

15. Making the Game

The class *fly* will bring all of our other classes together to actually build the interactive game. It will need to extend *Applet*, so it will also import *Applet*. It will need to control the window, so all Abstract Window Toolkit (*awt*) methods will be imported. In addition, the game will be interactive, so all event classes will be imported.

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

Class *fly* will not only extend *Applet* in order to make an applet, it will implement a Java interface called *Runnable*:

```
public final class fly extends Applet implements Runnable {
```

Java interfaces impose certain methods on a program as well as an order of calls to these methods. In general, interface methods must be overridden in the class that implements an interface.

The interface *Runnable* is an interface that uses Java threads, which supports, among other things, interactive use of a Java applet. Java threads allow events in a Java applet to be synchronized so that they happen in a set order or to exist independently and compete for processor resources according to a priority much in the way that multiple processes compete for resources in a modern operating system.

The *Runnable* interface imposes the methods *start*, *run* and *stop*. Method *start* is used called before *run*, then *run* is called and finally, when *run* ends, *stop* is called. In class *fly*, this is complicated by *Applet*, which will cause method *init* (if it exists) to be called first when the applet starts, followed by a call to method *paint*. When *Runnable* and *Applet* are combined, *Applet's* logic is performed first, then *Runnable's* logic is performed.

The overall operation of class *fly* is further complicated by the fact that method *run* cannot directly call *paint* to update the applet to produce the animation desired. This is because *run* does not have access to the *Graphics* object that is part of using *Applet*. *Runnable* works around this by providing allowing a call to a method named *repaint*, which will call the method *update*, which has as its parameter the *Graphics* object from *paint*. Method *repaint* is not to be overridden by the programmer, but method *update* must be overridden in order to include a call to function *paint*. Here is a typical example:

```
public void update(Graphics g) {
    paint(g);
}
public void run() {
    while (true) {
        repaint();
        pause(50);
    }
}
```

(Notice the call to method pause. It is the same pause as found in the previously used version of class driver.)

Class *fly* introduces a Java feature that is new with Java 2. This is an internal class. Internal classes are classes that are defined inside of another class. Internal classes have access to all methods and variables of the enclosing class and are basically local in scope to the enclosing class.

In *fly*, buttons will be used to create events when the user makes a selection during the game. There is an internal class called *myButtonHandler* that implements *ActionListener*, which comes from the importing of *event*. Class *myButtonHandler* exists to handle any button selections that the user makes. The basic outline of an *ActionListener* class looks like this:

```
public class class-name implements ActionListener {
    public void actionPerformed(ActionEvent action-var) {
        if (button-name == action-var.getSource()) {
            *** Java statements ***
        } [else if ( *** etc. ***)]
    }
}
```

In class *fly*, there will be three buttons. These are *leftButton* (for a left turn), *rightButton* (for a right turn) and *start*, which will allow the user to look over the window before beginning the game. Here is the internal class *myButtonListener* for class *fly*. (Note: *p* is a Bug object and will be under direct control of the user.)

```
public class myButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (leftButton == e.getSource()) {
            p.turnleft(); p.turnleft();
        } else if (rightButton == e.getSource()) {
            p.turnright(); p.turnright();
        } else if (startButton == e.getSource()) {
            startPressed = true;
        }
    }
}
```

Instance Variables of Class *fly*

Variables that define the width and length of the game area are needed to avoid having to use literal constants throughout the program:

```
private int w = 500;
private int l = 400;
```

Variables that define the number of *Bug* objects, *evilBug* objects and *BugZapper* objects are needed. Note the use of method *random* to vary these numbers.

```
private int maxbugs = (int)(Math.random()*30+10);
private int maxevilbugs = (int)(Math.random()*10);
private int maxbug_zappers = (int)(Math.random()*10);
```

A background object is needed, plus arrays to contain the *BugZapper*, *Bug* and *evilBug* objects. Placing these objects in arrays will assist in limiting the number of repetitious lines of code in the program. (*For example, determining if there has been a collision between any of the bugs and bug zappers becomes a few lines of code in a for loop or nested for loops.*)

```
private Background b = new Background(w, l);
private BugZapper rarray[ ];
private Bug parray[ ];
private evilBug earray[ ];
```

Since winning the game depends on all regular bugs and evil bugs being defunct, it is necessary to have variables to keep count of the number of defunct bugs and evil bugs:

```
private int deadbugs = 0;
private int deadevilbugs = 0;
```

An object of class *Bug* that will be under the direct control of the user must be created. Notice the use of the method *random* to vary the starting location of the user's bug.

```
private Bug p = new Bug(w,l,w/4+(int)((w/2)*Math.random()),l-20);
```

The buttons discussed earlier must be created as objects of class *Button*. In addition, a Boolean variable is needed to serve as a flag to start the game when the *start* button is selected.

```
private Button leftButton, rightButton, startButton;
private boolean startPressed = false;
```

Since the interface *Runnable* is implemented in class *fly*, there must be a *Thread* variable.

```
private Thread animate;
```

To help eliminate video flicker, all graphical manipulation will take place in an *Image* class buffer that is separate from the actual video buffer. When the programmer-defined buffer has been completely updated, the contents of that buffer will be copied to the video buffer in one operation. This will help reduce the jerky quality of video that often results from drawing many objects directly to the video buffer. The *Image* class buffer is created as an instance variable:

```
private Image myBuffer;
```

Method *init*

When class *fly* is run, the first method called is *init*, which is responsible for initializing the previously created instance variables.

```
public void init() {
```

The thread variable must be constructed. Notice the call to *this*. Since Java programs are classes there must be a way of referring to the object created when a class is run. The run-time object of class *fly* is referred to as *this*.

```
    animate = new Thread(this);
```

The Image buffer must be constructed:

```
    myBuffer = createImage(w, l);
```

The array of *Bug* objects must be constructed. In addition, each element must be constructed, have its color set and its initial direction set. Notice the use of the method *random* to create positional, directional and color variety. All of this is greatly simplified because the objects are stored in an array.

```
    parray = new Bug[maxbugs];
    for (int x=0; x<maxbugs; x++) {
        parray[x] = new Bug( w,l, (int)(w*Math.random( )),
                           (int)(l*Math.random( )) );
        parray[x].setBugColor( (int)(100*Math.random( )+50),
                               (int)(200*Math.random( )+50),
                               (int)(200*Math.random( )+50) );
        for(int y=0; y<((int)(16*Math.random( ))); y++)
            parray[x].turnright( );
    }
}
```

An array of *evilBug* objects must be constructed. In addition, each element must be constructed, have its color set and its initial direction set. Notice once again the use of the method *random* to create positional and directional variety. Notice also that all evil bugs are set to the color white and that all are given the user controlled object *p* as a target. All of this is greatly simplified because the objects are stored in an array.

```
earray = new evilBug[maxevilbugs];
for (int x=0; x<maxevilbugs; x++) {
    earray[x] = new evilBug( w,l, (int)(w*Math.random( )),
                          (int)(l*Math.random( )), p);
    earray[x].setBugColor(255,255,255);
    for(int y=0; y<((int)(16*Math.random( ))); y++)
        earray[x].turnright( );
}
```

An array of *BugZapper* objects must be constructed. In addition, each element must be constructed, plus have its color set and its scale set. Notice the use of the method *random* yet again to create positional and scale variety. Once again, all of this is greatly simplified because the objects are stored in an array.

```
rarray = new BugZapper[maxbug_zappers];
for (int y=0; y<maxbug_zappers; y++)
    rarray[y] = new BugZapper((int)(w*Math.random( )),
                             (int)(l*Math.random( )),
                             (int)(50*Math.random( )+20));
```

The user-controlled bug must have its color set. Notice that this is red. Since the evil bugs are white and the random colors of the regular bugs are skewed away from white and red, red should stand out.

```
p.setBugColor(255,0,0);
```

Each button must be constructed with its label (the string in the constructor call). In addition, each button must be linked to class *myButtonHandler* via a call to *addActionListener*. Finally, each button must be added to the graphics window with a call to method *add*.

```
startButton = new Button("Start");
startButton.addActionListener(new myButtonHandler( ));
add(startButton);

leftButton = new Button("Turn Left");
leftButton.addActionListener(new myButtonHandler( ));
add(leftButton);
```

```

rightButton = new Button("Turn Right");
rightButton.addActionListener(new myButtonHandler( ));
add(rightButton);

} // *** end of init ***

```

Method *paint* and all of its Painting Methods

After the call to *init*, the next method called is *paint*, which is synchronized with the various move-the-bugs methods to insure that the thread logic cycle calling the move methods does not get ahead of the *paint* method. The thread logic cycle has not yet started upon the first call to *paint*.

```

public synchronized void paint(Graphics g) {

```

In *paint*, a *Graphics* object is needed that is not the graphics object *g*. *Graphics* object *g* represents the video buffer, which we do not wish to draw in directly. The *Graphics* object *hiddenG* is created and linked to the *Image* buffer *myBuffer* via a constructor like call to the *Image* class method *getGraphics*. All calls to all of the various *paint* methods are made with *hiddenG* rather than using *g*, thus all painting will be done to *myBuffer*, not to the video buffer. The video buffer will receive a completed copy of the image.

```

Graphics hiddenG = myBuffer.getGraphics( );
b.paint(hiddenG);
paintBugZappers(hiddenG);
paintBugArray(hiddenG);
paintEvilBugArray(hiddenG);
p.paint(hiddenG);

```

As the game progresses, the user needs to be informed as to how many *Bug* and *evilBug* objects are extant. Here, the *Graphics* method *drawString* is used to output a string to a specific (x,y) pixel coordinate pair giving that information. (Notice that the *Integer* method *toString* has to be used to convert each occurrence of an integer result of an integer expression to a string.)

```

hiddenG.drawString(Integer.toString(maxbugs-deadbugs)
+ " Bugs, "
+ Integer.toString(maxevilbugs-deadevilbugs)
+ " Evil Bugs, "
+ Integer.toString(maxbug_zappers)
+ "BugZappers",1/2-40,40);

```

Finally, the contents of *myBuffer* need to be copied into the video buffer object *g*. This is performed via a call to the *Graphics* method *drawImage*, with *myBuffer* as the first parameter, the pair 0,0 as the second and third parameters (to indicate that drawing is to start from the 0,0 pixel location of the video buffer) and a reference to *this* as the fourth parameter.

```
g.drawImage(myBuffer,0,0,this);

} // *** end of paint ***
```

The methods that paint the *BugZapper*, *Bug* and *evilBug* objects are very simple, consisting of for loops to insure that all objects are called. Notice that method *isAlive* is called before any *Bug* or *evilBug* object is painted to insure that only bugs that have not collided with other bugs or bug zappers are drawn.

```
private void paintBugZappers(Graphics g){
    for (int x=0; x<maxbug_zappers; x++) {
        rarray[x].paint(g);
    }
}

private void paintBugArray(Graphics g){
    for (int x=0; x<maxbugs; x++)
        if (parray[x].isAlive( )) parray[x].paint(g);
}

private void paintEvilBugArray(Graphics g){
    for (int x=0; x<maxevilbugs; x++)
        if (earray[x].isAlive( )) earray[x].paint(g);
}
```

After the first call to method *paint*, the *Runnable* interface logic takes over. First *start* is called, which activates the only *Thread* object in class *fly* with the call *animate.start()*.

```
public void start( ){
    animate.start( );
}
```

After method *start* is called, method *run* is called. When method *run* terminates, method *stop* is called. Because of Java's garbage collection system for collecting all unused objects and returning memory to the heap, there isn't anything for *stop* to do!

```
public void stop( ){;}
```

In between calls to *start* and *stop*, method *run* is called. In a class using a *Runnable* interface, method *run* is the heart of the logic of the program. Notice that in class *fly*, *run* consists of an infinite loop that will be exited when a game over condition is reached.

```
public void run(){
    out:while(true) {
```

The expression *out:* in front of the while loop is a *labeled goto*. When a *break* to this label (written *break out;*) is executed, the loop is exited. It looks as if the break will cause the while loop to begin again, but that is not the case. Control will move to the next line of code after the loop, which in this case is the end of the *run* method.

Here is the code that is found at the very end of the while loop which is used to test for an end-of-game condition. If all of the objects in the Bug and evilBug arrays are dead, the game will end (and the user's bug is the winner). If the users bug has impacted with any other bug or bug zapper, the game is over and the user loses.

Again, notice the call to the pause method just before the loop repeats. This pause amount should be set to regulate the relative speed of the game.

```
        :
        :
        :
        if (maxbugs <= deadbugs & maxevilbugs <=
            deadevilbugs) break out;
        for (int x=0; x<maxbugs; x++)
            if (parray[x].isAlive() &
                p.impactBug(parray[x])) break out;
        for (int x=0; x<maxevilbugs; x++)
            if (earray[x].isAlive() &
                earray[x].impactBug(p)) break out;
        for (int y=0; y<maxbug_zappers; y++)
            if (rarray[y].impactBug(p)) break out;

        pause(50);
    } // *** end of if-start-pressed logic ***
} // *** end of while-true loop ***
} // *** end of run ***
```



```

// Check for evilBug-to-evilBug collisions
for (int a=0; a<maxevilbugs-1; a++)
    for (int b=a+1; b<maxevilbugs; b++)
        if ( earray[a].isAlive( ) &
            earray[b].isAlive( ) &
            earray[b].impactBug(earray[a])){
                earray[a].isDead( );
                earray[b].isDead( );
                deadevilbugs++;
                deadevilbugs++;
            }
// Check for Bug-to-BugZapper collisions
for (int a=0; a<maxbugs; a++)
    for (int b=0; b<maxbug_zappers; b++)
        if ( parray[a].isAlive( ) &
            rarray[b].impactBug(parray[a])){
                parray[a].isDead( );
                deadbugs++;
            }
// Check for evilBug-to-BugZapper collisions
for (int a=0; a<maxevilbugs; a++)
    for (int b=0; b<maxbug_zappers; b++)
        if ( earray[a].isAlive( ) &
            rarray[b].impactBug(earray[a])){
                earray[a].isDead( );
                deadevilbugs++;
            }
}

```

Complete Listing of Class fly

```

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public final class fly extends Applet implements Runnable{

    private int maxbugs = (int)(Math.random( )*30+10);
    private int maxevilbugs = (int)(Math.random( )*10);
    private int maxbug_zappers = (int)(Math.random( )*10);
    private int w = 500;
    private int l = 400;

    private Thread animate;
    private Image myBuffer;

```

```

private Background b = new Background(w, l);
private BugZapper rarray[ ];
private Bug parray[ ];
private evilBug earray[ ];
private Bug p = new Bug(w,l,w/4+(int)((w/2)*Math.random()),l-20);

private int deadbugs = 0;
private int deadevilbugs = 0;

private Button leftButton, rightButton, startButton;
private boolean startPressed = false;

public void init( ) {
    animate = new Thread(this);
    myBuffer = createImage(w,l);

    // setup bug array
    parray = new Bug[maxbugs];
    for (int x=0; x<maxbugs; x++) {
        parray[x] = new Bug( w,l, (int)(w*Math.random( )),
            (int)(l*Math.random( )) );
        parray[x].setBugColor( (int)(100*Math.random( )+50),
            (int)(200*Math.random( )+50),
            (int)(200*Math.random( )+50) );
        for(int y=0; y<((int)(16*Math.random( ))); y++)
            parray[x].turnright( );
    }

    // setup evilbug array
    earray = new evilBug[maxevilbugs];
    for (int x=0; x<maxevilbugs; x++) {
        earray[x] = new evilBug( w,l, (int)(w*Math.random( )),
            (int)(l*Math.random( )), p);
        earray[x].setBugColor(255,255,255);
        for(int y=0; y<((int)(16*Math.random( ))); y++)
            earray[x].turnright( );
    }

    // set up bug zapper array
    rarray = new BugZapper[maxbug_zappers];
    for (int y=0; y<maxbug_zappers; y++)
        rarray[y] = new BugZapper((int)(w*Math.random( )),
            (int)(l*Math.random( )),
            (int)(50*Math.random( )+20));

```

```
// set up bug
p.setBugColor(255,0,0);

// set up directional buttons
startButton = new Button("Start");
startButton.addActionListener(new myButtonHandler( ));
add(startButton);

leftButton = new Button("Turn Left");
leftButton.addActionListener(new myButtonHandler( ));
add(leftButton);

rightButton = new Button("Turn Right");
rightButton.addActionListener(new myButtonHandler( ));
add(rightButton);
}

public synchronized void paint(Graphics g) {

    Graphics hiddenG = myBuffer.getGraphics( );
    b.paint(hiddenG);
    paintBugZappers(hiddenG);
    paintBugArray(hiddenG);
    paintEvilBugArray(hiddenG);
    p.paint(hiddenG);
    hiddenG.drawString(Integer.toString(maxbugs-deadbugs)
        + " Bugs, "
        + Integer.toString(maxevilbugs-deadevilbugs)
        + " Evil Bugs, "
        + Integer.toString(maxbug_zappers) + "
        BugZappers",1/2-40,40);
    g.drawImage(myBuffer,0,0,this);
}

private void paintBugZappers(Graphics g){
    for (int x=0; x<maxbug_zappers; x++) {
        rarray[x].paint(g);
    }
}
```

```
private synchronized void moveBugArray( ){
    int d;
    for (int x=0; x<maxbugs; x++) {
        d = (int)(Math.random()*5);
        if (d < 1) parray[x].turnright( );
        else if (d >3) parray[x].turnleft( );
        parray[x].go( );
    }
}

private void paintBugArray(Graphics g){
    for (int x=0; x<maxbugs; x++)
        if (parray[x].isAlive( )) parray[x].paint(g);
}

private synchronized void moveEvilBugArray(){
    int d;
    for (int x=0; x<maxevilbugs; x++) {
        earray[x].calcDirection( );
        earray[x].go( );
    }
}

private void paintEvilBugArray(Graphics g){
    for (int x=0; x<maxevilbugs; x++)
        if (earray[x].isAlive( )) earray[x].paint(g);
}

public void start( ){
    animate.start( );
}

public void stop( ){;}

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

```
public void run() {
    out:while(true) {
        if (startPressed) {
            moveBugArray();
            moveEvilBugArray();
            p.go();
            repaint();

            // Check for Bug-to-Bug collisions
            for (int a=0; a<maxbugs-1; a++)
                for (int b=a+1; b<maxbugs; b++)
                    if ( parray[a].isAlive() &
                        parray[b].isAlive() &
                        parray[a].impactBug(parray[b])){
                        parray[a].isDead();
                        parray[b].isDead();
                        deadbugs++;
                        deadbugs++;
                    }
            // Check for Bug-to-evilBug collisions
            for (int a=0; a<maxbugs-1; a++)
                for (int b=0; b<maxevilbugs; b++)
                    if ( parray[a].isAlive() &
                        earray[b].isAlive() &
                        earray[b].impactBug(parray[a])){
                        parray[a].isDead();
                        earray[b].isDead();
                        deadbugs++;
                        deadevilbugs++;
                    }
            // Check for evilBug-to-evilBug collisions
            for (int a=0; a<maxevilbugs-1; a++)
                for (int b=a+1; b<maxevilbugs; b++)
                    if ( earray[a].isAlive() &
                        earray[b].isAlive() &
                        earray[b].impactBug(earray[a])){
                        earray[a].isDead();
                        earray[b].isDead();
                        deadevilbugs++;
                        deadevilbugs++;
                    }
        }
    }
}
```

```

// Check for Bug-to-BugZapper collisions
for (int a=0; a<maxbugs; a++)
    for (int b=0; b<maxbug_zappers; b++)
        if ( parray[a].isAlive() &
            rarray[b].impactBug(parray[a])){
            parray[a].isDead( );
            deadbugs++;
        }
// Check for evilBug-to-BugZapper collisions
for (int a=0; a<maxevilbugs; a++)
    for (int b=0; b<maxbug_zappers; b++)
        if ( earray[a].isAlive() &
            rarray[b].impactBug(earray[a])){
            earray[a].isDead( );
            deadevilbugs++;
        }

// Check for Game Over
if (maxbugs <= deadbugs & maxevilbugs <=
    deadevilbugs) break out;
for (int x=0; x<maxbugs; x++)
    if (parray[x].isAlive( ) &
        p.impactBug(parray[x])) break out;
for (int x=0; x<maxevilbugs; x++)
    if (earray[x].isAlive( ) &
        earray[x].impactBug(p)) break out;
for (int y=0; y<maxbug_zappers; y++)
    if (rarray[y].impactBug(p)) break out;

    pause(50);
}
}
}

private void pause (int time) {
    try    { Thread.sleep(time); }
    catch (Exception e) {}
}

```

```
// **** internal class to handle button actions ****
public class myButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (leftButton == e.getSource() ) {
            p.turnleft(); p.turnleft();
        } else if (rightButton == e.getSource() ) {
            p.turnright(); p.turnright();
        } else if (startButton == e.getSource() ) {
            startPressed = true;
        }
    }
}
}
```

Sample Run of Class Fly

```
C:\jdk1.3\projects\gameexample>javac fly.java
C:\jdk1.3\projects\gameexample>appletviewer fly.html
```

