Fun with Java: Biomorphs and Artificial Life

Baldwin shows you how to write an artificial life program that models selective breeding, sometimes referred to as artificial selection.

Published: April 27, 2004
By Richard G. Baldwin

Java Programming, Lecture Notes # 1480

Preface

Programming in Java doesn't have to be dull and boring. In fact, it's possible to have a lot of fun while programming in Java. This lesson will concentrate on having fun while learning something at the same time.

In this lesson, I will show you how to write programs that model the selective breeding process, sometimes referred to as *artificial selection.* This is as opposed to natural selection, sometimes referred to as *survival of the fittest.*

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

Artificial life

In this lesson, I will present and explain a program that falls in the general category of *Artificial Life.* In the book with the same title, Christopher Langton tells us *"Artificial Life is the study of man-made systems that exhibit behavior characteristics of natural living systems."*

I will present and explain three separate programs *(three man-made systems).* The first two programs provide background information designed to help you understand the behavior of the more-important third program.

The third program exhibits behavior commonly referred to as *evolution.* In particular, this program exhibits the behavior of *artificial selection* (as opposed to *natural selection*). Artificial selection makes it possible to evolve creatures *(such as Dalmation dogs)* that accentuate certain desirable characteristics *(such as black spots on a white coat)* and suppress other less desirable characteristics *(such as a shaggy red coat).*

Natural selection versus artificial selection

For example, the variety of plants, animals, and birds that exist on an uninhabited island represent natural selection, sometimes referred to as *survival of the fittest.* A Dalmation dog, on the other hand, is probably the result of artificial selection. In other words, over a long period of time, people selected certain dogs for breeding to accentuate certain characteristics *(such as black spots on a white coat)* and to suppress other characteristics *(such as a shaggy red coat).* Over time, what resulted was a type of dog that we know as the Dalmation.

Those who participated in the process of artificial selection resulting in a Dalmation dog may not have known that those characteristics were represented by genes that were accentuated or suppressed through selective breeding. However, we know *(or at least believe)* that to be the case now.

Selectively breeding Biomorph objects

The third program that I will present and explain makes it possible for you to selectively breed successive generations of artificial creatures known as Biomorph objects.

> *(For brevity, I will refer to a Biomorph object as a Biomorph and will refer to multiple Biomorph objects as Biomorphs.)*

Each Biomorph is a recursively branching tree consisting of many branches *(stems)* of different lengths that branch off in different directions. For example, Figure 1 shows nine stages in the growth of a simple Biomorph.

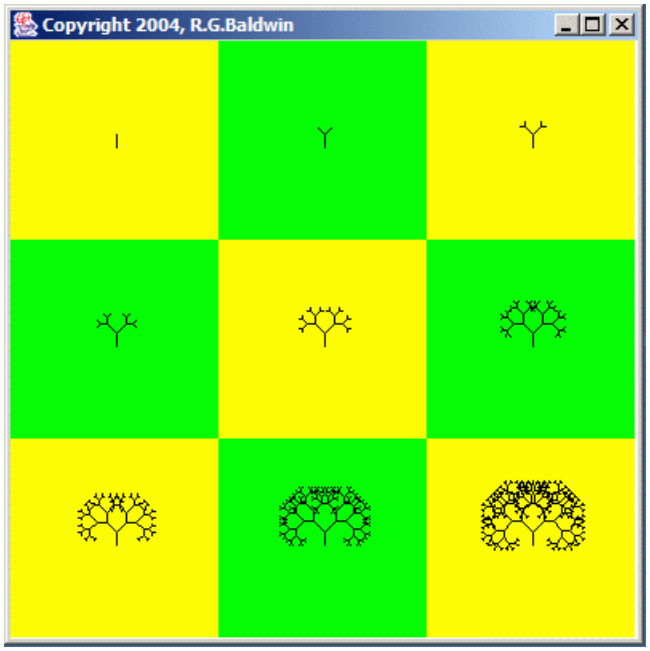

Figure 1 Nine stages in the growth of a Biomorph.

Biomorph genes

Each Biomorph has eight genes that control the size, the number, and the angle of the branches.

Reproduction

In the third program that I will present and explain, a single parent in each generation produces eight offspring in the next generation.

During the creation of each new generation of offspring, one of the genes for each of the eight offspring is randomly mutated to produce a creature that is similar to, but different from its parent. Thus each of the eight siblings in a single generation differs from the parent by the value of a single gene. Each of the eight siblings has a mutated value in a different gene, so no two offspring are exactly alike.

Stage versus generation

I probably need to explain the difference between the terms *stage* and *generation.* In human or plant terms, *stage* is somewhat analogous to age. For example, many plants grow a set of new branches during each growing season. A five year old tree, for example, will normally have more branches than a one year old tree. Similarly, a five-stage Biomorph will have more branches than a one-stage Biomorph, as illustrated in Figure 1.

However, many trees can produce a new generation of offspring at any age, or at least at any age beyond some minimum age. The same is true of Biomorphs. A Biomorph can become the

parent of a new generation of Biomorphs at any stage. There is no such thing as puberty in the word of Biomorphs.

Illustrating stage versus generation

Figure 1 illustrates the stages in the growth of a Biomorph.

Figure 2 illustrates Biomorph reproduction. The Biomorph in the lower right-most corner of Figure 2 was the parent of the eight other Biomorphs shown in Figure 2. Thus, the Biomorph in the lower right-most corner represents one generation and the eight offspring represent the next generation.

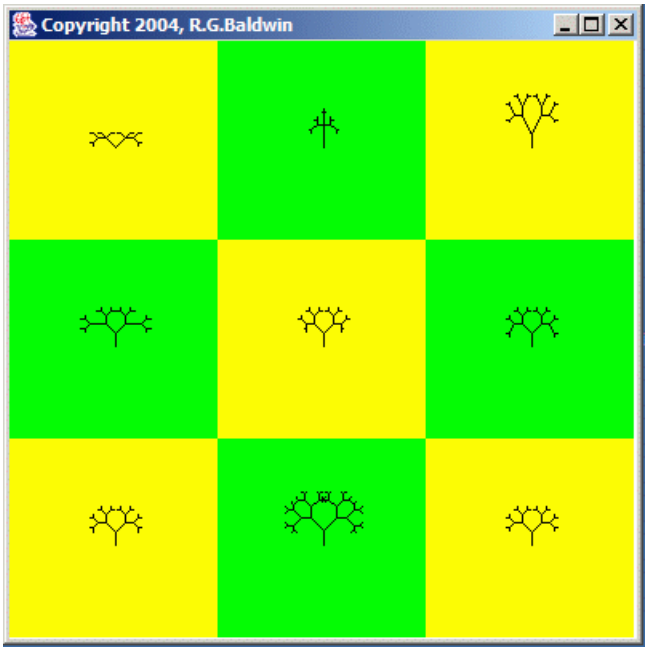

Figure 2 Illustration of Biomorph reproduction

Note how the complexity of the Biomorphs in Figure 1 progresses from least complex to most complex as you move from top left to lower right on a line-by-line basis. On the other hand, all of the Biomorphs in Figure 2 are of similar complexity, because they are all offspring of the Biomorph in the bottom right-most corner in Figure 2.

Artificial selection

In the third program that I will explain in this lesson, you can select one of the offspring from each new generation of Biomorphs to become the parent of the next generation. Through careful selection, you can accentuate certain characteristics of the Biomorph family and suppress other characteristics. By continuing this process through several generations, you can cause the resulting Biomorphs to resemble birds, bugs, animals, airplanes, human faces, or whatever strikes your fancy.

(Skipping ahead a bit, Figure 10 shows the result of using artificial selection to breed a family of Biomorphs that resemble stealth aircraft.)

Acknowledgement

This program is loosely based on Chapter 8 of the book entitled *Windows Hothouse* by Mark Clarkson. That chapter was based on a book and a paper published by Richard Dawkins. Dawkins' book is entitled *The Blind Watchmaker*. The paper was entitled *The Evolution of Evolvability* and appeared in the book entitled *Artificial Life.*

Discussion and Sample Program

As mentioned earlier, I will present and explain three different programs in this lesson. Only the third program supports the artificial selection process. The first two programs are designed to help you to understand the third program.

The Biomorph class

All three programs make use of a common class named **Biomorph**. Objects instantiated from this class are central to all three programs, so I will begin by presenting and explaining the **Biomorph** class.

This class is used to instantiate a Biomorph object. As mentioned earlier, it is based loosely on Chapter 8 of the book entitled *Windows Hothouse* by Mark Clarkson. However, the C++ algorithm presented in that book appears to contain several typographical errors. It was necessary for me to find and fix those errors when writing a Java version of the algorithm.

Construction of the Biomorph

The constructor for the **Biomorph** class receives an array of eight gene values. *(I will discuss the first seven values in the gene array later.)* The eighth value in the array specifies the number of stages used to construct the **Biomorph**.

(For example, the value of the eighth gene was 1 for the Biomorph shown in the top left-most corner of Figure 1. The value of the eighth gene was 9 for the Biomorph shown in the bottom right-most corner.)

Stages of growth

The Biomorph in the top left-most corner of Figure 1 shows the result of one stage of growth for a simple **Biomorph**.

The Biomorph in the bottom right-most corner of Figure 1 shows the result of nine stages of growth for the same simple **Biomorph**. For this Biomorph, the first seven gene values were all 1.0.

*(The gene values are essentially whole numbers, but I treated them as type **double** instead of type **int** to avoid the possibility of integer arithmetic problems.)*

Bifurcating stems

The first stage of construction produces a single stem as shown by the top left-most Biomorph in Figure 1. Each successive stage causes all existing stems to bifurcate into two new stems. You can clearly see this bifurcation process by examining the three Biomorphs in the top row of Figure 1. The left-most Biomorph is a single-stage Biomorph. The right-most Biomorph is a three-stage Biomorph, and the one is the center of the top row is a two-stage Biomorph.

You can also see this behavior in the fourth and fifth-stage Biomorphs in the two left-most positions in the center row. By the time you get to the six-stage Biomorph in the right-most position in the center row, however, the stems start to overlap and it is more difficult to visually distinguish the bifurcation process.

Increase by a power of two

Thus, the number of individual stems belonging to a Biomorph increases as a power of two based on the number of stages used in its creation. For example, a Biomorph created with two stages contains three stems as shown in the center of the top row in Figure 1. A three-stage Biomorph contains seven stems as shown by the right-most object in the top row of Figure 1. A four-stage Biomorph contains fifteen stems, etc.

(Those of you who are familiar with the binary number system will recognize the series 1, 3, 7, 15, 31, 63, etc. as being values that commonly arise in the binary number system. For example, a four-bit unsigned binary number can contain any of the values from 0 through 15 inclusive.)

Arithmetic accuracy

The algorithm in Clarkson's book is based on the use of integer gene values. However, when writing the Java version of the algorithm, I elected to maintain all of the data as type **double** in order to avoid the possibility of integer arithmetic problems, particularly when scaling the data for display. Values in my version of the algorithm are converted from **double** to **int** at the very last step before displaying the Biomorph on the screen.

Gene mutation

As you will see later when we examine the code for the **Biomorph** class, the constructor receives a random number generator and a count value that are used to mutate the genes.

One gene in the array of genes is mutated by a random value of plus or minus one whenever the count value is within the range from 0 through 7 inclusive. The actual gene that is mutated is the one whose position in the gene array matches the count value. If the count value is outside this range, there is no gene mutation.

Returning the mutated gene array

A method named **getGenes** returns the gene array containing the possibly mutated gene. This is useful for experiments in *artificial selection*, such as the third program that I will present and explain in this lesson.

Scaling the plot for display

The constructor receives a scale factor that is used to adjust the overall size of the picture for each individual Biomorph in an attempt to cause it to fit in the allocated plotting area.

Generally speaking, the size of the raw display of a Biomorph increases as the number of stages increases. When the scale factor is used later, coordinate values are multiplied by the reciprocal of the scale factor. Therefore, it is useful to cause the scale factor to increase as the number of stages increases. As you will see when you experiment with the second and third programs, sometimes even this is not sufficient to keep the size of an individual Biomorph within the allocated area.

Positioning the plot

The constructor receives a pair of **int** values that are used to move the plotting origin from the default upper-left corner to another location in the plotting area. All three programs presented in this lesson position the plotting origin at the center of each of the nine individual plotting areas shown in Figure 1.

Direction of the first stem

The direction of the first stem displayed for the Biomorph is hard-coded to be vertical going up the screen, starting at the origin.

Beginning of the Biomorph class

As is my custom, I will discuss these three programs in fragments. You will find complete listings of the programs in Listing 31, Listing 32, and Listing 33 near the end of the lesson.

The code fragment in Listing 1 shows the beginning of the **Biomorph** class used in all three programs.

```
class Biomorph extends Panel{
  double[] xInc = new double[8];
  double[] yInc = new double[8];
  double[] genes;
  double xCoor = 0;//Start drawing
here
  double yCoor = 0;//Start drawing
here
  int direction = 0;//Initial drawing
direction
```

```
  double length;
  double scale;
  int xOrigin;
  int yOrigin;

Listing 1
```

Listing 1 simply declares some instance variables, initializing some of them. The purpose of these variables will become clear in the discussion that follows.

Constructor for the Biomorph class

The constructor begins in Listing 2. The purpose of each of the constructor parameters was discussed in the preceding paragraphs.

```
  Biomorph(double[] genes,
           Random rGen,
           int cnt,
           double scale,
           int xOrigin,
           int yOrigin){

    this.genes =
(double[])genes.clone();
    this.scale = scale;
    this.xOrigin = xOrigin;
    this.yOrigin = yOrigin;

Listing 2
```

Save local copies

The code in Listing 2 saves copies of four of the incoming parameters in instance variables. These are parameter values that require access within the class but outside the constructor.

Cloning the gene array

Note in particular that the **clone** method is used to save a copy of the incoming array of gene values. This is necessary to prevent the code from changing gene values in the original gene array passed to the constructor. In some cases, the same gene array will be used for constructing multiple objects of the **Biomorph** class, and the construction of any one of those objects cannot be allowed to modify the values stored in the original gene array.

Mutate a gene

Listing 3 mutates one of the genes stored in the local copy of the gene array, provided that the value of **cnt** is within the range from 0 through 7 inclusive.

```
    if((cnt>=0) && (cnt<=7)){
      double mutantValue =
rGen.nextInt(2)*2-1;
      this.genes[cnt] += mutantValue;
      //Don't allow the eighth gene to
go
      // negative
      if(this.genes[7] <
0)this.genes[7] = 0;
    }//end if
```

Listing 3

The particular gene that is mutated is specified by the value of **cnt**. If **cnt** is outside the specified range, no gene mutation takes place.

Mutation changes the gene value

When mutation does take place, the value of the gene is increased or decreased by a value of 1.0. The determination as to whether to increase or decrease the value is based on a random number generator. You can examine the documentation for the library class named **Random** if need be to understand how this code works.

The code in Listing 3 also ensures that the value of the eighth gene cannot go negative. The idea of instantiating a Biomorph where the number of stages is negative doesn't make any sense.

The first seven gene values

This is where things get somewhat complicated. In order for you to understand the remaining code in the constructor, I need to explain the meaning of the first seven values in the array of genes. For that, I will turn to a discussion of the physical drawing process for the Biomorph.

Branching off in different directions

Consider the rightmost Biomorph in the top row of Figure 1 as an example. When the time comes for a stem to bifurcate into two new stems, they branch off in different and somewhat opposite directions.

Eight possible directions

A new stem can branch off in any one of eight different directions. Two of the eight directions are horizontal to the right or to the left. Two of the directions are vertical, either up or down. That takes care of four of the eight possible directions.

The other four directions are generally neither horizontal nor vertical. Considering the first four directions as representing north, south, east, and west, the other four directions are generally northeast, northwest, southeast, and southwest. I say generally because these directions are not fixed like points on the compass. For example, a stem that branches off in a northeasterly

direction can be horizontal, vertical, or anything in between. In other words, the actual direction of a stem that branches off in a northeasterly direction can be any direction in the ninety degrees between pure north and pure east.

Genes determine the direction of a new stem

The combined values of a subset of the first seven genes determine the directions that the two new stems take when the old stem bifurcates.

For brevity in the following discussion, I will use the following notation to represent each of the first seven values in the gene array: g0, g1, g2, g3, g4, g5, and g6.

Defined by two pairs of coordinates

As you will see when we examine the code later, each new stem is defined by two pairs of coordinates. One pair of coordinates represents the starting point of a straight line, which is the same as the point in two-dimensional space where the old stem bifurcates.

The second pair of coordinates represents the other end of the stem in the same two-dimensional space.

Compute second pair of coordinate values

The location of the other end of the stem is computed by selecting a pair of values from the gene array and scaling those values.

> *(In some cases, one of the coordinate values is zero, in which case it isn't necessary to select a value from the gene array.)*

The first seven values in the gene array are used to determine the end points of every stem.

Relationship between gene values and coordinate values

Figure 3 shows my attempt to represent this relationship in visual form. *(As you can see, I am not a graphics artist.)*

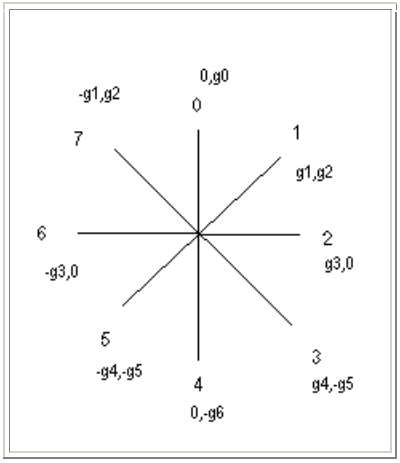

Figure 3 Eight possible stem directions

Figure 3 shows each of the eight possible directions in which a stem can be drawn. Those eight directions are numbered from 0 through 7, starting at the top and going in a clockwise direction.

Figure 3 also shows how values from the gene array are used to specify the end point of a stem to be drawn in any of those eight possible directions.

> *(For example, the end point of a stem in direction 1 is determined by applying a scale factor to values in the gene array represented by g1 and g2. You will see that this is genes[1] and genes[2].)*

Horizontal and vertical stems

As you can see, the gene values of g0, g3, g6 and -g3 produce horizontal and vertical stems of different lengths along directions 0, 2, 4, and 6.

Note that the same absolute value from the gene array, g3, is used to specify the end of a stem going in either direction along the horizontal axis *(direction 2 and direction 6).* Only the algebraic sign of the value differs.

Also note that the same absolute values from the gene array, g1, and g2, are used to specify the end of a stem drawn in either direction 1 or direction 7. Again, only the algebraic sign applied to the value of g1 differs between the two directions.

Finally, note that the same absolute values from the gene array, g4, and g5, are used to specify the end of a stem drawn in either direction 3 or direction 5. In this case, a negative sign is applied to g5 for both directions 3 and 5. A negative sign is applied to g4 for direction 5.

Horizontal symmetry

The result is that all Biomorphs are horizontally symmetrical, as shown in Figure 1 and Figure 2.

Only seven different gene values are required

A close examination of Figure 3 shows that only seven different gene values are required to specify the end point of any stem regardless of its direction.

As the genes mutate and change values, the lengths of horizontal and vertical stems constructed using those gene values also change.

Both the length and the angle relative to the horizontal changes for stems drawn along directions 1, 3, 5, and 7 as the genes mutate and change values.

Effect of changes in gene values

The effect of changes in gene values on the shape of the Biomorphs is illustrated in Figure 4. This figure shows the generation and display of Biomorphs through a controlled set of changes to the values in the gene array.

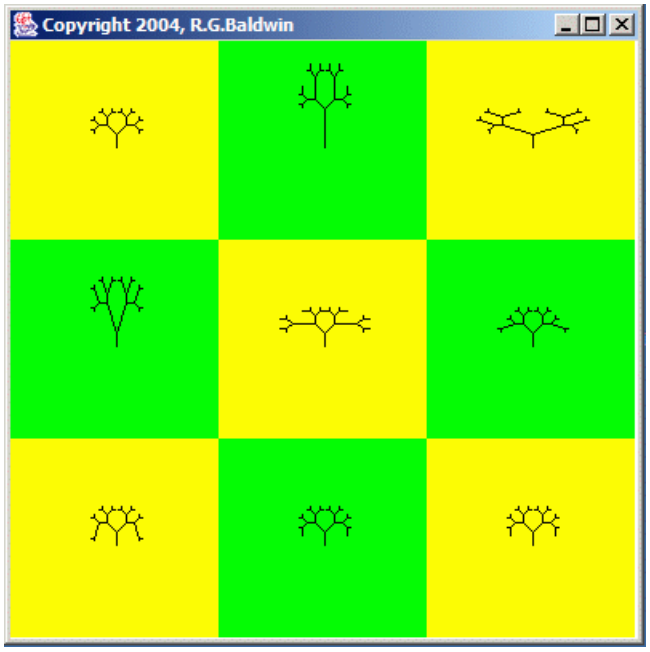

Figure 4 Biomorphs with controlled changes to the genes.

The baseline Biomorph

The top left-most Biomorph in Figure 4 was generated using a gene array with all ones in the first seven genes and a five in the eighth gene. Thus, it is a five-stage Biomorph.

Changes in the gene values

The next seven Biomorphs, beginning with the Biomorph in the center of the top row, were generated by multiplying one, and only one of the genes by a factor of three, starting with the first gene and ending with the seventh gene.

> *(The ninth Biomorph was a duplicate of the eighth Biomorph, and was included only to balance out the display.)*

Correlate pictures with gene values

You should be able to correlate the shapes of the Biomorphs in Figure 4 with the information in Figure 3. This should help you to understand why the Biomorphs in Figure 4 look the way that they do.

Increasing the value of the first gene

For example, when the value of the first gene was multiplied by a factor of three, this had the effect of increasing the value of g0 at the top of Figure 3, thus elongating the Biomorph in the vertical direction. *(See the center Biomorph in the top row of Figure 4.)*

Increasing the value of the second gene

When the second gene was multiplied by a factor of three, this had the effect of increasing the value of g1 without increasing the value of g2 in Figure 3. This tended to stretch the Biomorph horizontally for those stems that had a northeasterly and northwesterly direction. *(See the right-most Biomorph in the first row in Figure 4.)*

> *(For the record, Figure 4 was produced by an unpublished program named Biomorph01x1. I mention this here just in case I need to go back and retrieve the program later.)*

Back to the code

With that as background information, it is now time to return to an explanation of the code in the constructor for the **Biomorph** class.

Recall that the shorthand notation g0, g1, g2, etc., in Figure 3 actually represents values in the gene array given by **genes[0]**, **genes[1]**, **genes[2]**, etc.

Create and populate two arrays

The code in Listing 4 populates two arrays referred to by **xInc** and **yInc** on the basis of the values stored in the gene array named **genes**.

The values in the new array referred to by **xInc** represent the horizontal components of the outer ends of the eight lines shown in Figure 3.

The values in the new array referred to by **yInc** represent the vertical components of the outer ends of the eight lines shown in Figure 3.

The coordinates of the ends of the lines

Taken together, the sixteen values in the two new arrays specify the coordinates of the outer ends of each of the eight lines shown in Figure 3.

```
    xInc[0] =  0;
                          yInc[0] =
this.genes[0];
    xInc[1] =  this.genes[1];
                          yInc[1] =
this.genes[2];
    xInc[2] =  this.genes[3];
                          yInc[2] =  0;
    xInc[3] =  this.genes[4];
                          yInc[3] = -
this.genes[5];
    xInc[4] =  0;
                          yInc[4] = -
this.genes[6];
    xInc[5] = -this.genes[4];
                          yInc[5] = -
this.genes[5];
    xInc[6] = -this.genes[3];
                          yInc[6] =  0;
    xInc[7] = -this.genes[1];
                          yInc[7] =
this.genes[2];
```

Listing 4

Array indices match direction numbers

The values stored at index 0 in the two new arrays specify the end of the line along direction 0 in Figure 3.

Similarly, each of the direction numbers in Figure 3 matches a corresponding index value in the two new arrays.

The coordinate values stored in the two new arrays will be used later, along with a multiplicative scale factor, to compute the coordinates of the end points of new stems in the Biomorph object.

The lengths of the stems

A careful examination of Figure 1 reveals that the length of each new pair of stems is shorter than the length of the stem that spawned them. Listing 5 shows the end of the constructor for the **Biomorph** class.

```
    length = this.genes[7];
```

```
  }//end constructor

Listing 5
```

The code in Listing 5 establishes the length of the first stem belonging to the Biomorph.

This initial line length is based on the number of stages to be drawn in the construction of the Biomorph. You will see later that the length of each new stem is reduced by a value of one relative to the length of the stem that spawned it. The algorithm terminates when the length of the stem reaches zero.

> *(The algorithm actually terminates after the last stem of length is one is drawn. There is no point in attempting to draw a stem with zero length.)*

The getGenes method

As I mentioned earlier, a Biomorph has the ability to return a reference to its *(potentially mutated)* gene array.

```
  double[] getGenes(){
    return this.genes;
  }//end getGenes

Listing 6
```

The method to accomplish this is very simple, and is shown in Listing 6. I wanted to get that out of the way before getting into the more complex topic of the overridden **paint** method.

The overridden paint method

Screen graphics are created in Java by overriding the method named **paint**. When time comes to redraw the screen, the Java virtual machine, in conjunction with the operating system causes the overridden **paint** method to be invoked. The code written into the overridden **paint** method determines what gets drawn.

The overridden **paint** method is shown in Listing 7.

```
  public void paint(Graphics g){
    g.translate(xOrigin,yOrigin);

drawIt(g,xCoor,yCoor,length,direction,xInc,

yInc,scale);
  }//end paint()

Listing 7
```

The method appears to be very simple, but looks can be deceiving.

Adjust the plotting origin

The overridden **paint** method starts out simple enough. Recall that a Biomorph is an object that extends the class named **Panel**. Thus, the code in Listing 7 overrides the **paint** method inherited from the **Panel** class.

By default, all coordinate values in Java are relative to the upper left-most corner of the component in which the coordinates are being determined. In other words, the plotting origin for a **Panel** object is the upper left-most corner of the panel.

The first statement in Listing 7 translates the plotting origin to a point near the center of the panel. You will see later that the actual coordinate values of the new origin are based on the overall size of the GUI and the number of panels placed in the GUI.

Invoke the drawIt method

It is the second statement in Listing 7 that is deceivingly simple. This statement invokes the method named **drawIt** to cause the new Biomorph to be drawn on the screen. Some of you may find the **drawIt** method to be somewhat complex.

Recursion

If you are skilled in the use of recursion, you will probably find the **drawIt** method to be relatively straightforward. However, the method is invoked recursively to draw the Biomorph, and those of you who are not skilled in the use of recursion may find it more difficult.

The Graphics object

The **paint** method always receives a reference to an object of the **Graphics** class. As a practical matter, you can think of this object as representing the screen on which you are going to draw a picture. When you draw a picture on the **Graphics** object, it appears on the portion of the screen that belongs to your Java application.

As you can see in Listing 7, the reference to the **Graphics** object is passed as the first parameter to the **drawIt** method, giving that method the capability to draw pictures on the computer screen.

The drawIt method

The signature for the **drawIt** method is shown in Listing 8. As mentioned above, the first parameter is a reference to a **Graphics** object that gives the method the ability to draw pictures on the computer screen.

```
  void drawIt(Graphics g,
              double oldX,
              double oldY,
              double len,
              int newDir,
```

```
                double[] xInc,
                double[] yInc,
                double scale){

Listing 8
```

The oldX and oldY parameters

The Biomorph is actually constructed while it is being drawn in the **drawIt** method. The construction of the Biomorph consists of the definition of the end points of the stems that make up the Biomorph. Each of those stems is a straight line segment.

The **drawIt** method is invoked recursively to draw each stem.

The parameters named **oldX** and **oldY** specify the coordinates of the starting point of the stem. After the first call to the **drawIt** method, each recursive call to the method passes the coordinates of the end point of the current stem in these two parameters. That causes the end of one stem to become the starting point for the two stems spawned by that stem.

The length and direction of the line segment

As mentioned earlier, the length of the line that represents each new stem is reduced by one relative to the length of the stem that spawned it through the bifurcation process. Also, as mentioned earlier, each stem in the pair of new stems spawned in the bifurcation process go off in generally opposite directions.

The fourth and fifth parameters in the **drawIt** method signature contain the length and direction of the new stem.

The direction is specified as an integer according to the direction numbers shown in Figure 3.

The end-point arrays

The sixth and seventh parameters are references to objects that encapsulate the arrays containing the end points for lines in each of the eight possible directions.

Recall that the coordinate values stored in these two array objects were computed by the code in Listing 4 based on the values stored in the gene array. Like the gene values, the end-point values do not change during the recursive construction of a Biomorph.

The scale factor

The last parameter shown in Listing 8 is a scale factor that is applied to the drawing in an attempt to cause the size of the drawing to fit within its allocated space. The overall size of a Biomorph tends to increase as more stages are used to construct it. Stated differently, the overall size tends to increase as more and more stems are added.

As you will see later, the programs in this lesson cause the scale factor to increase as the number of stages increase. The reciprocal of the scale factor is used to scale the overall size of the drawing. This is an attempt to cause the size of the Biomorph to be appropriate for the allocated space regardless of the number of stages used in its construction.

Constraining the direction values

The new direction value passed to the **drawIt** method is computed by adding a positive or negative integer to the direction value for the stem that spawned it. Left unchecked, this could result in negative direction values, or direction values that exceed the allowable positive limit of 7.

The code in Listing 9 constrains the direction value to the range from 0 through 7 inclusive. Hopefully you understand enough about fundamental Java programming that the code in Listing 9 won't be a mystery.

```
    newDir = (newDir + 8)%8;
```

Listing 9

Compute end points of the new stem

The code in Listing 10 computes the coordinates of the end point of the new stem.

```
    double newX = oldX + len *
xInc[newDir];
    double newY = oldY + len *
yInc[newDir];
```

Listing 10

The stem is drawn as a straight line segment that begins at the coordinates given by the parameters **oldX** and **oldY**, and ends at the coordinates computed in Listing 10.

The genes determine the shape of the Biomorph

The values of the new coordinates are the product of the length of the new stem *(received as an incoming parameter),* and the coordinate values stored in the two arrays discussed earlier.

The value of the new direction is used as an index to retrieve the coordinate values from the two arrays. Since the coordinate values in the two arrays was computed earlier on the basis of the values of the genes, the values of the genes and the length of the stem determine the coordinates of the end of the new stem. This explains how the values of the genes ultimately determine the shape of the Biomorph, one stem at a time.

Draw the current stem

The code in Listing 11 invokes the **drawLine** method of the **Graphics** class to actually draw the stem on the computer screen. If you are unfamiliar with this method, you can look it up in the Sun documentation.

```
    g.drawLine((int)(oldX/scale),
               (int)(-oldY/scale),
               (int)(newX/scale),
               (int)(-newY/scale));

Listing 11
```

Correct for vertical direction

By default, positive vertical values are drawn going down the screen. This is backwards to what most of us are comfortable with in a Cartesian coordinate system where positive vertical values normally go up.

The minus signs in Listing 11 correct for this situation causing positive vertical values to be drawn going up the screen.

Make a recursive call to the drawIt method

Having drawn one stem, it is time to make two recursive calls to the **drawIt** method to implement the bifurcation process and to cause the two stems that result from that process to be drawn. This is accomplished by the code in Listing 12.

```
    if(len > 1){
      drawIt(g,newX,newY,len-
1,newDir+1,xInc,

yInc,scale);
      drawIt(g,newX,newY,len-1,newDir-
1,xInc,

yInc,scale);
    }//end if
  }//end drawIt

Listing 12
```

The length of the new stem

Each time a recursive call is made to the **drawIt** method, the length of the stem to be drawn is reduced by one. The code in Listing 12 causes the stems to continue bifurcating and drawing new stems for as long as the current length is greater than 1.

When the length of the current stem reaches 1, the **drawIt** method returns without bifurcating and drawing any more new stems.

(It wouldn't make any sense to try to draw a stem whose length is zero.)

That is what terminates the recursion process.

The starting point for the new stem

Note that the coordinates of the end point for the current stem are passed to the **drawIt** method where they become the coordinates for the starting point of the new stem.

The directions of the new stems

The only difference between the two recursive calls to the **drawIt** method in Listing 12 has to do with the direction parameter. For the first call to the **drawIt** method, the new direction is increased by one relative to the current direction. For the second call to the **drawIt** method, the new direction is decreased by one relative to the current direction.

Referring back to Figure 3, for example, we see that if the direction number for the current stem is 2, the direction number for one of the new stems will be 3 and the direction number for the other new stem will be 1.

Similarly, referring both to Figure 3 and Listing 9, we see that if the direction number for the current stem is 0, the direction number for one of the new stems will be 2 and the direction number for the other new stem will be 7.

Recursive behavior

A complete explanation of recursive behavior is beyond the scope of this lesson. However, it might be useful to provide a visual illustration of recursion.

The recursive behavior provided by Listing 12 is quite complex due to the fact that two successive statements make recursive calls to the **drawIt** method. That behavior can be greatly simplified by modifying the code in Listing 12 to that shown in Listing 13.

```
    if(len > 1){
      drawIt(g,newX,newY,len-
1,newDir+1,xInc,

yInc,scale);
/*
      drawIt(g,newX,newY,len-1,newDir-
1,xInc,

yInc,scale);
*/
    }//end if
  }//end drawIt
```

Listing 13

The second recursive call to the **drawIt** method has been disabled in Listing 13 by turning it into a comment.

The simplified output

When the program that produced the output shown in Figure 1 is modified in the manner shown in Listing 13 and then rerun, the new simplified output is as shown in Figure 5.

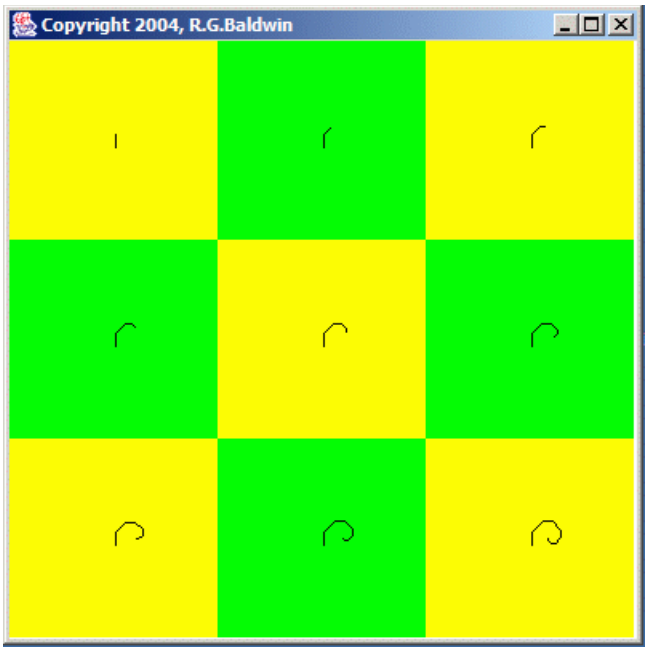

Figure 5 Nine stages in the growth of a simplified Biomorph

Compare Figure 5 with Figure 1

If you compare Figure 5 with Figure 1, you will see that the output in each of the nine drawing areas consists only of the successive recursively generated stems for which the new direction value is one greater than the old direction value.

You will also see that each stem is shorter than the one that spawned it, and recursion terminates when the length of the current stem reaches a value of one.

> *(Recall that the overall plotting scale factor applied to the drawing in the upper left-most position is greater than the scale factor applied to the drawing in the lower right-most position. Thus a stem length of one unit in the upper left-most position is longer than a stem length of one unit in the lower right-most position.)*

End of the Biomorph class

The code in Listing 12 signals the end of the class named **Biomorph**, from which our Biomorph objects are instantiated.

Now that you understand the **Biomorph** class, you should have little trouble understanding the three programs shown in Listing 31, Listing 32, and Listing 33 near the end of the lesson. All three of the programs use the **Biomorph** class to create Biomorph objects. The difference in the three programs lies in how they manage the Biomorph objects.

The program named Biomorph01

As usual, I will explain all three of the programs in fragments. A complete listing of the program named **Biomorph01** is presented in Listing 31 near the end of the lesson.

The purpose of the program named **Biomorph01** is to compute and display the first nine stages of growth for a Biomorph based on a simple gene set where each of the seven genes that control the shape of the Biomorph have a fixed value of 1.

The program begins in Listing 14, which declares and initializes two static member variables. The variables are declared static to make it possible to refer to them from the static **main** method.

```
public class Biomorph01{
  static double[] genes = new
double[8];

  static Random rGen =
                new Random(new
Date().getTime());
```

Listing 14

The first member variable is used later to store the eight genes required by the **Biomorph** class. As you learned earlier, the first seven genes control the shape of the Biomorph, while the eighth gene controls the number of bifurcating stages used to construct the Biomorph.

The second member variable provides a random number generator. As you learned earlier, this random number generator is required by the **Biomorph** class. However, it isn't actually used by this program.

The main method

The **main** method is shown in its entirety in Listing 15.

```
  public static void main(String[]
args){
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt]=1;
    }//end for

    genes[7] = 1;
```

```
    new GUI(genes,rGen);
  }//end main

Listing 15
```

The code in the **main** method performs the following tasks:

- Populate the gene array with seven fixed gene values of 1.
- Specify the number of stages used to construct the first Biomorph.
- Instantiate an object of the GUI class that will take care of the remaining tasks.

This is all relatively straightforward and shouldn't require further explanation.

Listing 15 also signals the end of the class named **Biomorph01**.

The class named GUI

This class is used to instantiate a graphical user interface object that causes the first nine stages of a simple Biomorph to be created and displayed in nine grid cells in a **Frame** object. Figure 1 shows the output produced by this class.

The beginning of the **GUI** class is shown in Listing 16.

```
class GUI extends Frame{
  Random rGen;
  double[] genes;

Listing 16
```

Listing 16 declares instance variables used to store references to the gene array object and the random number generator object.

The GUI constructor

The constructor for the **GUI** class begins in Listing 17.

```
  public GUI(double[] genes,Random
rGen){
    this.rGen = rGen;
    this.genes = genes;

Listing 17
```

The code in Listing 17 saves the incoming parameters in the instance variables declared in Listing 16.

Set the layout manager

Listing 18 replaces the default layout manager for the **Frame** object with a layout manager that subdivides the frame into nine grid cells of equal size.

```
    setLayout(new GridLayout(3,3));

Listing 18
```

See Figure 1 for an example of the new layout.

Create and display nine Biomorphs

Listing 19 shows the beginning of a **for** loop that creates and displays the nine stages of growth for a Biomorph using the same genes for each stage.

```
    for(int cnt = 0; cnt < 9; cnt++){
      Biomorph biomorph =
                          new
Biomorph(genes,

rGen,
                                    -
1,

genes[7]/8,

66,

66);

Listing 19
```

> *(Actually, the **for** loop creates and displays nine different Biomorphs, with each successive Biomorph being constructed with one more stage than the previous Biomorph.)*

You already know all about the **Biomorph** class, so the only discussion that should be needed is a discussion of the parameters to the **Biomorph** constructor.

Biomorph constructor parameters

The first two parameters pass the gene array and the random number generator to the **Biomorph** constructor.

The third parameter value of -1 causes the value of **cnt** to be *out of range* so that the code in the body of the **if** statement in Listing 3 is not executed. As a result, no mutation of the genes takes place.

The scale factor, which is passed as the fourth parameter, is proportional to the number of stages used to construct the Biomorph. As described earlier, this is an attempt to cause each individual Biomorph shown in Figure 1 to be appropriate for the allocated drawing area for that Biomorph.

Finally, the last two parameters cause the plotting origin to be placed in the center of the allocated drawing area for each Biomorph.

Setting the background colors

An object of the class **Panel** doesn't have a border of any kind. Therefore, when two or more **Panel** object are drawn adjacent to one another, it isn't possible to tell where on ends and the other begins unless something is done to cause the boundary between the two to be visible.

As shown in Figure 1, the code in Listing 20 causes the background colors of adjacent Biomorphs to alternate between yellow and green. This makes it easy to recognize the boundary between two Biomorphs.

```
      if(cnt%2 == 0){

biomorph.setBackground(Color.YELLOW);
      }else{

biomorph.setBackground(Color.GREEN);
      }//end else

Listing 20
```

Add the new Biomorph to the Frame

Recall that we are still discussing the body of the **for** loop that began in Listing 19. The code in Listing 21 adds the new Biomorph that was constructed in Listing 19 to the next grid cell on the frame.

```
      this.add(biomorph);

Listing 21
```

Increase the number of stages

Listing 22 increments the eighth gene in the gene array. This causes the number of stages that will be used to construct the next Biomorph to be one greater than the number of stages that were used to construct the current Biomorph.

```
      genes[7] += 1;
    }//end for loop

Listing 22
```

Listing 22 also signals the end of the **for** loop that began in Listing 19.

Complete the GUI class definition

Listing 23 shows the remaining code in the definition of the class named **GUI**. This code is completely straightforward and shouldn't require an explanation. It is included here only for completeness.

```
    setTitle("Copyright 2004,
R.G.Baldwin");
    setSize(400,400);
    setVisible(true);

    //Instantiate and register a
Listener object
    // that will terminate the program
when the
    // user closes the Frame
    addWindowListener(
      new WindowAdapter(){
        public void
windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

}//end class GUI definition

Listing 23
```

The program named Biomorph02

Listing 32 near the end of the lesson presents a complete listing of the program named **Biomorph02**.

The purpose of this program is to compute and display the first nine stages of growth for a Biomorph based on a complex gene set where each of the seven genes that control the shape of the
Biomorph are obtained from a random number generator.

This program should produce Biomorph objects having different appearances each time it is run.

The main method

The **main** method for this program is shown in Listing 24.

```
  public static void main(String[]
args){
```

```
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt] = rGen.nextInt(7)-3;
    }//end for

    genes[7] = 1;

    new GUI(genes,rGen);
  }//end main
```

Listing 24

The only real difference between this program and the program named **Biomorph01** is the code in the **main** method that is highlighted in red boldface in Listing 24.

Recall that the first seven gene values in the program named **Biomorph01** had a value of 1. This program, on the other hand, uses a random number generator to create those seven gene values. The purpose is to illustrate the variety of ways that the Biomorphs differ when the gene values differ. To see those differences, simply compile this program and run it several times in succession. Each time you run it, you should see a different set of Biomorphs.

Sample outputs

Three sample outputs from this program are shown in Figure 6, Figure 7, and Figure 8.

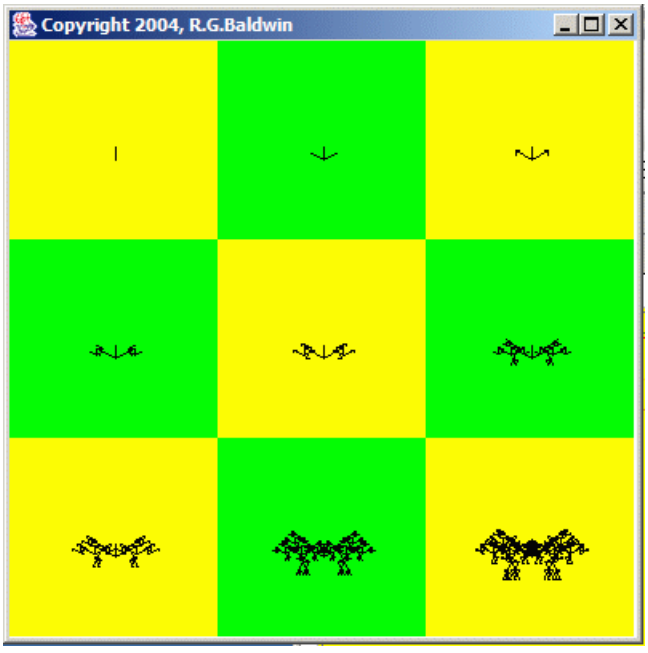

Figure 6 Biomorphs based on random gene values

I don't know what you think, but the Biomorph in Figure 6 looks remarkably like an eagle to me.

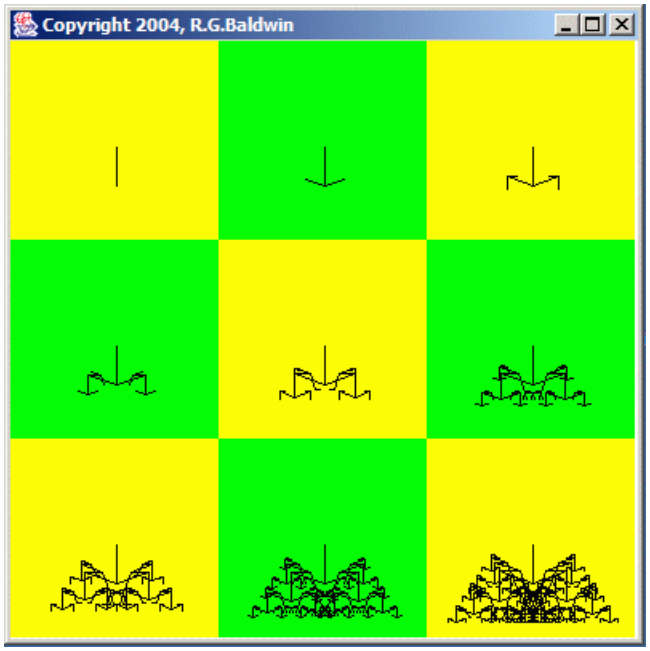

Figure 7 Biomorphs based on random gene values

The Biomorph in Figure 7 reminds me of a *"daddy longlegs"* spider hanging from a single strand of its web.

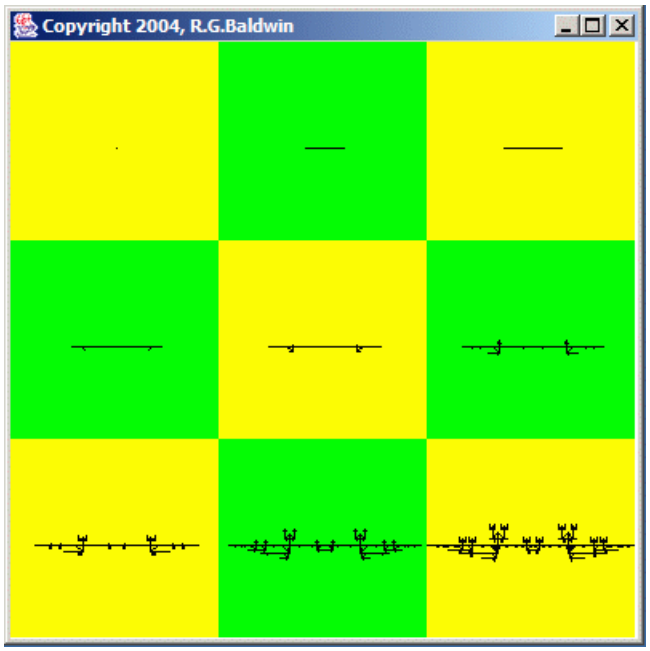

Figure 8 Biomorphs based on random gene values

The Biomorph in Figure 8 makes me think of looking across a pond and seeing the reflection of trees or buildings in the pond.

No artificial selection was involved

No artificial selection was involved in these three samples. These images were produced simply by setting the gene values at random and using that gene set to produce the Biomorph. I will get into artificial selection in the next section.

The program named Biomorph03

This is the program that provides the *artificial selection* capability described in the Preview section of this lesson.

The behavior of the program has been described in general terms in previous sections of this lesson. To use artificial selection to breed a Biomorph having desirable characteristics, simply run the program and select the Biomorph that you consider to be *"best"* with the mouse. That Biomorph will become the parent of the next generation of Biomorphs. The parent will appear in the lower left-most cell in the display and the siblings in the new generation will occupy the other eight cells.

Repeat that process until you have bred a generation of Biomorphs that match your desired characteristics.

Some results

I will explain the inner workings of the program in the remainder of this lesson.

Before getting into the technical details, let's take a look at some results, as shown in Figure 9 and Figure 10.

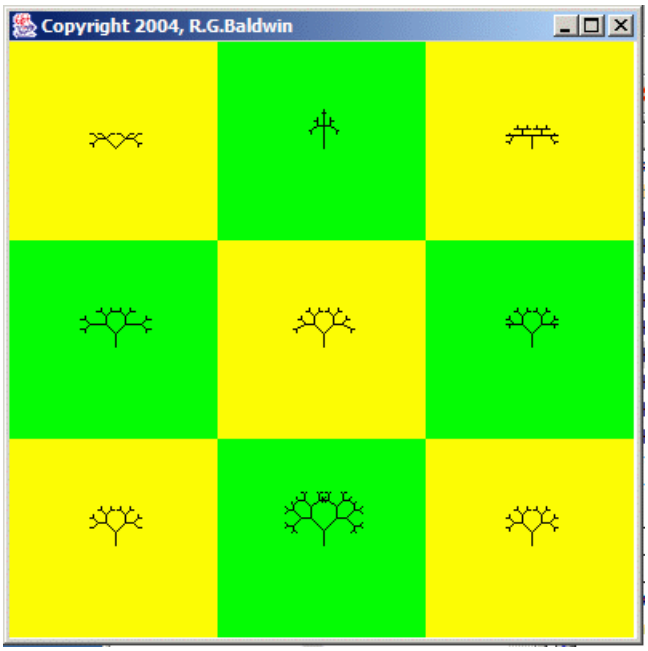

Figure 9 Starting point for *artificial selection* process

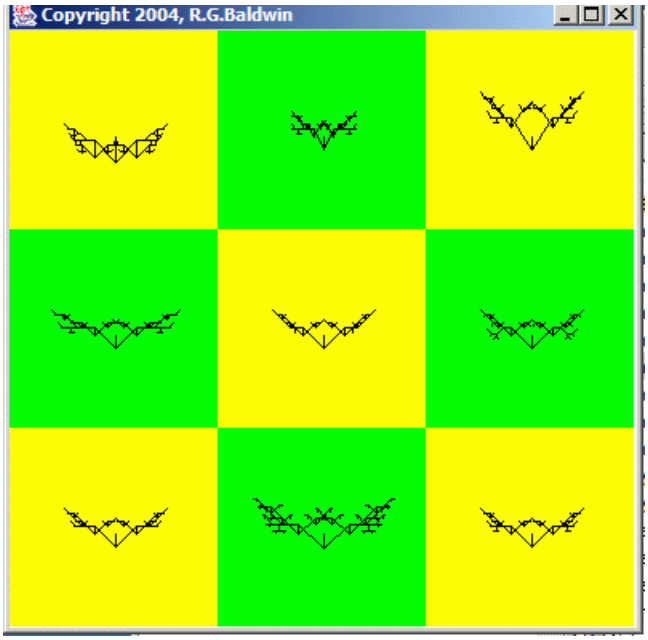

Figure 10 Stealth aircraft produced through *artificial selection*

The starting point

The bottom right-most Biomorph in Figure 9 is the original parent Biomorph provided by this program. The other eight Biomorphs shown in Figure 9 are the eight offspring that constitute the first new generation. Each of the offspring differs from the parent in terms of the value of one gene. Each of the offspring has a mutated value in a different gene, so no two offspring are exactly alike.

As you can see, some of the offspring strongly resemble the parent, while others have little resemblance to the parent. It all depends on which gene was mutated.

Several generations later

Figure 10 shows a parent and eight offspring produced about nine or ten generations later.

After a couple of clicks, I noticed something developing that looked a little like a stealth aircraft. I decided to emphasize that characteristic, and after a few more clicks, the Biomorphs that you see in Figure 10 had evolved. If you use your imagination, most of the Biomorphs in Figure 10 look something like a stealth aircraft.

Will discuss in fragments

I will discuss the program named **Biomorph03** in fragments. A complete listing of the program is shown in Listing 33 near the end of the lesson.

Much of the code in this program is very similar to code in one or the other of the two programs discussed earlier in this lesson. I won't repeat a discussion of that code, but rather will emphasize the differences between this program and two previous programs.

A mouse listener

The **Biomorph03** class begins in Listing 25.

```
public class Biomorph03 implements
MouseListener{
  static double[] genes = new
double[8];
  static GUI gui;
  static MouseListener listener;
  static Random rGen = new Random(
                           new
Date().getTime());

Listing 25
```

The most significant new thing in Listing 25 is the fact that this class implements the **MouseListener** interface. Thus, an object instantiated from this class can be registered on any component capable of firing mouse events, *(which includes objects of the Biomorph class).*

In addition to implementing the **MouseListener** interface, the code in Listing 25 also declares a static member variable of type **MouseListener**.

The main method

The **main** method begins in Listing 26. The code in Listing 26 instantiates an object of the **Biomorph03** class and saves its reference as type **MouseListener**.

```
  public static void main(String[]
args){
    listener = new Biomorph03();

    //Create initial set of genes.
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt] = 1.0;
    }//end for

Listing 26
```

Later on, this **MouseListener** object will be registered on each of the **Biomorph** objects. This will make it possible for the user to use the mouse to select one of the nine Biomorphs to serve as the parent Biomorph for the next generation.

Initial gene set

As was the case in the program named **Biomorph01**, this program causes the first seven genes to have an initial value of 1.0. Thus, the starting set of genes for this program is always the same.

> *(You may also find it interesting to modify this program to use a set of seven random values for the initial set of genes. You can lift that code from the program named **Biomorph02.**)*

A five-stage Biomorph

The value stored in the eighth gene specifies the number of stages that will be used to construct the Biomorph. The code in Listing 27 sets the initial value of this gene to 5.

```
    genes[7] = 5;

    gui = new
GUI(genes,listener,rGen);
  }//end main

Listing 27
```

Recall, however, that this value can increase or decrease due to mutation of the genes. If this value goes to zero, the Biomorph created using that gene set will disappear. As you learned earlier, the value of the eighth gene is not allowed to go negative.

A new GUI object

The code in Listing 27 also instantiates a new **GUI** object. The major difference is that this version passes the **MouseListener** object's reference to the constructor for the **GUI** object.

Listing 27 also signals the end of the **main** method.

The MouseListener methods

Because this class implements the **MouseListener** interface, it must provide concrete definitions of the five methods declared in the interface. One of those methods is named **mouseClicked**.

The definition of the **mouseClicked** method is shown in Listing 28. The purpose of this method is to make it possible for the user to select one of the nine Biomorphs to become the parent for the next generation of Biomorphs.

```
  public void mouseClicked(MouseEvent
e){
    Biomorph theMorph =

(Biomorph)(e.getSource());
    genes = theMorph.getGenes();

    gui.dispose();
```

```
    gui = new
GUI(genes,listener,rGen);
  }//end mouseClicked

Listing 28
```

Behavior of the mouseClicked method

The **mouseClicked** method does the following:

- Identify the specific Biomorph object that was selected with the mouse.
- Get and save the mutated gene array belonging to that Biomorph object. This will be the gene array belonging to the parent of the next generation. Each offspring Biomorph in the next generation will have these genes except that each offspring will mutate one gene in the array.
- Dispose of the existing **GUI** object in preparation for creating a new one that displays the parent and eight offspring in the next generation of Biomorphs.
- Instantiate a new **GUI** object based on the gene array belonging to the Biomorph that was selected.

The remaining MouseListener methods

Listing 29 defines the remaining four methods declared in the **MouseListener** interface as empty methods.

```
  public void mousePressed(MouseEvent
e){};
  public void mouseReleased(MouseEvent
e){};
  public void mouseEntered(MouseEvent
e){};
  public void mouseExited(MouseEvent
e){};

}//end class Biomorph03


Listing 29
```

Listing 29 also signals the end of the definition of class named **Biomorph03**.

The GUI class

The entire definition for the **GUI** class is shown in Listing 30. This class is used to instantiate a graphical user interface object that displays nine Biomorphs in a 3x3 grid. A **MouseListener** is registered on each Biomorph as it is added to the graphical user interface.

```
class GUI extends Frame{
  Random rGen;
  Biomorph[] morphs = new Biomorph[9];
  double[] genes;

  //Constructor
  public GUI(double[] genes,
             MouseListener listener,
             Random rGen){
    //Save incoming parameters.
    this.rGen = rGen;
    this.genes = genes;

    //Subdivide the GUI into nine grid
cells of
    // equal size.
    setLayout(new GridLayout(3,3));

    //Instantiate nine Biomorph objects.
Add
    // them to the GUI.  They are placed
in the
    // grid cells in the GUI from left
to right,
    // top to bottom.
    //Register a mouse listener on each
Biomorph
    // object.  Set the background color
for each
    // Biomorph object to produce
alternating
    // green and yellow backgrounds.
    for(int cnt = 0; cnt < 9; cnt++){
      //Instantiate and save a new
Biomorph
      // object.  Set the origin to be
the center
      // of the grid cell.
      morphs[cnt] = new Biomorph(genes,
                                 rGen,
                                 cnt,

genes[7]/8,
                                 66,
                                 66);

      //Add this Biomorph object to the
Frame in
      // the next grid cell.
      this.add(morphs[cnt]);

      //Register a mouse listener on the
Biomorph
      // object.

morphs[cnt].addMouseListener(listener);
```

```
      //Cause the background colors of
the
      // Biomorph objects to alternate
between
      // yellow and green so that they
will be
      // visually separable in the
Frame.
      if(cnt%2 == 0){

morphs[cnt].setBackground(Color.YELLOW);
      }else{

morphs[cnt].setBackground(Color.GREEN);
      }//end else
    }//end for loop

    //Finish the GUI and make it
visible.
    setTitle("Copyright 2004,
R.G.Baldwin");
    setSize(400,400);
    setVisible(true);

    //Instantiate and register a
Listener object
    // that will terminate the program
when the
    // user closes the Frame
    addWindowListener(
      new WindowAdapter(){
        public void
windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

}//end class GUI definition
```

Listing 30

Registering the MouseListener

About the only thing that causes this **GUI** class to be different from the classes with the same name in the previous two programs is the registration of the **MouseListener** object on each **Biomorph** object as it is added to the frame. I highlighted that statement in red boldface so that it will be easy for you to find.

Run the Programs

I encourage you to copy the code from the program listings near the end of this lesson. Compile and run the programs. Experiment with them, improving them as you see fit.

Above all, have fun.

Summary

I showed you how to write programs that model the selective breeding process, sometimes referred to as *artificial selection.* This is as opposed to *natural selection,* sometimes referred to as *survival of the fittest.*

Whether or not you found these programs to be useful, I hope you found them to be fun. Hopefully you also learned a few new things about Java programming based on the way that these programs are written.

Complete Program Listings

Complete listings of the three programs explained in this lesson are provided in Listing 31, Listing 32, and Listing 33 below.

```
/*File Biomorph01.java Copyright 2004,R.G.Baldwin
Revised 4/8/04

The purpose of this program is to compute and
display the first nine stages of growth for a
Biomorph based on a simple gene set where each of
the seven genes that control the shape of the
Biomorph have a fixed value of 1.

This program is loosely based on material in
Chapter 8 of the book entitled Windows Hothouse
by Mark Clarkson.  However, it was necessary for
me to find and fix several typographical errors
in the C++ algorithm presented in that book.

Tested using J2SE 1.4.2 under WinXP.
**********************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Biomorph01{
  //Store the genes here.  The first seven genes
  // control the shape of the Biomorph.  The
  // eighth gene specifies the number of stages
  // used to construct the Biomorph.
  static double[] genes = new double[8];

  //This random number generator is required by
  // the Biomorph constructor, but isn't used for
```

```
  // any purpose in this program.
  static Random rGen =
                new Random(new Date().getTime());

  public static void main(String[] args){
    //Create 7 fixed gene values.
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt]=1;
    }//end for
    //Specify the number of stages in the first
    // Biomorph object.
    genes[7] = 1;
    //Instantiate the GUI
    new GUI(genes,rGen);
  }//end main

}//end class Biomorph01

//=========================================//

//The following class is used to instantiate a
// graphical user interface object that causes
// the first nine stages of a Biomorph object to
// be created and displayed in nine grid cells in
// a Frame object.
class GUI extends Frame{
  Random rGen;
  double[] genes;

  //Constructor
  public GUI(double[] genes,Random rGen){
    //Save incoming parameters in local
    // variables.
    this.rGen = rGen;
    this.genes = genes;

    //Subdivide the Frame into nine grid cells.
    setLayout(new GridLayout(3,3));

    //Create and display nine stages of growth
    // for a Biomorph object using the same genes
    // for each stage.  Specify the third
    // parameter value to be negative to prevent
    // the Biomorph constructor from mutating the
    // genes.
    for(int cnt = 0; cnt < 9; cnt++){
      Biomorph biomorph =
                        new Biomorph(genes,
                                     rGen,
                                     -1,
                                     genes[7]/8,
                                     66,
                                     66);
      //Cause the background colors of the
      // Biomorph objects to alternate between
      // yellow and green so that they will be
```

```
      // visually separable in the Frame.
      if(cnt%2 == 0){
        biomorph.setBackground(Color.YELLOW);
      }else{
        biomorph.setBackground(Color.GREEN);
      }//end else

      //Add the Biomorph object to the Frame in
      // the next grid cell.
      this.add(biomorph);

      //Increase the number of stages for the
      // next Biomorph object.
      genes[7] += 1;
    }//end for loop

    setTitle("Copyright 2004, R.G.Baldwin");
    setSize(400,400);
    setVisible(true);

    //Instantiate and register a Listener object
    // that will terminate the program when the
    // user closes the Frame
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

}//end class GUI definition
//=============================================//

//This class is used to instantiate a Biomorph
// object.  It is based loosely on Chapter 8 of
// the book entitled Windows Hothouse by Mark
// Clarkson.  However, the C++ algorithm
// presented in that book contains several
// serious typographical errors.  It was
// necessary for me to find and fix those errors
// when writing a Java version of the algorithm.
//The constructor receives an array of eight gene
// values. The final value in the array specifies
// the number of stages to be used to construct
// the Biomorph object.  The first stage produces
// a single stem.  Each successive stage causes
// all existing stems to bifurcate into two new
// stems.  Thus, the number of stems increases as
// a power of two based on the number of stages.
// For example, a Biomorph created with two
// stages contains three stems.  A three-stage
// Biomorph contains seven stems, a four-stage
// Biomorph contains fifteen stems, etc.
//When writing the Java version of the algorithm,
```

```
// I elected to maintain all of the data as type
// double in order to preserve arithmetic
// accuracy.  Values are converted from double to
// int at the very last step before displaying
// the Biomorph on the screen.
//The constructor receives a random number
// generator and a count value that are used to
// mutate a gene in the array of genes by a
// random value of plus or minus one whenever the
// count value is within the range from 0 to 7.
// If the count value is outside this range,
// there is no gene mutation.
//A method named getGenes returns the gene array
// containing the possibly mutated gene.  This is
// useful for experiments in selective breeding.
//The constructor receives a scale factor that is
// used to adjust the overall size of the plot to
// cause it to fit in the allocated plotting
// area.  Generally speaking, the size of the raw
// display of the Biomorph object would increase
// as the number of stages increases.  Therefore,
// it is useful to cause the scale factor to vary
// inversely with the number of stages.
//The constructor receives a pair of int values
// that are used to move the plotting origin
// from the default upper-left corner to another
// location in the plotting area.
//The direction of the first stem displayed for
// the Biomorph object is hard-coded to be
// vertical going up the screen, starting at the
// origin.
class Biomorph extends Panel{
  double[] xInc = new double[8];
  double[] yInc = new double[8];
  double[] genes;
  double xCoor = 0;//Start drawing here
  double yCoor = 0;//Start drawing here
  int direction = 0;//Initial drawing direction
  double length;
  double scale;
  int xOrigin;
  int yOrigin;

  //Constructor
  Biomorph(double[] genes,Random rGen,int cnt,
           double scale,int xOrigin,int yOrigin){

    //Save local copies of incoming parameters.
    this.genes = (double[])genes.clone();
    this.scale = scale;
    this.xOrigin = xOrigin;
    this.yOrigin = yOrigin;

    //Mutate gene at position cnt unless cnt is
    // out of the range from 0 through 7
    // inclusive.
```

```
    if((cnt>=0) && (cnt<=7)){
      double mutantValue = rGen.nextInt(2)*2-1;
      this.genes[cnt] += mutantValue;
      //Don't allow the eighth gene to go
      // negative
      if(this.genes[7] < 0)this.genes[7] = 0;
    }//end if

    //Compute incremental ends of lines based on
    // gene values.  Note that the C++ algorithm
    // presented in the Clarkson book appears to
    // contain several errors at this point.
    // Either that, or perhaps I don't fully
    // understand his version of the algorithm.
    xInc[0] =  0;
                        yInc[0] =  this.genes[0];
    xInc[1] =  this.genes[1];
                        yInc[1] =  this.genes[2];
    xInc[2] =  this.genes[3];
                        yInc[2] =  0;
    xInc[3] =  this.genes[4];
                        yInc[3] = -this.genes[5];
    xInc[4] =  0;
                        yInc[4] = -this.genes[6];
    xInc[5] = -this.genes[4];
                        yInc[5] = -this.genes[5];
    xInc[6] = -this.genes[3];
                        yInc[6] =  0;
    xInc[7] = -this.genes[1];
                        yInc[7] =  this.genes[2];


    //Initial line length is based on the number
    // of stages to be drawn.  Line length is
    // reduced by one as each successive stage is
    // drawn.  Algorithm terminates when length
    // reaches zero.
    length = this.genes[7];
  }//end constructor

  double[] getGenes(){
    return this.genes;
  }//end getGenes

  //Override the paint method
  public void paint(Graphics g){
    //Adjust location of the plotting origin
    g.translate(xOrigin,yOrigin);
    //Draw the Biomorph object recursively
    drawIt(g,xCoor,yCoor,length,direction,xInc,
                                      yInc,scale);
  }//end paint()
  //-----------------------------------------//

  void drawIt(Graphics g,double oldX,
              double oldY,double len,int newDir,
```

```
              double[] xInc,double[] yInc,
              double scale){
    //Direction values are limited to the range
    // from 0 to 7.
    newDir = (newDir + 8)%8;

    //Compute the end points of the line to be
    // drawn based ultimately on the values in
    // the gene array.
    double newX = oldX + len * xInc[newDir];
    double newY = oldY + len * yInc[newDir];

    //Draw the line.  Correct for the fact that
    // the default direction for positive y is
    // down the screen.
    g.drawLine((int)(oldX/scale),
               (int)(-oldY/scale),
               (int)(newX/scale),
               (int)(-newY/scale));

    //Continue drawing lines recursively until
    // the length of the next line reaches zero.
    // Decrease the length of the line by one for
    // each successive stage.  The values for
    // newX and newY become the incoming oldX and
    // oldY values for the next recursion.  Don't
    // waste time trying to draw a line with zero
    // length.
    if(len > 1){
      drawIt(g,newX,newY,len-1,newDir+1,xInc,
             yInc,scale);
      drawIt(g,newX,newY,len-1,newDir-1,xInc,
             yInc,scale);
    }//end if
  }//end drawIt

}//end class Biomorph
//=============================================//

Listing 31
```

```
/*File Biomorph02.java Copyright 2004,R.G.Baldwin
Revised 4/8/04

The purpose of this program is to compute and
display the first nine stages of growth for a
Biomorph based on a complex gene set where each
of the seven genes that control the shape of the
Biomorph are obtained from a random number
generator.  This program should produce Biomorph
objects having different appearances each time it
is run.
```

```
This program is loosely based on material in
Chapter 8 of the book entitled Windows Hothouse
by Mark Clarkson.  However, it was necessary for
me to find and fix several typographical errors
in the C++ algorithm presented in that book.

Tested using J2SE 1.4.2 under WinXP.
**********************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Biomorph02{
  //Store the genes here.  The first seven genes
  // control the shape of the Biomorph.  The
  // eighth gene specifies the number of stages
  // used to construct the Biomorph.
  static double[] genes = new double[8];

  //This random number generator is required by
  // the Biomorph constructor.  In this program,
  // it is used to create the gene set, but is
  // not used otherwise in the construction of
  // the Biomorph objects.
  static Random rGen =
                new Random(new Date().getTime());

  public static void main(String[] args){
    //Create 7 random gene values.  This is the
    // code in this program that is different
    // from the code in the program named
    // Biomorph01.
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt] = rGen.nextInt(7)-3;
    }//end for
    //Specify the number of stages in the first
    // Biomorph object.
    genes[7] = 1;
    //Instantiate the GUI
    new GUI(genes,rGen);
  }//end main

}//end class Biomorph02

//============================================//

//The following class is used to instantiate a
// graphical user interface object that causes
// the first nine stages of a Biomorph object to
// be created and displayed in nine grid cells in
// a Frame object.
class GUI extends Frame{
  Random rGen;
  double[] genes;
```

```
  //Constructor
  public GUI(double[] genes,Random rGen){
    //Save incoming parameters in local
    // variables.
    this.rGen = rGen;
    this.genes = genes;

    //Subdivide the Frame into nine grid cells.
    setLayout(new GridLayout(3,3));

    //Create and display nine stages of growth
    // for a Biomorph object using the same genes
    // for each stage.  Specify the third
    // parameter value to be negative to prevent
    // the Biomorph constructor from mutating the
    // genes.
    for(int cnt = 0; cnt < 9; cnt++){
      Biomorph biomorph =
                        new Biomorph(genes,
                                     rGen,
                                     -1,
                                     genes[7]/8,
                                     66,
                                     66);
      //Cause the background colors of the
      // Biomorph objects to alternate between
      // yellow and green so that they will be
      // visually separable in the Frame.
      if(cnt%2 == 0){
        biomorph.setBackground(Color.YELLOW);
      }else{
        biomorph.setBackground(Color.GREEN);
      }//end else

      //Add the Biomorph object to the Frame in
      // the next grid cell.
      this.add(biomorph);

      //Increase the number of stages for the
      // next Biomorph object.
      genes[7] += 1;
    }//end for loop

    setTitle("Copyright 2004, R.G.Baldwin");
    setSize(400,400);
    setVisible(true);

    //Instantiate and register Listener object
    // that will terminate the program when the
    // user closes the Frame
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end WindowAdapter
```

```
    );//end addWindowListener
  }//end constructor

}//end class GUI definition
//===========================================//

//This class is used to instantiate a Biomorph
// object.  It is based loosely on Chapter 8 of
// the book entitled Windows Hothouse by Mark
// Clarkson.  However, the C++ algorithm
// presented in that book contains several
// serious typographical errors.  It was
// necessary for me to find and fix those errors
// when writing a Java version of the algorithm.
//The constructor receives an array of eight gene
// values. The final value in the array specifies
// the number of stages to be used to construct
// the Biomorph object.  The first stage produces
// a single stem.  Each successive stage causes
// all existing stems to bifurcate into two new
// stems.  Thus, the number of stems increases as
// a power of two based on the number of stages.
// For example, a Biomorph created with two
// stages contains three stems.  A three-stage
// Biomorph contains seven stems, a four-stage
// Biomorph contains fifteen stems, etc.
//When writing the Java version of the algorithm,
// I elected to maintain all of the data as type
// double in order to preserve arithmetic
// accuracy.  Values are converted from double to
// int at the very last step before displaying
// the Biomorph on the screen.
//The constructor receives a random number
// generator and a count value that are used to
// mutate a gene in the array of genes by a
// random value of plus or minus one whenever the
// count value is within the range from 0 to 7.
// If the count value is outside this range,
// there is no gene mutation.
//A method named getGenes returns the gene array
// containing the possibly mutated gene.  This is
// useful for experiments in selective breeding.
//The constructor receives a scale factor that is
// used to adjust the overall size of the plot to
// cause it to fit in the allocated plotting
// area.  Generally speaking, the size of the raw
// display of the Biomorph object would increase
// as the number of stages increases.  Therefore,
// it is useful to cause the scale factor to vary
// inversely with the number of stages.
//The constructor receives a pair of int values
// that are used to move the plotting origin
// from the default upper-left corner to another
// location in the plotting area.
//The direction of the first stem displayed for
// the Biomorph object is hard-coded to be
```

```
// vertical going up the screen, starting at the
// origin.
class Biomorph extends Panel{
  double[] xInc = new double[8];
  double[] yInc = new double[8];
  double[] genes;
  double xCoor = 0;//Start drawing here
  double yCoor = 0;//Start drawing here
  int direction = 0;//Initial drawing direction
  double length;
  double scale;
  int xOrigin;
  int yOrigin;

  //Constructor
  Biomorph(double[] genes,Random rGen,int cnt,
           double scale,int xOrigin,int yOrigin){

    //Save local copies of incoming parameters.
    this.genes = (double[])genes.clone();
    this.scale = scale;
    this.xOrigin = xOrigin;
    this.yOrigin = yOrigin;

    //Mutate gene at position cnt unless cnt is
    // out of the range from 0 through 7
    // inclusive.
    if((cnt>=0) && (cnt<=7)){
      double mutantValue = rGen.nextInt(2)*2-1;
      this.genes[cnt] += mutantValue;
      //Don't allow the eighth gene to go
      // negative
      if(this.genes[7] < 0)this.genes[7] = 0;
    }//end if

    //Compute incremental ends of lines based on
    // gene values.  Note that the C++ algorithm
    // presented in the Clarkson book appears to
    // contain several errors at this point.
    // Either that, or perhaps I don't fully
    // understand his version of the algorithm.
    xInc[0] =  0;
                         yInc[0] =  this.genes[0];
    xInc[1] =  this.genes[1];
                         yInc[1] =  this.genes[2];
    xInc[2] =  this.genes[3];
                         yInc[2] =  0;
    xInc[3] =  this.genes[4];
                         yInc[3] = -this.genes[5];
    xInc[4] =  0;
                         yInc[4] = -this.genes[6];
    xInc[5] = -this.genes[4];
                         yInc[5] = -this.genes[5];
    xInc[6] = -this.genes[3];
                         yInc[6] =  0;
    xInc[7] = -this.genes[1];
```

```
                        yInc[7] =  this.genes[2];


    //Initial line length is based on the number
    // of stages to be drawn.  Line length is
    // reduced by one as each successive stage is
    // drawn.  Algorithm terminates when length
    // reaches zero.
    length = this.genes[7];
  }//end constructor

  double[] getGenes(){
    return this.genes;
  }//end getGenes

  //Override the paint method
  public void paint(Graphics g){
    //Adjust location of the plotting origin
    g.translate(xOrigin,yOrigin);
    //Draw the Biomorph object recursively
    drawIt(g,xCoor,yCoor,length,direction,xInc,
                                      yInc,scale);
  }//end paint()
  //--------------------------------------//

  void drawIt(Graphics g,double oldX,
              double oldY,double len,int newDir,
              double[] xInc,double[] yInc,
              double scale){
    //Direction values are limited to the range
    // from 0 to 7.
    newDir = (newDir + 8)%8;

    //Compute the end points of the line to be
    // drawn based ultimately on the values in
    // the gene array.
    double newX = oldX + len * xInc[newDir];
    double newY = oldY + len * yInc[newDir];

    //Draw the line.  Correct for the fact that
    // the default direction for positive y is
    // down the screen.
    g.drawLine((int)(oldX/scale),
               (int)(-oldY/scale),
               (int)(newX/scale),
               (int)(-newY/scale));

    //Continue drawing lines recursively until
    // the length of the next line reaches zero.
    // Decrease the length of the line by one for
    // each successive stage.  The values for
    // newX and newY become the incoming oldX and
    // oldY values for the next recursion.  Don't
    // waste time trying to draw a line with zero
    // length.
    if(len > 1){
```

```
      drawIt(g,newX,newY,len-1,newDir+1,xInc,
             yInc,scale);
      drawIt(g,newX,newY,len-1,newDir-1,xInc,
             yInc,scale);
    }//end if
  }//end drawIt

}//end class Biomorph
//=============================================//

Listing 32
```

```
/*File Biomorph03.java Copyright 2004,R.G.Baldwin
Revised 04/08/04

This program falls in the general category of
Artificial Life.  The program models
an experiment in the evolutionary concept of
artificial selection as opposed to natural
selection.  For example, the variety of plants,
animals, and birds that exist on on uninhabited
island represent natural selection, sometimes
referred to as survival of the fittest.

A dalmation dog, on the other hand, is probably
the result of artificial selection. In other
words, over a long period of time, people
selected certain dogs for breeding to accentuate
certain characteristics (such as black spots on a
white coat) and to suppress other characteristics
(such as a long red coat).  Over time, what
resulted was a type of dog that we know as the
dalmation dog.  Although those who did that may
not have known that those characteristics were
represented by genes that were accentuated or
suppressed through selective breeding, we know
(or at least believe) that to be the case now.

This program makes it possible for you to
selectively breed successive generations of
artificial creatures known as Biomorph objects.
A single parent in one generation produces eight
offspring in the next generation.

Each Biomorph object is a recursively branching
tree consisting of many limbs of different
lengths that branch off in different directions.
Each such Biomorph object has eight genes that
control the size,the number, and the angle of the
branches.

During the creation of each new generation, one
of the genes for each of the eight offspring is
randomly mutated to produce a creature that is
```

```
similar to, but different from its parent.  You
can select one of the offspring from each
generation to become the parent of the next
generation in order to accentuate certain
characteristics and to suppress other
characteristics.  By continuing this process
through a large number of generations, you can
cause the resulting Biomorph objects to resemble
birds, bugs, animals, airplanes,human faces, or
whatever strikes your fancy.

The parent for each generation is displayed as
the ninth Biomorph object.  If you don't like any
of the eight offspring of that parent, you can
select it again and cause it to produce eight
more offspring based on random mutations of the
genes.

This program is loosely based on Chapter 8 of the
book entitled Windows Hothouse by Mark Clarkson.
That chapter was based on a book and a paper
published by Richard Dawkins.  The book was
entitled The Blind Watchmaker.  The paper was
entitled The Evolution of Evolvability and
appeared in the book entitled Artificial Life.

Tested using J2SE 1.4.2 under WinXP.
*********************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Biomorph03 implements MouseListener{
  static double[] genes = new double[8];
  static GUI gui;
  static MouseListener listener;
  static Random rGen = new Random(
                           new Date().getTime());

  public static void main(String[] args){
    //An object of this class is a mouse listener
    listener = new Biomorph03();

    //Create initial set of genes.
    for(int cnt = 0; cnt < 7; cnt++){
      genes[cnt] = 1.0;
    }//end for

    //Establish the initial number of stages that
    // will be used to create the first
    // generation of Biomorph objects.  This
    // value, which is contained in the eighth
    // gene can increase or decrease due to
    // mutation of the genes.  If it goes to
    // zero, that Biomorph object will disappear.
```

```
    // It is not allowed to go negative.
    genes[7] = 5;

    //Instantiate a new GUI object.
    gui = new GUI(genes,listener,rGen);
  }//end main

  //Define a MouseEvent handler to handle mouse
  // clicks on Biomorph objects.  The mouse is
  // used to select one of nine Biomorph objects
  // to become the parent of the next generation.
  public void mouseClicked(MouseEvent e){
    //Identify the specific Biomorph object that
    // was selected with the mouse.  Get and save
    // the mutated gene array belonging to that
    // object.  This will be the gene array of
    // the parent of the next generation.
    Biomorph theMorph =
                        (Biomorph)(e.getSource());
    genes = theMorph.getGenes();

    //Dispose of the existing GUI object in
    // preparation for creating a new one.
    gui.dispose();

    //Instantiate a new GUI object based on the
    // mutated gene array obtained from the
    // Biomorph object that was selected.
    gui = new GUI(genes,listener,rGen);
  }//end mouseClicked

  //Define remaining methods of the MouseListener
  // interface as empty methods.
  public void mousePressed(MouseEvent e){};
  public void mouseReleased(MouseEvent e){};
  public void mouseEntered(MouseEvent e){};
  public void mouseExited(MouseEvent e){};

}//end class Biomorph03
//===========================================//

//The following class is used to instantiate a
// graphical user interface object that displays
// nine Biomorph objects in a 3x3 grid.
class GUI extends Frame{
  Random rGen;
  Biomorph[] morphs = new Biomorph[9];
  double[] genes;

  //Constructor
  public GUI(double[] genes,
             MouseListener listener,
             Random rGen){
    //Save incoming parameters.
    this.rGen = rGen;
    this.genes = genes;
```

```
    //Subdivide the GUI into nine grid cells of
    // equal size.
    setLayout(new GridLayout(3,3));

    //Instantiate nine Biomorph objects.  Add
    // them to the GUI.  They are placed in the
    // grid cells in the GUI from left to right,
    // top to bottom.
    //Register a mouse listener on each Biomorph
    // object.  Set the background color for each
    // Biomorph object to produce alternating
    // green and yellow backgrounds.
    for(int cnt = 0; cnt < 9; cnt++){
      //Instantiate and save a new Biomorph
      // object.  Set the origin to be the center
      // of the grid cell.
      morphs[cnt] = new Biomorph(genes,
                                 rGen,
                                 cnt,
                                 genes[7]/8,
                                 66,
                                 66);

      //Add this Biomorph object to the Frame in
      // the next grid cell.
      this.add(morphs[cnt]);

      //Register a mouse listener on the Biomorph
      // object.
      morphs[cnt].addMouseListener(listener);

      //Cause the background colors of the
      // Biomorph objects to alternate between
      // yellow and green so that they will be
      // visually separable in the Frame.
      if(cnt%2 == 0){
        morphs[cnt].setBackground(Color.YELLOW);
      }else{
        morphs[cnt].setBackground(Color.GREEN);
      }//end else
    }//end for loop

    //Finish the GUI and make it visible.
    setTitle("Copyright 2004, R.G.Baldwin");
    setSize(400,400);
    setVisible(true);

    //Instantiate and register a Listener object
    // that will terminate the program when the
    // user closes the Frame
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
```

```
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

}//end class GUI definition
//===========================================//

//This class is used to instantiate a Biomorph
// object.  It is based loosely on Chapter 8 of
// the book entitled Windows Hothouse by Mark
// Clarkson.  However, the C++ algorithm
// presented in that book contains several
// serious typographical errors.  It was
// necessary for me to find and fix those errors
// when writing a Java version of the algorithm.
//The constructor receives an array of eight gene
// values. The final value in the array specifies
// the number of stages to be used to construct
// the Biomorph object.  The first stage produces
// a single stem.  Each successive stage causes
// all existing stems to bifurcate into two new
// stems.  Thus, the number of stems increases as
// a power of two based on the number of stages.
// For example, a Biomorph created with two
// stages contains three stems.  A three-stage
// Biomorph contains seven stems, a four-stage
// Biomorph contains fifteen stems, etc.
//When writing the Java version of the algorithm,
// I elected to maintain all of the data as type
// double in order to preserve arithmetic
// accuracy.  Values are converted from double to
// int at the very last step before displaying
// the Biomorph on the screen.
//The constructor receives a random number
// generator and a count value that are used to
// mutate a gene in the array of genes by a
// random value of plus or minus one whenever the
// count value is within the range from 0 to 7.
// If the count value is outside this range,
// there is no gene mutation.
//A method named getGenes returns the gene array
// containing the possibly mutated gene.  This is
// useful for experiments in selective breeding.
//The constructor receives a scale factor that is
// used to adjust the overall size of the plot to
// cause it to fit in the allocated plotting
// area.  Generally speaking, the size of the raw
// display of the Biomorph object would increase
// as the number of stages increases.  Therefore,
// it is useful to cause the scale factor to vary
// inversely with the number of stages.
//The constructor receives a pair of int values
// that are used to move the plotting origin
// from the default upper-left corner to another
// location in the plotting area.
//The direction of the first stem displayed for
```

```
// the Biomorph object is hard-coded to be
// vertical going up the screen, starting at the
// origin.
class Biomorph extends Panel{
  double[] xInc = new double[8];
  double[] yInc = new double[8];
  double[] genes;
  double xCoor = 0;//Start drawing here
  double yCoor = 0;//Start drawing here
  int direction = 0;//Initial drawing direction
  double length;
  double scale;
  int xOrigin;
  int yOrigin;

  //Constructor
  Biomorph(double[] genes,Random rGen,int cnt,
           double scale,int xOrigin,int yOrigin){

    //Save local copies of incoming parameters.
    this.genes = (double[])genes.clone();
    this.scale = scale;
    this.xOrigin = xOrigin;
    this.yOrigin = yOrigin;

    //Mutate gene at position cnt unless cnt is
    // out of the range from 0 through 7
    // inclusive.
    if((cnt>=0) && (cnt<=7)){
      double mutantValue = rGen.nextInt(2)*2-1;
      this.genes[cnt] += mutantValue;
      //Don't allow the eighth gene to go
      // negative
      if(this.genes[7] < 0)this.genes[7] = 0;
    }//end if

    //Compute incremental ends of lines based on
    // gene values.  Note that the C++ algorithm
    // presented in the Clarkson book appears to
    // contain several errors at this point.
    // Either that, or perhaps I don't fully
    // understand his version of the algorithm.
    xInc[0] =  0;
                         yInc[0] =  this.genes[0];
    xInc[1] =  this.genes[1];
                         yInc[1] =  this.genes[2];
    xInc[2] =  this.genes[3];
                         yInc[2] =  0;
    xInc[3] =  this.genes[4];
                         yInc[3] = -this.genes[5];
    xInc[4] =  0;
                         yInc[4] = -this.genes[6];
    xInc[5] = -this.genes[4];
                         yInc[5] = -this.genes[5];
    xInc[6] = -this.genes[3];
                         yInc[6] =  0;
```

```
    xInc[7] = -this.genes[1];
                        yInc[7] =  this.genes[2];


    //Initial line length is based on the number
    // of stages to be drawn.  Line length is
    // reduced by one as each successive stage is
    // drawn.  Algorithm terminates when length
    // reaches zero.
    length = this.genes[7];
  }//end constructor

  double[] getGenes(){
    return this.genes;
  }//end getGenes

  //Override the paint method
  public void paint(Graphics g){
    //Adjust location of the plotting origin
    g.translate(xOrigin,yOrigin);
    //Draw the Biomorph object recursively
    drawIt(g,xCoor,yCoor,length,direction,xInc,
                                      yInc,scale);
  }//end paint()
  //---------------------------------------//

  void drawIt(Graphics g,double oldX,
              double oldY,double len,int newDir,
              double[] xInc,double[] yInc,
              double scale){
    //Direction values are limited to the range
    // from 0 to 7.
    newDir = (newDir + 8)%8;

    //Compute the end points of the line to be
    // drawn based ultimately on the values in
    // the gene array.
    double newX = oldX + len * xInc[newDir];
    double newY = oldY + len * yInc[newDir];

    //Draw the line.  Correct for the fact that
    // the default direction for positive y is
    // down the screen.
    g.drawLine((int)(oldX/scale),
               (int)(-oldY/scale),
               (int)(newX/scale),
               (int)(-newY/scale));

    //Continue drawing lines recursively until
    // the length of the next line reaches zero.
    // Decrease the length of the line by one for
    // each successive stage.  The values for
    // newX and newY become the incoming oldX and
    // oldY values for the next recursion.  Don't
    // waste time trying to draw a line with zero
    // length.
```

```
    if(len > 1){
      drawIt(g,newX,newY,len-1,newDir+1,xInc,
             yInc,scale);
      drawIt(g,newX,newY,len-1,newDir-1,xInc,
             yInc,scale);
    }//end if
  }//end drawIt

}//end class Biomorph
//=============================================//

Listing 33
```

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-