

10. Class Bug

The most complex class of this example project is class *Bug*. This partly because of the number of state variables and methods required, but also because of the complexity of some of the actions. *Bug* objects must have a recognizable shape, move in a 2-dimensional environment and rotate to indicate direction of movement. In addition to this, *Bug* objects may collide with other *Bug* objects and must be able to detect this.

First, class *Bug* must import both AWT and Swing:

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

public class Bug {
}
```

Global Instance Variables for Storing State Information

To accomplish all of its goals, *Bug* must include the following global instance variables in order to record the necessary state information. Most of them are fairly obvious, but some bear further explanation.

```
protected int x, y;           // location to display bug
protected double rx, ry;     // actual location of bug
protected int color;         // color of bug
protected int bugPoints=12;  // number of points in a bug
protected int speed;         // number of pixels to cover in one
                             // move
protected int width;         // width of flying area in pixels
protected int height;        // height of flying area in pixels
protected int direction;     // direction to move in
protected int oledirection;  // previous direction moved in
protected int MAXDIRECTION=16;
                             // number of directions possible in
                             // a circle
protected int R=255, G=255, B=255;
                             // default red, green and blue
                             // values for bug color
protected int impactzone = 5; // radius from center front of bug
                             // to test for other objects
protected boolean alive = true;
                             // the bug can be alive or not,
                             // this value can be used or
                             // ignored
```

Variable *direction* contains a value from 0 to *MAXDIRECTION*-1. If *MAXDIRECTION* is 16 (as is the case in this example), there will be 16 possible directions for a bug in 360 degrees and direction 0 will point due north, direction 4 will point due east, etc. *MAXDIRECTION* can be altered and *Bug* recompiled to change the possible number of directions.

Variable *oledirection* is used to determine if the direction has changed.

The following two arrays are used to store the original relative (to x and y) points of the polygon that makes up the bug before any movement or rotation has occurred. A bug always starts off pointing in the same direction (north). It is then rotated to its correct direction using matrix multiplication. (This prevents bugs from become shapeless blobs on the screen as small math truncations add up over time.) The values in $xpto$ and $ypto$ are never changed. All changes are placed in the array pair xpt and ypt , which follow. The bug is displayed based on the values in xpt and ypt relative to the values in x and y .

```
protected double xpto[]
    = {2.0, 2.0, 5.0, 5.0, 2.0, 1.0, -1.0, -2.0, -5.0, -5.0, -2.0, -2.0};
protected double ypto[]
    = {0.0, 3.0, 2.0, 6.0, 5.0, 9.0, 9.0, 5.0, 6.0, 2.0, 3.0, 0.0};
```

These two arrays contain the results of moving and rotating the relative points of the bug. Displaying the bug is always performed from the values contained in these arrays, so they start off with duplicate values of arrays $xpto$ and $ypto$:

```
protected double xpt[]
    = {2.0, 2.0, 5.0, 5.0, 2.0, 1.0, -1.0, -2.0, -5.0, -5.0, -2.0, -2.0};
protected double ypt[]
    = {0.0, 3.0, 2.0, 6.0, 5.0, 9.0, 9.0, 5.0, 6.0, 2.0, 3.0, 0.0};
```

A Constructor for Bug

The constructor for class *Bug* requires parameter values to initialize the width and height of the flight area and the initial location of the bug. In addition, it sets the speed arbitrarily to a middle value and sets *direction* and *oledirection* to point north.

```
public Bug(int w, int h, int initx, int inity) {
    width = w;
    height = h;
    speed = 5;
    x = initx;
    y = inity;
    rx = x;
    ry = y;
    direction = 0;
    oledirection = 0;
    alive = true;
}
```

Basic Control and Reporting Methods

The simple methods *returnx* and *returny* report the location pixel pair of the front of the bug. There are written:

```
public int returnx() { return x; }
public int returny() { return y; }
```

Method *setdirection* accepts a direction as a parameter and changes the value in the instance variable *direction*. Variable *oledirection* has its value set to -1, an impossible direction, which will insure the direction set will be regarded by the program as a new direction. Method *setdirection* contains the first example in the tutorial of the Java *if* control structure. This structure works like its C++ counterpart (as do *if-else*, *nested-ifs* and *nested if-elses*).

```
public void setdirection(int d) {
    if (d >=0 && d < MAXDIRECTION) {
        direction = d;
        oledirection = -1;
    }
}
```

Like C++, there are two logical *and* operators and two logical *or* operators in Java. The logical *and* operators are & and &&. The logical *or* operators are | and ||. The single character operators always evaluate both sides of the equation, whether it is necessary or not. The double character operators will only evaluate the right side of a Boolean expression if the result of the left side of the Boolean expression warrants it. This can be used to control whether or not a method call in the right side of a Boolean expression is executed or not.

Other logical operators in Java include:

```
^    exclusive or
!    logical not (unary operator)
&=  logical and assignment
|=  logical or assignment
^=  exclusive or assignment
==  equal to
!=  not equal to
?:  ternary if-then-else
```

ternary expression syntax:

```
variable = boolean_expression ? expression1 : expression2;
```

which means that if *boolean_expression* is true, the result of *expression1* is assigned to *variable*; if it is false, the result of *expression2* is assigned to *variable*.

Methods *turnright* and *turnleft* adjust the direction of the bug by one unit. In them, *oledirection* is always set to the current value of *direction* before the value of *direction* is adjusted. Adjustment is up by one for a right turn, down by one for a left turn.

```
public void turnright() {
    oledirection = direction;
    direction++;
    if (direction >= MAXDIRECTION) direction = 0;
}

public void turnleft() {
    oledirection = direction;
    direction--;
    if (direction < 0) direction = MAXDIRECTION-1;
}
```

Method *setBugColor* accepts a value for red, green and blue (variables *R*, *G* and *B*) but does not actually set the color of the bug, just preserves it.

```
public void setBugColor(int r, int g, int b) {
    if (r<256 & r>=0) R = r;
    if (g<256 & g>=0) G = g;
    if (b<256 & b>=0) B = b;
}
```

Methods *isAlive* and *isDead* seem like they should be opposites of one another, but they are not. Method *isAlive* returns the value in the Boolean variable *alive*. Method *isDead* sets the Boolean variable *alive* to *false*.

```
public boolean isAlive(){ return alive; }
public void isDead(){ alive = false; }
```

Method *impactBug* receives another *Bug* object and then checks if it is within the impact zone the of bug object. If it does, *true* is returned. If it does not, *false* is returned. This method does not set the value of *alive*. That relationship can be established later in a sub class (class that inherits, in this case, class *Bug*) in whatever manner is appropriate. (Note: As it is written, *impactBug* will cause a problem when it is inherited by another class. This is deliberately done to show some of the issues involved in creating classes by inheritance in Java.)

```
public boolean impactBug(Bug p) {
    if (p.returnx()<x+impactzone && p.returnx()>x-impactzone &&
        p.returny()<y+impactzone && p.returny()>y-impactzone){
        return true;
    } else return false;
}
```

paintComponent for Class Bug

To make a *Bug* object appear on the screen, method *paintComponent* is called with a *Graphics* object as the only parameter. Method *paintComponent* is in itself simple. If the direction has changed, the new location of all bug points must be calculated by a call to the method *void calculateBugParts()* (to be defined shortly). Whether or not the direction of the bug has changed, it is then drawn by a call to the method *void paintBug(Graphics)* (also to be defined shortly).

```
public void paintComponent (Graphics g) {
    if (oledirection != direction)
        calculateBugParts();
    paintBug(g);
}
```

Of the two called methods, *paintBug* is much the simpler. First, it receives a *Graphics* class object, which will be used to set a color and draw a filled polygon in that color. This polygon is the bug and will be constructed of (x,y) pairs from the arrays *xpt* and *ypt*.

Inside the method, an object of class *Polygon* is created. Using a *for* loop (which operates very much like the *for* loop of C++), a call is made to the *Polygon* class method *addPoint*. Each point added to the polygon is represented by an x and y value. The x value of the point is calculated by taking the actual x location of the bug then adding the rounded value of *xpt[i]* to it. (*The rounding is accomplished in much the same way that it would be performed in C++. First the value at xpt[i], which is a double, has .5 added to it. The result is then cast to integer before being added to x.*) The y value of the point is calculated by taking the actual y location of the bug then adding the rounded value of *ypt[i]* to it.

Once all of the points are added to the *Polygon* object, a new color is created (that color is set via a call to the *Graphics* method *setColor*) and the polygon is then drawn by a call to the *Graphics* method *fillPolygon*.

```
public void paintBug(Graphics g) {
    Polygon p = new Polygon();
    for (int i=0; i<bugPoints; i++)
        p.addPoint(x+((int)(xpt[i]+0.5)),
                  y+((int)(ypt[i]+0.5)));
    Color mycolor = new Color(R,G,B);
    g.setColor(mycolor);
    g.fillPolygon(p);
}
```

When the method *calculateBugParts* is called, the points that make up the bug must be rotated into a new direction. The bug will always be recalculated from the original values in arrays *xpto* and *ypto*. (This prevents the gradual disintegration of the bugs in to shapeless blobs due to truncation errors or the possible use of simple heuristics in classes that inherit *Bug*.)

The first step is to determine if the direction of the bug is north or not. If the direction is not north, the points that make up the bug (located in arrays *xpto* and *ypto*) must be rotated into the correct positions from the north position and placed in the arrays that will be used to draw the bug (*xpt* and *ypt*). If not, all that is necessary is a simple transfer of the original points (in arrays *xpto* and *ypto*) to the arrays that will actually be used to draw the bug (*xpt* and *ypt*).

If direction is not north, the next step is to calculate the new direction in terms of radians (*Note: The functions for sine and cosine will be needed. They are defined in terms of radians and expect radian parameters to be passed to them. This is the same as in C++.*). Since there are roughly 6.28 radians in a circle, the radian value of the new direction can be calculated by multiplying 6.28 times the new direction divided by the maximum possible number of directions. This is then assigned as the value of the variable *rotation*.

```
double rotation = 6.28 * ((double) direction) /
                  ((double)MAXDIRECTION);
```

After the rotation value is calculated, its sine and cosine must be computed and stored.

```
double c = Math.cos(rotation);
double s = Math.sin(rotation);
```

Since class *Math* has not been imported and no instance variables of class *Math* have been created, methods *sin* and *cos* would not normally be available. Java, however, provides a means for methods to be used individually. This is accomplished by specifying class *Math* and a period in front of each of methods *sin* and *cos*. (Most of the methods of *Math* have been given the *static* attribute.)

The sine and cosine of the rotation value will now be used to create a 2 dimensional matrix to be used to rotate each bug point via matrix multiplication. A one dimensional matrix containing a point to be rotated will be created, then that matrix will be multiplied against the 2 dimensional rotation matrix, which will result in a third matrix of a single dimension that will contain the new point (x,y) pair. All points in the bugs polygon must be rotated in this manner. These matrix calculations take on the following form.

$$\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot [x_o \quad y_o \quad 1] = [x_n \quad y_n \quad 1]$$

Matrix multiplication is accomplished via the following formula:

$$C[i,k] = \sum_j A[i,j] \cdot B[j,k]$$

Since we are multiplying a matrix of two dimensions against a matrix of one dimension, the result will be a matrix of one dimension. The above formula can be simplified to

$$C[i] = \sum_j A[i,j] \cdot B[j]$$

To accomplish this rotation, the two-dimensional rotation matrix is created and populated.

```
double trt[][] = new double[3][3];
trt[0][0] = c;  trt[0][1] = -s;  trt[0][2] = 0;
trt[1][0] = s;  trt[1][1] = c;   trt[1][2] = 0;
trt[2][0] = 0;  trt[2][1] = 0;   trt[2][2] = 1;
```

The matrix to multiply against the rotation matrix must be created, as must the matrix to receive the rotated point.

```
double point[]    = new double[3];
double newpoint[] = new double[3];
```

A for loop can be used to generate the needed loop and index changes necessary to apply the rotation operation to all (x,y) pairs.

```
for (int p=0; p<bugPoints; p++) {
```

Given the index generated by the for loop, the point matrix can be populated:

```
    point[0] = xpto[p];
    point[1] = ypto[p];
    point[2] = 1;
```

Now that both the rotation matrix and point matrix contain values, the matrix multiplication can now occur.

```
    for (int i=0; i<3; i++) {
        newpoint[i] = 0;
        for (int j=0; j<3; j++)
            newpoint[i] += trt[i][j] *
                point[j];
    }
```

All that is left is to set *oledirection* to equal the new direction (this will insure that unnecessary rotation does not occur later) and to transfer the new point data into the *xpt* and *ypt* arrays (from which the bug will actually be drawn), using the index generated by the original for loop.

```

        xpt[p] = newpoint[0];
        ypt[p] = newpoint[1];
        oledirection = direction;
    }

```

In the event that the new direction of the bug is north, none of the above calculations are necessary and the simple transfer of all data points in arrays *xpto* and *ypto* to arrays *xpt* and *ypt* will suffice.

```

        else if (direction == 0) {
            for (int p=0; p<bugPoints; p++) {
                xpt[p] = xpto[p];
                ypt[p] = ypto[p];
            }
            oledirection = direction;
        }
    } // ***** end of calculateBugParts *****

```

The complete listing of *calculateBugParts* can be found in the complete listing of *Bug* at the end of this section.

Moving the Bug

Moving the bug to a new location is based on the speed setting (number of pixels to be advanced each move) and the direction of the bug. The method *go* performs the steps necessary to advance the bug. Method *go* is the second most complex method in *Bug*, but it uses many of the same ideas found in *calculateBugParts*, but only on the (x,y) pair that gives the location of the bug on the board.

As in *calculateBugParts*, the first step in *go* is to determine if the bug is moving north. If not, the new position must be calculated. If the direction of the bug is north, all that need be done is add the amount in speed to the y value of the (x,y) pair representing the bugs current location.

If the direction is not north, a number of steps must be performed. First, a matrix is created with the relative location of the bug assumed to be the pair (0,speed). This location can then rotated and the results added to the actual (x,y) pair of the bugs location.

```

    if (direction > 0) {

```

Here the matrix for the relative starting point is created and populated.

```
double cpts[] = new double[3];
cpts[0] = 0.0;
cpts[1] = (double)speed;
cpts[2] = 1.0;
```

Then the matrix to receive the results of the rotation is created.

```
double newpoints[] = new double[3];
```

The rotation value in radians is calculated.

```
double rotation = 6.28 * ((double) direction) /
                ((double)MAXDIRECTION);
```

The sine and cosine of the rotation value is calculated and inserted into the newly created rotation matrix.

```
double c = Math.cos(rotation);
double s = Math.sin(rotation);
double trt[][] = new double[3][3];
trt[0][0] = c;  trt[0][1] = -s;  trt[0][2] = 0;
trt[1][0] = s;  trt[1][1] = c;   trt[1][2] = 0;
trt[2][0] = 0;  trt[2][1] = 0;   trt[2][2] = 1;
```

Now the matrix multiplication for rotation can proceed to calculate the correct relative (x,y) pair.

```
double newpoint[] = new double[3];
for (int i=0; i<3; i++) {
    newpoint[i] = 0;
    for (int j=0; j<3; j++)
        newpoint[i] += trt[i][j] * cpts[j];
}
```

The resulting relative adjustments to the actual location of the bug can then be subtracted (because the screen is upside down!) from the actual location points.

```
rx -= newpoint[0];
ry -= newpoint[1];
}
```

Of course, if the direction of the bug is north, the only calculation needed is to subtract speed from the y value of the (x,y) location pair.

```
else if (direction == 0) ry -= (double)speed;
```

Finally, the adjusted (x,y) location of can be checked for boundary violations and adjusted if necessary, rounded and assigned to the integer variables that will be used to actually place the bug in the correct location on the screen.

```

    if ((rx+0.5) >= ((double)width))
        rx = (double)(width-1);
    else if (rx < 0.0)
        rx = 0.0;
    if ((ry+0.5) >= ((double)height))
        ry = double)(height-1);
    else if (ry < 0.0)
        ry = 0.0;

    x = (int)(rx+0.5);
    y = (int)(ry+0.5);
}

```

The Complete Listing of Bug

```

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

public class Bug {

    protected int x, y;
    protected double rx, ry;
    protected int color;
    protected int bugPoints=12;
    protected int speed;
    protected int width;
    protected int height;
    protected int direction;
    protected int oledirection;
    protected int MAXDIRECTION=16;
    protected int R=255, G=255, B=255;
    protected int impactzone = 5;
    protected boolean alive = true;

    protected double xpto[ ]
        = {2.0,2.0,5.0,5.0,2.0,1.0,-1.0,-2.0,-5.0,-5.0,-2.0, -2.0};
    protected double ypto[ ]
        = {0.0,3.0,2.0,6.0,5.0,9.0, 9.0, 5.0, 6.0, 2.0, 3.0, 0.0};
    protected double xpt[ ]
        = {2.0,2.0,5.0,5.0,2.0,1.0,-1.0,-2.0,-5.0,-5.0,-2.0, -2.0};
    protected double ypt[ ]
        = {0.0,3.0,2.0,6.0,5.0,9.0, 9.0, 5.0, 6.0, 2.0, 3.0, 0.0};
}

```

```
public Bug(int w, int h, int initx, int inity) {
    width = w;
    height = h;
    speed = 5;
    x = initx;
    y = inity;
    rx = x;
    ry = y;
    direction = 0;
    oledirection = 0;
    alive = true;
}

public void go( ) {
    if (direction > 0) {
        double cpts[ ] = new double[3];
        cpts[0] = 0.0;
        cpts[1] = (double)speed;
        cpts[2] = 1.0;

        double newpoints[ ] = new double[3];

        double rotation = 6.28 * ((double) direction) /
            ((double)MAXDIRECTION);
        double c = Math.cos(rotation);
        double s = Math.sin(rotation);

        double trt[ ][ ] = new double[3][3];
        trt[0][0] = c; trt[0][1] = -s; trt[0][2] = 0;
        trt[1][0] = s; trt[1][1] = c; trt[1][2] = 0;
        trt[2][0] = 0; trt[2][1] = 0; trt[2][2] = 1;

        double newpoint[ ] = new double[3];
        for (int i=0; i<3; i++) {
            newpoint[i] = 0;
            for (int j=0; j<3; j++)
                newpoint[i] += trt[i][j] * cpts[j];
        }
        rx -= newpoint[0];
        ry -= newpoint[1];
    } else if (direction == 0) ry -= (double)speed;

    if ((rx+0.5) >= ((double)width)) rx = (double)(width-1);
    else if (rx < 0.0) rx = 0.0;
    if ((ry+0.5) >= ((double)height)) ry = (double)(height-1);
    else if (ry < 0.0) ry = 0.0;

    x = (int)(rx+0.5);
    y = (int)(ry+0.5);
}

public int returnx( ) { return x; }
public int returny( ) { return y; }
```

```

public void setdirection(int d) {
    if (d >=0 & d < MAXDIRECTION) {
        direction = d;
        oledirection = -1;
    }
}

public void turnright( ) {
    oledirection = direction;
    direction++;
    if (direction >= MAXDIRECTION) direction = 0;
}

public void turnleft( ) {
    oledirection = direction;
    direction--;
    if (direction < 0) direction = MAXDIRECTION-1;
}

public void setBugColor(int r, int g, int b) {
    if (r<256 & r>=0) R = r;
    if (g<256 & g>=0) G = g;
    if (b<256 & b>=0) B = b;
}

public void paintComponent (Graphics g) {
    if (oledirection != direction) calculateBugParts( );
    paintBug(g);
}

public void calculateBugParts( ) {
    if (direction > 0) {
        double rotation = 6.28 * ((double) direction) /
            ((double)MAXDIRECTION);
        double c = Math.cos(rotation);
        double s = Math.sin(rotation);

        double trt[ ][ ] = new double[3][3];
        trt[0][0] = c; trt[0][1] = -s; trt[0][2] = 0;
        trt[1][0] = s; trt[1][1] = c; trt[1][2] = 0;
        trt[2][0] = 0; trt[2][1] = 0; trt[2][2] = 1;

        double point[ ] = new double[3];
        double newpoint[ ] = new double[3];
        for (int p=0; p<bugPoints; p++) {
            point[0] = xpto[p];
            point[1] = ypto[p];
            point[2] = 1;

            for (int i=0; i<3; i++) {
                newpoint[i] = 0;
                for (int j=0; j<3; j++)
                    newpoint[i] +=
                        trt[i][j] * point[j];
            }
            xpt[p] = newpoint[0];
            ypt[p] = newpoint[1];
        }
    }
}

```

```

        oledirection = direction;
    }
}
else if (direction == 0) {
    for (int p=0; p<bugPoints; p++) {
        xpt[p] = xpto[p];
        ypt[p] = ypto[p];
    }
    oledirection = direction;
}
}

public void paintBug(Graphics g) {
    Polygon p = new Polygon();
    for (int i=0; i<bugPoints; i++)
        p.addPoint(x+((int)(xpt[i]+0.5)),
                  y+((int)(ypt[i]+0.5)));
    Color mycolor = new Color(R,G,B);
    g.setColor(mycolor);
    g.fillPolygon(p);
}

public boolean isAlive( ){ return alive; }

public void isDead( ){ alive = false; }

public boolean impactBug(Bug p) {
    if (p.returnx( )<x+impactzone & p.returnx( )>x-impactzone &
        p.returny( )<y+impactzone &
        p.returny( )>y-impactzone) {
        return true;
    } else return false;
}
}
}

```

11. Testing Class Bug

In order to test class *Bug*, class *MainPanel* must be modified to include objects of class *Bug*. It would be nice if this test included some simple movement of the bugs. To accomplish this without more sophisticated animation techniques, a loop will be added to *paintComponent* of *MainPanel* that will cause the *Bug* objects move and be displayed in their new locations.

To give us a little more feedback as to what is happening while the program is running (we will only see the final image with a loop in *paintComponent*), we will make use of the command line output of Java. Methods *print* (outputs the results of an expression) and *println* (outputs the results of an expression then changes line) are both *static*. That is they can be called with out an instance variable object. These methods are located in package *System*, class *out*. They can be accessed by creating expressions of the following syntax:

```

System.out.print(expression);
System.out.println(expression);

```

Print and *println* are limited to expressions of basic types or class *String*.

By using calls to *print* to output the (x,y) coordinate pairs of one of the *Bug* objects, the program can report that object's position as the program runs.

Here is one possible version *MainPanel*. Notice the use of *break* to drop out of the *for* loop if a specific collision occurs. (Both *break* and *continue* work as they do in C++.)

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

public class MainPanel extends JPanel{

    private final int WIDTH;
    private final int HEIGHT;

    public MainPanel(int w, int h) {
        WIDTH = w;
        HEIGHT = h;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.black);
        g.fillRect(0, 0, WIDTH, HEIGHT);

        Bug p = new Bug(500,400,100,275);
        Bug q = new Bug(500,400,300,50);
        Bug r = new Bug(500,400,200,350);
        Bug s = new Bug(500,400,200,50);

        p.setBugColor(0,0,255);
        q.setBugColor(255,0,0);
        r.setBugColor(0,255,255);
        s.setBugColor(255,255,0);

        p.paintComponent(g);
        q.paintComponent(g);
        r.paintComponent(g);
        s.paintComponent(g);

        for (int j=0; j<8; j++) s.turnleft();

        for (int i=0; i<100; i++) {
            g.setColor(Color.black);
            g.fillRect(0, 0, WIDTH, HEIGHT);
            p.turnleft(); q.turnright();
            p.go(); q.go(); r.go(); s.go();
            p.paintComponent(g);
            q.paintComponent(g);
            r.paintComponent(g);
            s.paintComponent(g);
            System.out.print("p=(" + p.returnx() + "," +
                + p.returny() + ")\t");
            if (r.impactBug(s)) break;
        }
    }
}
```

```
public void display() {  
    repaint();  
}  
}
```

When *Bug.java* and *MainPanel.java* are compiled and *JavaByBugs.class* is run, t

```
javac Bug.java  
javac MainPanel.java  
javac JavaByBugs
```

the following is output:

