

12. Inheriting *Bug* to Make Class *EvilBug*

Class *EvilBug* begins by inheriting *Bug* by means of the *extends* keyword and creating a new constructor for itself. This constructor must do the same things as the one created for *Bug* but for the inclusion of a parameter of class *Bug* to represent the intended victim of the evil bug. This is made necessary by the fact that objects of *EvilBug* are to seek out and destroy the *Bug* object controlled by the user. (Notice the addition of the private instance variable of class *Bug* called *victim*. Also notice a repeat of all the assignments in class *Bug* are avoided by a *super* call to the constructor of class *Bug*.)

```
public class EvilBug extends Bug {

    private Bug victim;

    public EvilBug(int w, int h,
                  int initx, int inity,
                  Bug initVictim) {
        super(w, h, initx, inity);
        victim = initVictim;
    }
}
```

The next feature that needs to be added to complete *EvilBug* is the ability to turn in the direction of the victim *Bug* object. This is the method *calcDirection*.

The basic idea of *calcDirection* is to get the (x,y) coordinate pair of the victim, determine the slope to the victim, then set direction to point down or up that slope. Care must be taken to insure that if the x data of the victim and the *EvilBug* object are the same that the slope calculation does not take place (to avoid division by zero). In that case, all that need be determined is if the victim is above or below the *EvilBug* object. In the case where the x data of both objects is the same, a shortcut is available because all that need be determined is whether the Bug is to the left or right of the *EvilBug* object.

```
public void calcDirection() {
    double vx = victim.returnx();
    double vy = victim.returny();
    int direction = 0;
    if (y == vy) {
        if (x > vx) direction = 12;
        else direction = 4;
    } else if (x == vx) {
        if (y > vy) direction = 8;
        else direction = 0;
    } else {
```

```

        double slope = (vy - y) / (vx - x);
        if (slope > 0) {
            if (slope < .5) {
                if (x > vx) direction = 12;
                else direction = 4;
            } else if (slope > 2) {
                if (x > vx) direction = 0;
                else direction = 8;
            } else {
                if (x > vx) direction = 14;
                else direction = 6;
            }
        } else if (slope < 0) {
            if (slope > -.5) {
                if (x > vx) direction = 12;
                else direction = 4;
            } else if (slope < -2) {
                if (x > vx) direction = 8;
                else direction = 0;
            } else {
                if (x > vx) direction = 10;
                else direction = 2;
            }
        }
    }
    setdirection(direction);
}

```

Full Listing of Class *EvilBug*

```

import java.awt.*;
import javax.swing.*;

public class EvilBug extends Bug {

    private Bug victim;

    public EvilBug(int w, int h,
                  int initx, int inity,
                  Bug initVictim) {
        super(w, h, initx, inity);
        victim = initVictim;
    }

    public void calcDirection() {
        double vx = victim.returnx();
    }
}

```

```
double vy = victim.returny();
int direction = 0;
if (y == vy) {
    if (x > vx) direction = 12;
    else direction = 4;
} else if (x == vx) {
    if (y > vy) direction = 8;
    else direction = 0;
} else {
    double slope = (vy - y) / (vx - x);
    if (slope > 0) {
        if (slope < .5) {
            if (x > vx) direction = 12;
            else direction = 4;
        }
        else if (slope > 2) {
            if (x > vx) direction = 0;
            else direction = 8;
        }
        else {
            if (x > vx) direction = 14;
            else direction = 6;
        }
    }
    else if (slope < 0) {
        if (slope > -.5) {
            if (x > vx) direction = 12;
            else direction = 4;
        }
        else if (slope < -2) {
            if (x > vx) direction = 8;
            else direction = 0;
        }
        else {
            if (x > vx) direction = 10;
            else direction = 2;
        }
    }
}
setdirection(direction);
}
```

13. Animation, *Runnable* and Testing Class *EvilBug*

Testing class *EvilBug* involves adding *EvilBug* objects to the current version of *MainPanel* and selecting suitable targets for them. Notice that objects of *Bug* class and *EvilBug* class can be tested for crashes via the method *impactBug*.

In this example, to give the *EvilBug* objects a bit of a workout, two *Bug* objects are set to move randomly. This is accomplished when the method *random* of class *Math* is called to make a value for the new instance variable *temp*. Since *random* returns a value from 0 to 1, multiplying the return of the call to *random* by 8 gives a value from 0 to 8. Casting it allows the result to be assigned to the integer variable *temp*, which can then be tested to determine the direction that two *Bug* objects should turn.

Except for the bizarre twists of logic necessary to understand what is going on in method *calcDirection*, class *EvilBug* did not include any new Java bits, so this is a good place to introduce the complications of animation into the tutorial.

To start with, we need to get the loop out of *paintComponent* in class *MainPanel*. The reason is that *paintComponent* basically assembles one visual frame to be placed in the *Container* object of the window. This is the reason that the loop in the current version of *MainPanel* does not cause any animation when the program runs. To show the movement seen as animation, the method *paintComponent* must be called repeatedly while the program runs. Each time the *paintComponent* is called, a new view is then built and placed in the window.

Just how is this to be done? Simply calling *paintComponent* from inside a loop in class *JavaByBugs* would work, but would make it more difficult to control the game later. Java includes a construct similar to a class called an *interface*, a number of which already exist. *Interfaces* impose various methods, some of which are *abstract*. *Abstract* methods contain no code and must be overwritten by the user. In addition, interfaces can include a *logic cycle*.

Interfaces are added to a program via the *implements* option of the class header. This has the following syntax:

```
public class class-name [extends class] [implements interface [, interface [...]]] {
```

The interface *Runnable* imposes a logic cycle that we can exploit for our interactive game. *Runnable* consists of these methods:

```
void start() {}
abstract void run() {}
void stop() {}
```

Notice that only *run* is abstract and must be implemented by the user. However, *start* is used to execute *run* and *stop* is used to clean up after *run*. Method *start* is cannot be called until after the constructor has been executed.

Any repetitive logic belongs in the *run* method. Since *run* has no access to the current *Graphics* object, it must use *repaint* in order to call *paintComponent*.

In Java, *threads* are used to control multiple paths of logic in a program. Unlike C++, they work as advertised in Java. Each Java program has a thread. Class *Thread* is used to create additional thread objects.

Declaring an object of class *Thread* and creating an instance of it have the usual declaration/creation syntax:

```
access Thread identifier[, identifier[, identifier[...]]];
```

```
thread-object = new Thread(thread-object);
```

In order to make a *Thread* that is associated with *Runnable*, the *Thread* object that is passed to the constructor is *this*. *this* is a keyword that is always self referential to the current context.

In *MainPanel*, a *Thread* object called *animate* will be created. It will become the main thread of the *Runnable* interface. First *animate* will be created as a global variable:

```
private Thread animate;
```

Then *animate* will be created and associated with *Runnable* in the constructor:

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

To start a thread, the *Thread* method *start* is called. This has the syntax:

```
void thread-object.start( )
```

In *MainPanel*, the method *start* will be written as follows:

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

This will cause the method *run* to become active, which will consist of a loop that repeatedly calls *repaint*.

An additional problem that exists and must be dealt with is the fact that animation can take place too quickly, especially in an interactive game. The *Thread* class object *animate* can be used to control this. By calling the *Thread* class method *sleep* with a parameter giving the number of milliseconds to pause, the animation can be slowed down. This has the syntax:

```
void sleep(int)
```

In *MainPanel*, this will be written as:

```
animate.sleep(int-expression);
```

Exceptions are how Java allows programmers to include code to deal with run time errors. Calling the method *sleep* can result in an exception of class *Exception* (there are other exception classes) being thrown. The *try-catch* control structure can be used to insure that the program does not halt or behave strangely just because a minor error occurs. The *try-catch* syntax is as follows:

```
try {
    ... code ...
} [catch (exception-class identifier) {
    ... code ...
}] [catch (exception-class identifier) {
    ... code ...
}] [...]
}[finally {
    ... code ...
}]
```

There can be as many *catch* sections as needed, or none (if there is a *finally*). There can be a *finally* section or not (if there is a *catch*). The *catch* and *finally* sections act as *cases* in a *switch*, locating the exception-class parameter that is the same as the exception that has been thrown. If no exceptions match, *finally* will be executed.

In *MainPanel*, a method called *pause* will be created that will receive an *int* parameter of milliseconds to pause and will call *sleep* in a *try-catch* control structure. This will be written:

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

Since there really isn't anything to do if an exception is thrown except keep going if possible, there is no code in the *catch* section.

Method *pause* will be called from the loop in method *run* in order to slow the program down by 50 milliseconds between calls to *repaint*. Method *run* will be written:

```
public void run() {
    for (int x=0; x<200; x++) {
        repaint();
        pause(50);
    }
}
```

The loop will repeat 200 times, an arbitrary number.

Before this instance of *MainPanel* is complete, one other issue must be dealt with. If graphical objects are drawn directly to the *Graphics* object that is being displayed and that drawing occurs in a loop such as needed for animation, the result will be “flicker”, where portions of the window will visibly appear and disappear. This is an unpleasant thing to have to focus on, so we must prevent flicker in our game.

Preventing flicker requires a technique called *double buffering*. In this technique, a second area of memory is created and used as a place to draw graphical objects. Then, when all objects have been drawn, this area is used to overwrite the current *Graphics* object.

The buffer can be declared from class *Image*, part of the Abstract Windows Toolkit. This has the syntax:

```
access Image identifier;
```

In *MainPanel*, the image will be declared a global variable as show here:

```
private Image myBuffer;
```

Image class object *myBuffer* will need to be created each time *paintComponent* is called. [This sounds like a waste of memory, but remember Java will automatically collect the “garbage” (no longer needed memory) and recycle it.] This has the syntax:

```
image-object = createImage(int width, int height);
```

The following line will be inserted into *paintComponent* immediately after the *super* call:

```
myBuffer = createImage(WIDTH, HEIGHT);
```

Now that the memory has been allocated, this memory must be given to a *Graphics* object. (Only *Graphics* class objects can draw.) This is done by declaring a *Graphics* object, but calling the *Image* method *getGraphics* (which returns an object of class *Graphics* that refers to the memory allocated to the *Image* object) rather than the *Graphics* class constructor. This has the syntax:

```
Graphics identifier = image-object.getGraphics();
```

In the *paintComponent* method, the next statement after *myBuffer* is created will be:

```
Graphics hiddenG = myBuffer.getGraphics();
```

From this point on and with one exception, any graphics methods must be called by *hiddenG* instead of *g*. Each time *hiddenG* is used to call a *Graphics* method or is passed as a parameter to a *paintComponent* (or *paint*) method, the memory belonging to *myBuffer* will receive the drawing.

The only other time that *g* will be used to draw will be when it is time to pass the data in the memory allocated to *myBuffer* to *g*. This will be done with the *drawImage* *Graphics* class method, which has the syntax:

```
void graphics-object.drawImage(image-object,0,0,this)
```

The following will be one of the last lines in the method *paintComponent* of *MainPanel*:

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

Finally, in this version of *MainPanel*, a number of variables of class *Bug* and class *EvilBug* will be declared as global variables. They will be instantiated in the constructor and have their colors set. Method *paintComponent* will use these objects to draw in the *hiddenG* before it is displayed in the window.

Here is the listing of *MainPanel.java* that will be used to test class *EvilBug*:

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

public class MainPanel extends JPanel implements Runnable
{
    private final int WIDTH;
    private final int HEIGHT;

    Bug p, q, r;
    EvilBug s, t, u, v, w;

    private Image myBuffer;
    private Thread animate;

    public MainPanel(int wInit, int hInit) {
        WIDTH = wInit;
        HEIGHT = hInit;

        animate = new Thread(this);

        p = new Bug(WIDTH,HEIGHT,100,275);
        q = new Bug(WIDTH,HEIGHT,300,50);
        r = new Bug(WIDTH,HEIGHT,200,50);

        s = new EvilBug(WIDTH,HEIGHT,200,350,r);
        t = new EvilBug(WIDTH,HEIGHT,0,0,r);
        u = new EvilBug(WIDTH,HEIGHT,500,400,r);
        v = new EvilBug(WIDTH,HEIGHT,0,400,p);
        w = new EvilBug(WIDTH,HEIGHT,500,0,q);
    }
}
```

```
    p.setBugColor(0,0,255);
    q.setBugColor(255,0,0);
    r.setBugColor(0,255,255);
    s.setBugColor(255,255,0);
    t.setBugColor(127,127,127);
    u.setBugColor(127,127,127);
    v.setBugColor(127,127,127);
    w.setBugColor(127,127,127);
}

public void paintComponent(Graphics g) {
    super.paintComponent(g);

    myBuffer = createImage(WIDTH,HEIGHT);
    Graphics hiddenG = myBuffer.getGraphics();
    hiddenG.setColor(Color.black);
        hiddenG.fillRect(0, 0, WIDTH, HEIGHT);

    int temp;

    r.turnleft();

    temp = (int)(Math.random()*8);

    if (temp > 6) p.turnleft();
    else if (temp < 2) p.turnright();

    if (temp > 6) q.turnleft();
    else if (temp < 2) q.turnright();

    p.go();  q.go();  r.go();

    s.calcDirection();  s.go();
    t.calcDirection();  t.go();
    u.calcDirection();  u.go();
    v.calcDirection();  v.go();
    w.calcDirection();  w.go();

    p.paintComponent(hiddenG);
    q.paintComponent(hiddenG);
    r.paintComponent(hiddenG);
    s.paintComponent(hiddenG);
    t.paintComponent(hiddenG);
    u.paintComponent(hiddenG);
    v.paintComponent(hiddenG);
}
```

```
        w.paintComponent(hiddenG);

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

        if (s.impactBug(r)) System.out.print("crash\n");
    }

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

    public void run() {
        for (int x=0; x<200; x++) {
            repaint();
            pause(50);
        }
    }

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

14. One Last Alteration to JavaByBugs

Class *JavaByBugs* must be altered to include a call to the *start* method of the *MainPanel* object *gamePanel*. This will be the last line in the constructor of *JavaByBugs* and will be written:

```
gamePanel.start();
```

Here is the full listing of JavaByBugs:

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

class JavaByBugs extends JFrame {

    private final int HEIGHT = 400;
    private final int WIDTH = 500;

    private MainPanel gamePanel;

    public JavaByBugs() {

        setSize(WIDTH, HEIGHT);
        setTitle("Java by Bugs");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocation(200,50);

        Container c = getContentPane();

        c.setLayout(new BorderLayout());
        c.setFont(new Font("SansSerif", Font.BOLD, 12));
        c.setBackground(Color.white);

        gamePanel = new MainPanel(WIDTH, HEIGHT);
        c.add(gamePanel, BorderLayout.CENTER);

        gamePanel.start();
    }

    public static void main(String args[]) {
        JavaByBugs mainFrame = new JavaByBugs();
        mainFrame.show();
    }
}
```

15. Testing the Program

JavaByBugs.java, EvilBug.java and MainPanel.java must be compiled:

```
javac MainPanel.java
javac EvilBug.java
javac JavaByBugs.java
```

At this point the program can be run,

```
java JavaByBugs
```

which will produce an animated example similar to this:

