

17. Making the Game with Buttons

Now that all elements of the game are present, it is time to use class *MainPanel* to actually build the interactive game. *MainPanel* will have to be substantially rebuilt, but a good bit of the existing code will be used from the previous version of *MainPanel*. Code that will be retained is:

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

public class MainPanel extends JPanel implements Runnable {

    private final int WIDTH;
    private final int HEIGHT;

    private Image myBuffer;
    private Thread animate;

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

        animate = new Thread(this);
    }

    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);

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

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

    public void run() {
        repaint();
        pause(50);
    }

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

First, in order to detect user actions, it is necessary to add an import of the AWT *event* package,

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

Next, the user's bug needs to be created as a global object. This will be object *p* (for primary bug):

```
private Bug p;
```

Bug object *p* will be created in the constructor by:

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

This code will create the user's bug at the bottom of the screen, somewhere in the middle 50% of the across pixels. (The call to *Math.random()* returns a number between 0 and 1.)

The user's bug will be a unique color. This will be bright red. This code will be placed in the constructor and will set the color of the user's bug:

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

Once started, *run* needs to execute until the window is closed. This can be accomplished with a simple *while(true)* loop:

```
public void run( ){
    while (true) {
        repaint( );
        pause(50);
    }
}
```

The logic in *run* also needs a means of letting the user set it in motion. To this end the boolean variable *startPressed* will be created as a global variable and set to *false*.

```
private boolean startPressed = false;
```

A simple *if* statement can be inserted into the loop logic of *run* to make sure that the game cannot start until *startPressed* is true.

```
public void run( ){
    while (true) {
        if (!startPressed) continue;
        repaint();
        pause(50);
    }
}
```

It is now time to introduce 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 *MainPanel*, buttons will be used to create events when the user makes a selection during the game. The tutorial will include an internal class called *myButtonHandler* that implements *ActionListener*. (*ActionListener* comes from the importing of *event*.) Class *myButtonHandler* will listen for any button selections that the user makes and perform the required actions. 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 *MainPanel*, there will be three buttons. These are *leftButton* (for a left turn of the user's bug), *rightButton* (for a right turn of the user's bug) and *start*, which will allow the user to look over the window before beginning the game. Class *JButton* will be used to declare and create these buttons. The syntax of declaration and creation of a *JButton* object is as follows:

```
access JButton identifier[,identifier[...]];
JButton-object = new JButton(string-label);
```

The three buttons will be declared as global variables:

```
private JButton leftButton, rightButton, startButton;
```

They are then created in the constructor:

```
startButton = new JButton("Start");
leftButton = new JButton("Turn Left");
rightButton = new JButton("Turn Right");
```

Here is the internal class *myButtonListener* for class *MainPanel*. (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;
        }
    }
}
```

While it looks that we are all set to let the user enter moves via buttons, in truth there are several steps left. Just declaring and creating *JButtons* and making an *ActionListener* class does not do it. The *ActionListener* class must be explicitly connected to each button. This is accomplished via the *addActionListener* method of the *JButton* class. This has the syntax:

```
void addActionListener(actionlistener-class-object)
```

Top attach the three buttons to the *ActionListener* class *myButtonHandler*, the following three lines must be added to the constructor:

```
startButton.addActionListener(new myButtonHandler());
leftButton.addActionListener(new myButtonHandler());
rightButton.addActionListener(new myButtonHandler());
```

Finally, the three buttons must be added to the panel represented by *MainPanel* class. This is done with the *add* statement. Since *MainPanel* inherited *JPanel*, an instance variable is not needed. The following code is added to the constructor of *MainPanel*:

```
add(startButton);
add(leftButton);
add(rightButton);
```

Additional Global Instance Variables

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

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

Global arrays to contain the *BugZapper*, *Bug* and *EvilBug* objects are also needed. 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 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 global variables to keep count of the number of defunct bugs and evil bugs:

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

Additional New Constructor Code

The array of *Bug* objects must be created in the constructor of *MainPanel*. 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(WIDTH,HEIGHT,
        (int)(WIDTH*Math.random()),
        (int)(HEIGHT*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();
}
```

The array of *EvilBug* objects must be created in the constructor. 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 and array.

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

The array of *BugZapper* objects must also be created in the constructor. 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 and array.

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

Some Convenient Methods

Since the elements of the arrays of bugs, evil bugs, and bug zappers must be repeatedly displayed and, in the case of the arrays of bugs and evil bugs, moved, *run* and *paintComponent* could become quite large if all of that code was put directly in them. Rather than stuffing all of that code into *run* and *paintComponent*, this tutorial will include methods for displaying and moving the elements of the above mentioned arrays. These are as follows:

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

private 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].paintComponent(g);
}

private 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].paintComponent(g);
}
```

Some More Convenient Methods

In each pass through the body of the loop in method *run*, collisions will need to be detected between all of the elements of the arrays of bugs, evil bugs and bug zappers so that the appropriate bugs may be taken out of the game. These collisions can be between bugs and bugs, bugs and evil bugs, evil bugs and evil bugs, bugs and bug zappers and, finally, evil bugs and bug zappers.

Rather than placing all of this code in one very long loop in method *run*, this tutorial will create a method to check for each of the possible types of collisions. These are as follows:

```
public void CollideBugToBug() {
    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++;
            }
}

public void CollideBugToEvilBug() {
    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++;
            }
}
```

```
public void CollideEvilBugToEvilBug() {
    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])){
                array[a].isDead();
                earray[b].isDead();
                deadevilbugs++;
                deadevilbugs++;
            }
}

public void CollideBugToBugZapper() {
    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++;
            }
}

public void CollideEvilBugToBugBugZapper() {
    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++;
            }
}
```

New Code for the Loop in Method *run*

Method *run* can now be modified to include in the loop the move and the paint of bugs, evil bugs and the user's bug, after which collisions can be detected. This code is as follows:

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

        CollideBugToBug();
        CollideBugToEvilBug();
        CollideEvilBugToEvilBug();
        CollideBugToBugZapper();
        CollideEvilBugToBugBugZapper();

        pause(50);
    }
}
```

Lastly, the code in the loop of *run* must check to see if the game is over. This could be done by setting a Boolean variable to a desired value and using that variable in the Boolean expression of the *while* statement. Instead, this tutorial will put a *label* before the *while* statement and *break* to that *label*. In Java, *labels* are created when needed by the following syntax:

label-identifier:

In method *run*, the *label out:* will be placed before the while statement as follows:

```
out:while(true) {
```

When necessary, a *break* statement can be followed by a *label*. This will cause the *break* to break through all inclosing blocks of code all the way to the *label*. The syntax of a *break* to a label is:

```
break label;
```

In method *run*, this will be:

```
break out;
```

which will cause loop in *run* to halt and the game to stop.

The game conditions that would cause a halt are:

- All regular bugs and evil bugs are dead.
- The user's bug collides with any other type of bug.
- The user's bug collides with a bug-zapper.

The code to accomplish this is as follows:

```

    if (maxbugs < deadbugs+1 && maxevilbugs < deadevilbugs+1)
        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;

```

New Code for Method *paintComponent*

All that is left to write is the code in method *paintComponent* to cause the bugs, bug zappers, evil bugs and the users bug to be drawn and, for the user's convenience, information about the state of the game.

The code to cause the components of the game board to be drawn is as follows:

```

    paintBugZappers(hiddenG);
    paintBugArray(hiddenG);
    paintEvilBugArray(hiddenG);
    p.paintComponent(hiddenG);

```

The *Graphics* class method *drawString* allows text to be displayed in the window. It has the syntax:

```
void drawString(String, int x, int y)
```

where *int* x and *int* y are the upper left position of the string to be output

Since some of the data that needs to be displayed is in integer form, the method *toString* of wrapper class *Integer* will be used to convert the data from numeric form to *String*. This has the syntax:

```
static String toString(integer-expression)
```

By using `drawString` and the method `toString` from class `Integer`, the number of various bugs and bug zappers can be output at a fixed position:

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

18. A Complete Listing of `MainPanel.java`

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

public class MainPanel extends JPanel implements Runnable {

    private final int WIDTH;
    private final int HEIGHT;

    private Image myBuffer;
    private Thread animate;

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

    private BugZapper rarray[];
    private Bug parray[];
    private EvilBug earray[];
    private Bug p;

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

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

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

        p = new Bug(WIDTH,HEIGHT,
            WIDTH/4+(int)((WIDTH/2)*Math.random()),
            HEIGHT-20);

        animate = new Thread(this);
```

```
// setup bug array
parray = new Bug[maxbugs];
for (int x=0; x<maxbugs; x++) {
    parray[x] = new Bug(WIDTH,HEIGHT,
        (int)(WIDTH*Math.random()),
        (int)(HEIGHT*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(WIDTH,HEIGHT,
        (int)(WIDTH*Math.random()),
        (int)(HEIGHT*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)(WIDTH*Math.random()),
        (int)(HEIGHT*Math.random()),
        (int)(50*Math.random()+20));

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

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

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

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

```
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);

    paintBugZappers(hiddenG);
    paintBugArray(hiddenG);
    paintEvilBugArray(hiddenG);
    p.paintComponent(hiddenG);
    hiddenG.drawString(
        Integer.toString(maxbugs-deadbugs)
        + " Bugs, "
        + Integer.toString(maxevilbugs-deadevilbugs)
        + " Evil Bugs, "
        + Integer.toString(maxbug_zappers)
        + " BugZappers",HEIGHT/2-40,40);

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

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

private 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].paintComponent(g);
}

private 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].paintComponent(g);
}

public void CollideBugToBug() {
    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++;
            }
}

public void CollideBugToEvilBug() {
    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++;
            }
}

public void CollideEvilBugToEvilBug() {
    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++;
            }
}

public void CollideBugToBugZapper() {
    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++;
            }
}

```

```

public void CollideEvilBugToBugBugZapper() {
    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++;
            }
}

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

public void run(){

    out:while(true) {
        if (!startPressed) continue;
        moveBugArray();
        moveEvilBugArray();
        p.go();
        repaint();

        CollideBugToBug();
        CollideBugToEvilBug();
        CollideEvilBugToEvilBug();
        CollideBugToBugZapper();
        CollideEvilBugToBugBugZapper();

        // Check for Game Over
        if (maxbugs < deadbugs+1 &&
            maxevilbugs < deadevilbugs+1)
            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 { animate.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;
        }
    }
}
}
```

19. Running the Button Version of the Game

Now when the file MainPanel.java is compiled:

```
javac MainPanel.java
```

and the file JavaByBugs.class is run:

```
java JavaByBugs
```

this is an example of the game that will be produced and can be played by the user:

