portions of a previous image, this simple technique is not enough. Not erasing the screen means that previous images are not erased, and if they are not covered over they persist indefinitely.

Double Buffering

A straight forward way of removing old images without creating flicker is to assemble each new image off screen. The assembled image is then placed over the image on the screen — completely replacing it.

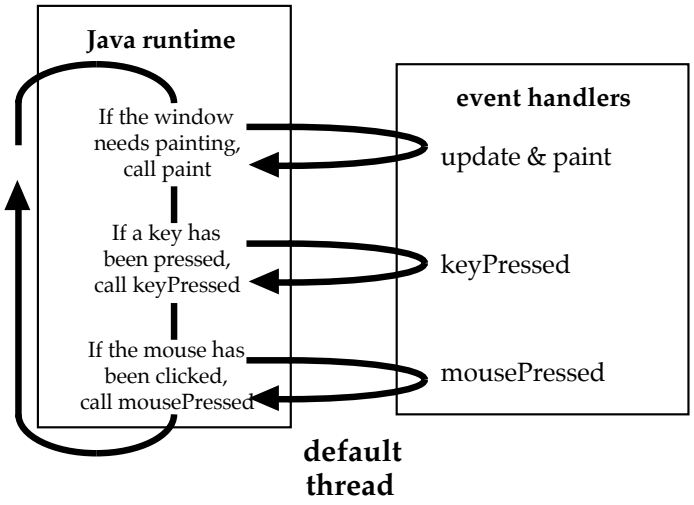
A portion of computer memory in which an image is stored is often called an *image buffer*. The offscreen image we use in this section is a second buffer (the onscreen image being the first). The term *double buffering,* has come to mean the assembly of a complete image off screen so the individual steps in the assembly are not seen.

The following program places an expandable hollow rectangle on the screen.

```
class GrowingFrame extends EventPanel
    {
    private int height;
    private Image offScreen;
    private Graphics osg;

    public GrowingFrame()
        {
        height = 16;
        }

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

    public void paint(Graphics g)
        {
        if (offScreen == null)
            {
            offScreen = createImage(getSize().width, getSize().height);
            osg = offScreen.getGraphics();
            }

        osg.setColor(getBackground());
        osg.fillRect(0, 0, size().width, size().height);

        osg.setColor(Color.red);
        osg.fillRect(200-height/4, 150-height/2, height/2, height/16);
        osg.fillRect(200-height/4, 150-height/2, height/16, height);
        osg.fillRect(200+height/4-height/16, 150-height/2, height/16, height);
        osg.fillRect(200-height/4, 150+height/2-height/16, height/2, height/16);

        osg.setColor(Color.black);
        osg.drawString("Even clicking very fast produces no flicker", 50, 270);
        g.drawImage(offScreen, 0, 0, null);
        }

    public void mousePressed(MouseEvent e)
        {
        if (height < 300)
            {
            height += 4;
            repaint();
            }
        }
    }
```

14.2 Threads and Animation

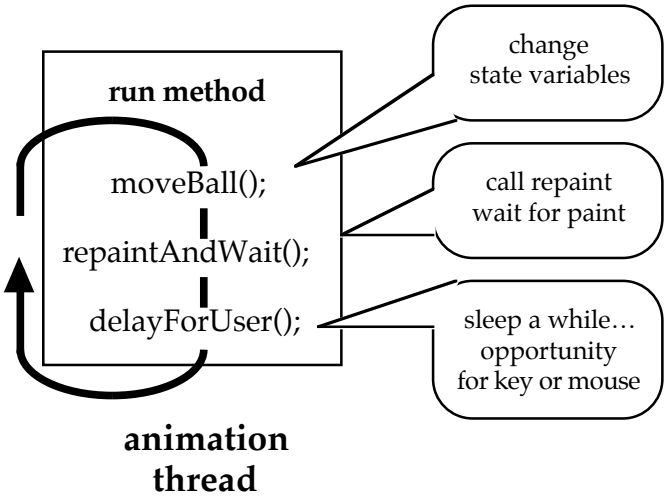
Interactive Java programs such as we have written contain several public event handling methods (*paint*, *keyPressed*, *mousePressed*). They perform their duties as quickly as possible and then quit, allowing the Java runtime system that runs these methods to run another such method.



The runtime system waits for one method to finish before starting another. The fact that each method is always allowed to finish before another is started is sometimes stated as 'the program has only one thread.' This is understood to mean that the system that runs the program will keep track of just one position among the program's instructions.

To create an animation with which the user can interact, we need an animating method that can pause for a while, let one or more of the event handling methods run and then continue. The standard way to run an animating method is to explicitly create a second thread, which runs the animating method. Creating a second thread explicitly requests that the system keep track of a second position among the programs instructions. In our case, the second thread will maintain a position within the animating method (and private methods it uses). Meanwhile, the default thread, which has existed in all of our programs, maintains a position within event handling methods (and any private methods they use).

The animating method changes the state variables, calls *repaint*, waits for the painting method, waits a bit more (to be sure the user has a chance to interact with the program), changes the state variables again, waits for the painting method again..



Threads

A Java thread always starts by executing the method *run* belonging to some object. In many programs, we don't want to create a second object, so we give the main program a run method. The thread we create will execute the *run* method while the other public methods (*update*, *paint*, *keyPressed*, *mousePressed*) will be executed by the default thread that was provided with our Java window.

The mechanics of thread use require four steps:

```
class AnimatedPanel extends EventPanel implements Runnable
                            // the first line of the class containing the run method advertises
                            // that it contains a run method — implements Runnable

private Thread runner;      // create a reference to a Thread object

runner = new Thread(this); // create a Thread object, telling it which object contains the run
                            // method it should run. In this case, it is the object containing
                            // this instruction.

public void run()           // put a run method into the program (for the thread to run).
```

Synchronization: avoiding interference between threads

When one thread is making changes to state variables that are available to another thread, it is appropriate to ensure that all changes are done as a unit. It would likely create problems if the second thread examined some of the state variables before they were changed and others after. To allow this kind of control, Java allows methods to be declared *synchronized*. If several methods are declared *synchronized*, a thread starting any one of them locks all other threads out from all such methods (until that method is finished). *Synchronized* is used to create a set of methods which can only be used by one thread at a time.

Having a thread wait

If a thread has nothing useful to do for a while, there are two appropriate courses of action. One is to relinquish rights to computer use for a specified period of time. The second is to relinquish computer time until another thread has completed some task. The other thread would give notification when it is finished.

To wait for a specified period of time, you call the method *sleep*. Calling *sleep* is complicated by the fact that the *sleep* method is a part of the *Thread* class, not part of a class you are writing. Secondly, *sleep* may be interrupted before the time is up and this would create an exception, which the program must be prepared to catch.

```
try { Thread.currentThread().sleep(100); }
catch (InterruptedException exp) { }
```

The two lines above put the current thread asleep for 100/1000 or 1/10 of a second.

To wait for a notification by another thread, you call the method *wait*. Like *sleep*, *wait* may be interrupted before the time is up and this would create an exception, which you must be prepared to catch. Also, the *wait* method must be called in a *synchronized* method. Third, while you can wait indefinitely, you should put a time limit on your wait, check explicitly whether or not there is anything else for you to do and go back to waiting if appropriate.

```
private synchronized waitForSomethingToDo()
    {
    while (!somethingToDo)
        try {wait(100); }
        catch (InterruptedException exp) { }
    }
```

The method above has the current thread wait for a maximum of 1/10 of a second, checks to see if there is something to do, waits some more if appropriate…

Notifying a waiting thread

To notify another thread, you call the method *notifyAll*. The *notifyAll* method must be called in a *synchronized* method. You should not only call *notifyAll*, you should also change some state variable to indicate that the thread you are notifying has something to do.

```
private synchronized notifyOtherThreads()
    {
    somethingToDo = true;
    notifyAll();
    }
```

14.3 A threaded animation

The following program animates a disk, lets the user move the disk with the mouse (clicking with the mouse makes the disk jump to the mouse), and lets the user change the color of the disk with the keyboard (r–red, b–blue, g–green).

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

public class BouncingDisk extends EventPanel implements Runnable
    {
    private int x, y, dx, dy;      // position and speed vectors for the disk
    private Color diskColor;       // color of the disk

    private Image offScreen;       // off screen drawing area
    private Graphics osg;          // graphics for use in drawing on the off screen area
    private int width, height;

    private Thread runner;         // the second thread
    private boolean busyPainting; // should the second thread be waiting for painting?

    public BouncingDisk()
        {
        x = 100; y = 100; dx = 3; dy = -3;
        diskColor = Color.red;

        busyPainting = true;
        runner = new Thread(this); // the thread will use the run method in this class
        runner.start();            // get the thread going
        }

    public void update(Graphics g)
        {
        paint(g);
        }
```

```
public synchronized void paint(Graphics g)
   {
   if (offScreen == null)        // before painting the first time, create the off screen image
      {
      width = getSize().width;
      height = getSize().height;
      offScreen = createImage(width, height);
      osg = offScreen.getGraphics();
      }

   osg.setColor(Color.lightGray);       // paint the background
   osg.fillRect(0, 0, width, height);

   osg.setColor(diskColor);             // paint the disk
   osg.fillOval(x-10, y-10, 20, 20);

   g.drawImage(offScreen, 0, 0, null); // copy the off screen image to the screen
   notifyRunner();                     // let the runner know we're finished
   }
public void run()                      // this is the method that the second thread executes
   {
   while (busyPainting)                // wait for initial paint
      {
      waitForPainting();
      }
   while (true)                        // loops indefinitely — until true becomes false!
      {
      moveBall();
      repaintAndWait();
      delayForUser();
      }
   }
private synchronized void moveBall()
   {
   if (x > width - 10) dx = -3;        // hit the east wall, bounce off
   if (x < 10) dx = 3;                 // hit the west wall, bounce off
   if (y > height - 10) dy = -3;       // hit the south wall, bounce off
   if (y < 10) dy = 3;                 // hit the north wall, bounce off
   x += dx; y += dy;                   // move the disk
   }
public synchronized void mousePressed(MouseEvent e)
   {
   x = e.getX();                       // move the disk
   y = e.getY();
   }
public synchronized void keyPressed(KeyEvent e)
   {
   char key = e.getKeyChar();
   if (key == 'r' || key == 'R')       // change color
      diskColor = Color.red;
   else if (key == 'b' || key == 'B')
      diskColor = Color.blue;
   else if (key == 'g' || key == 'G')
      diskColor = Color.green.darker();
   }
private synchronized void notifyRunner() // painting done, notify the waiting runner
   {
   busyPainting = false;
   notifyAll();
   }
```

```
private synchronized void repaintAndWait() // repaint, wait until painting is finished
    {
    busyPainting = true;
    repaint();
    while (busyPainting)                    // this is highly recommended paranoia
        {                                   // wait a while, then check, then wait…
        try {wait(100);}                    // waiting 100/1000 of a second
        catch (InterruptedException exp) { }
        }
    }
private synchronized void waitForPainting()  // wait until painting is finished
    {
    while (busyPainting)                    // this is highly recommended paranoia
        {                                   // wait a while, then check, then wait…
        try {wait(100);}                    // waiting 100/1000 of a second
        catch (InterruptedException exp) { }
        }
    }
private void delayForUser()                 // sleep for 5/1000 of a second,
    {                                       // let keyPressed or mousePressed have a chance.
    try { Thread.currentThread().sleep(5); }
    catch (InterruptedException exp) { }
    }
}
```