

DRAFT

jSIMPLEBUG: a Swarm tutorial for Java

Charles P. Staelin
Smith College
Northampton, MA

April 2000

based on ObjectiveC code and original text by
Chris Lanton & Swarm development team
Santa Fe Institute, NM

DRAFT

jSimpleBug: a Swarm tutorial for Java

Introduction

This tutorial takes the user through the development of a Swarm model using the Java programming language. The model itself is a very simple one, but the application we build around it makes use of a lot of the functionality of Swarm and demonstrates many of Swarm's features. This Java-based tutorial borrows heavily from an earlier Objective-C-based tutorial by Chris Lanton and the Swarm development team. Indeed, the Java code is pretty much a loose translation of their Objective-C code and portions of this text are theirs as well. There are a number of changes, however, to reflect this tutorial's focus on using Swarm with Java rather than Objective-C.

We start out with a very simple Swarm simulation, one agent, a "SimpleBug," taking a random walk on the X,Y plane. Through a progression of models we increase the number of agents and introduce basic object-oriented and Swarm-style programming in Java. The final version implements an experiment in which multiple invocations of the SimpleBug model are created, run, analyzed, reported on, and dropped. Along the way, we introduce many of the functions that Swarm provides for creating and interacting with multi-agent, artificial worlds. Although this is a relatively simple exercise, hopefully it will show how easy it can be to build fairly complex models from the simple building blocks provided by Swarm.

There are 10 subdirectories in the jTutorial directory, each with a complete application. You should start with the "SimpleCBug" subdirectory and then proceed through the others in the following order:

[SimpleCBug](#)
[SimpleJavaBug](#)
[SimpleJavaBug2](#)
[SimpleSwarmBug](#)
[SimpleSwarmBug2](#)
[SimpleSwarmBug3](#)
[SimpleObserverBug](#)
[SimpleObserverBug2](#)
[SimpleObserverBug3](#)
SimpleExperBug

Once you have gone all the way through this tutorial, you should be able to make sense of many of the applications on the [Swarm web-site](#)¹. Upon first look, these applications appear quite complex. However, they are really not that hard to understand once you get a feel for the underlying patterns and you are encouraged to build upon them for your own applications.

¹ Two Java applications, jheatbugs and jmousetrap, continue in the spirit of this tutorial. The versions appropriate to the latest version of Swarm may be found at <ftp://ftp.santafe.edu/pub/swarm/src/apps/java>. There are also a growing number of applications and tools, in both Objective-C and Java, on [Swarm's Community web page](#).

This tutorial

This version of the jSimpleBug tutorial was written for Swarm version 2.1.1. It is available as a formatted document in both postscript and PDF. The sections of the tutorial aimed at each of the SimpleBug applications are also reproduced in plain text in the "readme" files in the corresponding directories.

In the postscript and PDF versions, italics are used to denote Swarm classes, methods and variables.

Please report all errors in the tutorial text or code to [the author](#). Comments on the clarity and usefulness of the tutorial are also most welcome.

On building the applications

We assume that you have Swarm installed in either a Unix/Linux environment or a Windows environment. The applications can be built from the command line (the "Swarm Terminal" under Windows) by going to the application directory and typing

```
make
```

Then, to run any of them, just type

```
javaswarm StartSimpleBug
```

(There are variations on this procedure if you are using Sun's JDK rather than the Kaffe JDK supplied with Swarm, or if you are using Emacs. See the Appendix.)

/SimpleCBug

THE STARTING POINT

We begin with a simple C-style program written in Java, **StartSimpleBug.java**. There is no object orientation here. The program could as easily have been written in C, Pascal or Fortran.

StartSimpleBug implements an imagined "bug" taking a random walk on an imagined two-dimensional integer lattice. We have no objects here to provide substance to the bug or its world. The 80 by 80 lattice on which the bug walks is defined by the declarations of worldXSize and worldYSize. The bug's initial position is given by the declarations of xPos and yPos. The bug then takes 100 random walks. On each walk, the bug move first in the X direction a distance given by the randomMove() method, and then in the Y direction a distance given by another call to randomMove(). Because the bug's world is finite, we need to make sure it does not wander off one of its edges. The use of the modulus operator insures that if it wanders over one edge, it will magically be transported to the opposite edge of its world. The bug's world is a torus.

randomMove() makes use of the simple random number generator in Java's Math library. randomMove() returns with equal probability either -1 for a step backward, 0, for no step at all, or +1 for a step forward. (Swarm provides several more sophisticated random number generators. We'll introduce two of them in the next version of our application.)

There is no object-oriented code here and, except for a necessary call to *initSwarm()*, nothing of Swarm itself. (We'll explain *initSwarm()* in the next application.) This is just a starting point from which to develop Swarm concepts and object orientation as we move through the following applications.

/SimpleCBug/StartSimpleBug.java

```
// StartSimpleBug.java
// The Java SimpleBug application.

import swarm.Globals;

public class StartSimpleBug
{
    public static void main (String[] args)
    {
        // The size of the bug's world and its initial position.
        int worldXSize = 80;
        int worldYSize = 80;
        int xPos = 40;
        int yPos = 40;

        int i;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("SimpleBug", "2.1",
                               "bug-swarm@santafe.edu", args);
    }
}
```

```

System.out.println("I started at X = " + xPos + " Y = " + yPos);

// Have the bug randomly walk backward (-1), forward (+1), or
// not at all (0) in first the X and then the Y direction.
// (The randomMove() method, defined below, returns an
// integer between -1 and +1.) Note that the bug's world is a
// torus. If the bug walks off the edge of its rectangular
// world, it is magically transported (via the modulus
// operator) to the opposite edge.
for(i = 0; i < 100; i++)
    {
        xPos += randomMove();
        yPos += randomMove();
        xPos = (xPos + worldXSize) % worldXSize;
        yPos = (yPos + worldYSize) % worldYSize;

        System.out.println("I moved to X = " + xPos + " Y = " + yPos);
    }

return;
}

// Returns -1, 0 or +1 with equal probability.
static int randomMove()
{
    double randnum;

    // Math.random returns a pseudo random number in the interval
    // [0, 1).
    randnum = Math.random();

    if (randnum <= 0.33333)
        return -1;
    else if (randnum <= 0.66667)
        return 0;
    else
        return 1;
}
}

```

The **Makefile** for our application simply lists the Java source file and specifies that the javacswarm script provided in Swarm's /bin directory is to be used to compile it.

/SimpleCBug/Makefile

```

JAVA_SRC = StartSimpleBug.java

all: $(JAVA_SRC)
    $(SWARMHOME)/bin/javacswarm $(JAVA_SRC)

clean:
    rm -f *.class

```

If we type "javacswarm StartSimpleBug" at the console, we should get the birth announcement for our bug, followed by 100 reports on its position.

/SimpleJavaBug

THE STARTING POINT

This is an object-oriented version of the SimpleCBug program, one that also introduces a few elements of Swarm. The **SimpleBug.java** file objectifies the agents in our simulation, one or more simple bugs that take random walks within a rectangular world defined on an integer lattice, by defining the SimpleBug class to which all the bugs belong. Each bug we create will be an instantiated object of the class.

Note first that the SimpleBug class is a subclass of the class *SwarmObjectImpl* and that upon its creation each bug is given a *Zone* (aZone), the size of its world and its initial position on it (wXSize, wYSize, X and Y), and a bug number (bNum). We'll talk later about why it is desirable for our agents to be a type of *SwarmObjectImpl* and we'll have more to say about zones as well. For now, we can think of a zone as an area of memory allocated by Swarm in which the bug and all its resources will be created. The SimpleBug constructor begins by calling the constructor of its parent class (*SwarmObjectImpl*) and passing on the zone SimpleBug was passed. The constructor then saves the size of the new bug's world, the new bug's current position and its bug number. Finally, the newly created bug announces its presence to the console.

Before looking out our bugs' capabilities (methods), note that we have imported a number of class libraries. The first, *swarm.Globals*, is a class descriptor that will be included in every Swarm source file. It describes a set of global variables and methods that are useful throughout a Swarm program. The next two, *swarm.defobj.Zone* and *swarm.objectbase.SwarmObjectImpl*, are needed to provide definitions for the *Zone* and *SwarmObjectImpl* classes. We'll have more to say about including Swarm class descriptors in a later section when we talk about Swarm's Java API.

Our SimpleBugs are capable of two actions: they can take random walks on their X,Y integer lattice and they can report their position to the console. In the first method, *randomWalk()*, the bug walks randomly backward or forward in the X direction, and then randomly backward or forward in the Y direction. In each case, the distance is given by -1, 0 or +1, and the modulus operator is used to insure that the bug does not wander out of its world. (The world in which the bugs walk is a torus. If the bug walks off one edge of its world, it reappears on the opposite edge.)

The random number generator used by *randomWalk()* is one supplied by Swarm's *Globals* environment. When we start up a Swarm program, we create an instance of Swarm's *Globals* class, *Globals.env*. That object contains a number useful instance variables and methods, among them the *uniformIntRand.getIntegerWithMin\$Max()* method. Swarm supplies a number of very powerful and flexible random number generators in a special library package, *swarm.random*. However, two of the simpler of them are so commonly used that instances of them are contained in Swarm's *Globals* environment: *uniformIntRand.getIntegerWithMin\$Max()* and its real number counterpart, *uniformDblRand.getDoubleWithMin\$withMax()*.

/SimpleJavaBug/SimpleBug.java

```

// SimpleBug.java
// Defines the class for our SimpleBug agents.

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

public class SimpleBug extends SwarmObjectImpl
{
    // These instance variables keep track of the size of a given
    // bug's world, its position within it, and its identity.
    int worldXSize;
    int worldYSize;
    int xPos;
    int yPos;
    int bugNumber;

    // Constructor to create a SimpleBug object in Zone aZone and to
    // place it at the specified X,Y location in its world. The bug
    // is also given a numeric id, bNum.
    public SimpleBug(Zone aZone, int wXSize, int wYSize, int X, int Y,
                    int bNum)
    {
        // Call the constructor for the bug's parent class.
        super(aZone);

        // Record the bug's world size, its initial position and id
        // number.
        worldXSize = wXSize;
        worldYSize = wYSize;
        xPos = X;
        yPos = Y;
        bugNumber = bNum;

        // Announce the bug's presence to the console.
        System.out.println("SimpleBug number " + bugNumber +
                           " has been created at " + xPos + ", " + yPos);
    }

    // This is the method to have the bug take a random walk backward
    // (-1), forward (+1), or not at all (0) in first the X and then
    // the Y direction. The randomWalk method uses
    // getIntegerWithMin$withMax() to return an integer between a
    // minimum and maximum value, here between -1 and +1.
    // Globals.env.uniformRand is an instance of the class
    // UniformIntegerDistImpl, instantiated by the call to
    // Globals.env.initSwarm in StartSimpleBug. Note that the bug's
    // world is a torus. If the bug walks off the edge of its
    // rectangular world, it is magically transported (via the modulus
    // operator) to the opposite edge.
    public void randomWalk()
    {
        xPos += Globals.env.uniformIntRand.getIntegerWithMin$withMax(
                -1, 1);
        yPos += Globals.env.uniformIntRand.getIntegerWithMin$withMax(
                -1, 1);

        xPos = (xPos + worldXSize) % worldXSize;
        yPos = (yPos + worldYSize) % worldYSize;
    }
}

```

```

// Method to report the bug's location to the console.
public void reportPosition()
{
    System.out.println("Bug " + bugNumber + " is at " + xPos +
        ", " + yPos);
}
}

```

Now that we have our bugs, we are ready to start our simulation. The file **StartSimpleBug.java** initializes the Swarm environment, creates a SimpleBug and sets it in motion.

We begin by calling *Globals.env.initSwarm()*, a static method in Swarm's Global environment. This method sets up Swarm's global variables and methods and is called at the beginning of every Swarm program. We pass it four arguments, the name we choose to give our simulation, the version of Swarm we are using, an email address to use for bug reports, and the string of command-line arguments with which the program was started.

We then create a SimpleBug in an 80-unit rectangular world and set it at the midpoint of its world. The bug is created in Swarm's globalZone. (We'll talk later about the circumstances in which we might want to create special subzones in which to place our bug objects.)

Once the bug is created, we send it on its way, making 100 random walks and reporting its position at the end of each one by calling upon the bug's randomWalk() and reportPosition() methods.

/SimpleJavaBug/StartSimpleBug.java

```

// StartSimpleBug.java
// The Java SimpleBug application.

import swarm.Globals;
import swarm.defobj.Zone;

public class StartSimpleBug
{
    // The size of the bug's world and its initial position.
    static int worldXSize = 80;
    static int worldYSize = 80;
    static int xPos = 40;
    static int yPos = 40;

    public static void main (String[] args)
    {
        int i;
        SimpleBug abug;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("SimpleBug", "2.1",
            "bug-swarm@santafe.edu", args);
    }
}

```

```
// Create an instance of a SimpleBug, abug, and place it
// within its world at (xPos, yPos). The bug is created in
// Swarm's globalZone and is given a "bug id" of 1.
abug = new SimpleBug(Globals.env.globalZone, worldXSize, worldYSize,
                    xPos, yPos, 1);

// Loop our bug through a series of random walks asking it to
// report its position after each one.
for (i = 0; i < 100; i++)
{
    abug.randomWalk();
    abug.reportPosition();
}
}
```

The **Makefile** for our application simply lists the two Java source files.

/SimpleBug/Makefile

```
JAVA_SRC = SimpleBug.java StartSimpleBug.java

all: $(JAVA_SRC)
    $(SWARMHOME)/bin/javacswarm $(JAVA_SRC)

clean:
    rm -f *.class
```

If we type "javaswarm StartSimpleBug" at the console, we should get the birth announcement for our bug, followed by 100 reports on its position.

/SimpleJavaBug2

FROM THE LONE BUG TO HOME ON THE RANGE

In this version, we extend the previous model by providing the bug with a spatial "world" with which it can interact. In this case, the spatial world has "food" that the bug "eats" as it wanders, removing that food from the world. For obvious reasons, we call this world the bug's foodspace.

A foodspace is a two-dimensional lattice with either 1 (for food) or 0 (for no food) at each integer coordinate. We could easily construct this grid as a Java integer array, but in the interest of object-orientation, we will instead create a foodspace object. The Swarm library provides a template for creating such an object in the *Discrete2dImpl* class and we use it for our foodspace because it comes with some useful methods. *Discrete2dImpl* is basically a lattice of integer values and it has methods defined on it that can set its size, get and set values at sites in the lattice, and so forth.

The FoodSpace class is defined in the file **FoodSpace.java** as a subclass of *Discrete2dImpl*. (Note that we import *swarm.space.Discrete2dImpl* to provide the class descriptor.) The FoodSpace constructor is called with the zone in which the foodspace object is to be created and with the desired dimensions of the foodspace. The FoodSpace constructor first calls the constructor for the parent class, passing on the zone and the dimensions, and then fills itself with zeros using *fastFillWithValue()*, a method inherited from *Discrete2dImpl*. (Since the *Discrete2dImpl* constructor creates an empty lattice, we don't really need to fill the lattice with zeros. On the other hand, it never hurts to be sure!)

The FoodSpace class has only one method of its own, *seedFoodWithProb()*. This method is used to distribute food randomly across the foodspace. The method visits every point on the foodspace lattice and deposits food there with a probability *seedProb*. The *seedFoodWithProb()* method in turn uses four methods. The first is *uniformDblRand.getDoubleWithMin\$withMax()*. This is a method contained in Swarm's *Globals* environment and is the real number counterpart of the integer method we used previously. The second method is *putValue\$atX\$Y()*, a method inherited by the FoodSpace class from its *Discrete2dImpl* parent. This method puts an integer value at a particular location on the lattice. The third and fourth methods are also inherited from the *Discrete2dImpl* parent class: *getSizeX()* and *getSizeY()* return the X and Y dimension respectively of the foodspace lattice. (Note, we could have saved the dimensions as instance variables within the FoodSpace class itself when they were passed to it, but its best here to demonstrate all that the Swarm classes can do.)

/SimpleBug2/FoodSpace.java

```
// FoodSpace.java
// Defines the FoodSpace class as a subclass of Discrete2dImpl.
```

```

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.space.Discrete2dImpl;

public class FoodSpace extends Discrete2dImpl
{
    // The constructor for a foodspace. Create a foodspace in zone
    // aZone with dimensions xSize and ySize. Note that a
    // food space is based on the Swarm's Discrete2dImpl class.
    public FoodSpace(Zone aZone, int xSize, int ySize)
    {
        // Call the constructor for the parent class and then fill the
        // lattice with zeros.
        super(aZone, xSize, ySize);
        fastFillWithValue(0);
    }

    // Method to randomly seed the foodspace with food.
    public void seedFoodWithProb(double seedProb)
    {
        int x, y;

        // Visit each cell and seed it with food with probability
        // seedProb. Note that the getDoubleWithMin$WithMax method has
        // been provided by the call to InitSwarm in
        // StartSimpleBug.java. Note too that we depend upon the getXSize()
        // and getYSize() methods inherited from the Discrete2dImpl
        // class to retrieve the dimensions of the foodspace.
        for (y = 0; y < getSizeY(); y++)
            for (x = 0; x < getSizeX(); x++)
                if (Globals.env.uniformDblRand.getDoubleWithMin$withMax(0.0, 1.0)
                    <= seedProb )
                    putValue$atX$Y(1, x, y);
    }
}

```

Now that we have a foodspace, we need to make one simple change to the behavior of our SimpleBugs: when a bug finds food, it should eat it. The new **SimpleBug.java** file shows this change in two places. First, we have altered the constructor for a SimpleBug to allow us to pass the bug's foodspace to the newly created bug. The constructor saves the foodspace in a new instance variable, myFoodSpace, and a SimpleBug now gets the dimensions of its world from its foodspace, using *getSizeX()* and *getSizeY()*. We also have altered the randomWalk() method to check for food at the location to which the bug has walked. Again, the *getValueAtX\$Y()* method has been inherited from *Discrete2dImpl* and thus is defined for myFoodSpace, the bug's instance of a FoodSpace. If the bug finds food (i.e. sees a 1 at its position on myFoodSpace), it eats it by setting the value at that location to zero. The bug also reports its find to the console.

/SimpleBug2/SimpleBug.java

```
// SimpleBug.java
```

```
// Defines the class for our SimpleBug agents/

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

public class SimpleBug extends SwarmObjectImpl
{
    // These instance variables keep track of a given bug's foodspace,
    // position and identity. We also save the dimensions of the
    // foodspace so that we can make fewer calls to the getSizeX() and
    // getSizeY() methods in the bug's randomWalk().
    FoodSpace myFoodSpace;
    int xPos;
    int yPos;
    int bugNumber;

    int worldXSize;
    int worldYSize;

    // Constructor to create a SimpleBug object in Zone aZone and to
    // place it in the foodspace, fSpace, at the specified X,Y
    // location. The bug is also given a numeric id, bNum.
    public SimpleBug(Zone aZone, FoodSpace fSpace, int X, int Y,
        int bNum)
    {
        // Call the constructor for the bug's parent class.
        super(aZone);

        // Record the bug's foodspace, initial position and id number.
        myFoodSpace = fSpace;
        worldXSize = myFoodSpace.getSizeX();
        worldYSize = myFoodSpace.getSizeY();
        xPos = X;
        yPos = Y;
        bugNumber = bNum;

        // Announce the bug's presence to the console.
        System.out.println("SimpleBug number " + bugNumber +
            " has been created at " + xPos + ", " + yPos);
    }

    // This is the method to have the bug take a random walk backward
    // (-1), forward (+1), or not at all (0) in first the X and then
    // the Y direction. The randomWalk method uses
    // getIntegerWithMin$withMax() to return an integer between a
    // minimum and maximum value, here between -1 and +1.
    // Globals.env.uniformRand is an instance of the class
    // UniformIntegerDistImpl, instantiated by the call to
    // Globals.env.initSwarm in StartSimpleBug. Note that the bug's
    // world is a torus. If the bug walks off the edge of its
    // rectangular world, it is magically transported (via the modulus
    // operator) to the opposite edge. If on its walk the bug finds
    // food, it eats it.
    public void randomWalk()
    {
        xPos += Globals.env.uniformIntRand.getIntegerWithMin$withMax(
```

```

                -1, 1);
yPos += Globals.env.uniformIntRand.getIntegerWithMin$withMax(
                -1, 1);
xPos = (xPos + worldXSize) % worldXSize;
yPos = (yPos + worldYSize) % worldYSize;

// If there is food at this cell, eat it!
if (myFoodSpace.getValueAtX$Y(xPos, yPos) == 1)
{
    myFoodSpace.putValue$atX$Y(0, xPos, yPos);
    System.out.println("Bug " + bugNumber + " has found food at " + xPos
        + ", " + yPos);
}
}

// Method to report the bug's position to the console.
public void reportPosition()
{
    System.out.println("Bug " + bugNumber + " is at " + xPos +
        ", " + yPos);
}
}

```

The introduction of a foodspace requires only minor changes to **StartSimpleBug.java**. Before creating our bug we need to create a foodspace for it to walk in and to seed that foodspace with food. When we next create our bug, we pass it that newly created foodspace object, `foodSpace`. Finally, we instruct the bug to wander around as before, reporting on where it is and what it finds.

/SimpleBug2/StartSimpleBug.java

```

// StartSimpleBug.java
// The Java SimpleBug application.

import swarm.Globals;
import swarm.defobj.Zone;

public class StartSimpleBug
{
    // The size of a bug's world.
    static int worldXSize = 80;
    static int worldYSize = 80;

    // The probability for seeding the foodspace.
    static double seedProb = 0.5;

    public static void main (String[] args)
    {
        int i, xPos, yPos;
        SimpleBug abug;
        FoodSpace foodSpace;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("SimpleBug", "2.1",

```

```
        "bug-swarm@santafe.edu", args);

// Create a foodspace of the desired dimension.
foodSpace = new FoodSpace(Globals.env.globalZone,
                          worldXSize, worldYSize);

// Seed the foodspace with food.
foodSpace.seedFoodWithProb(seedProb);

// To place our new SimpleBug in the middle of its foodspace,
// use foodSpace's own knowledge of its dimensions to find the
// middle.
xPos = (foodSpace.getSizeX())/2;
yPos = (foodSpace.getSizeY())/2;

// Create an instance of a SimpleBug, abug, and place it
// within its foodSpace world at (xPos, yPos). The bug is
// created in Swarm's globalZone and is given a "bug id" of
// 1. Since foodSpace knows its dimensions, abug can get them
// directly from the foodSpace. We no longer have to provide
// them in creating a SimpleBug.
abug = new SimpleBug(Globals.env.globalZone, foodSpace,
                    xPos, yPos, 1);

// Loop our bug through a series of random walks in its
// foodspace, asking it to report its position after each one.
// The bug itself will report when it finds food.
for (i = 0; i < 100; i++)
{
    abug.randomWalk();
    abug.reportPosition();
}
}
```

The new **Makefile** adds `FoodSpace.java` to the list of files in our model. Compiling and running the model results in our bug reporting its position and the food it finds. (You may want to delete in `StartSimpleBug.java` the request for the bug to report its position after every move so that you can concentrate on the food reports.) Increasing and decreasing the value of `seedProb` will increase and decrease the frequency of the bug's finding food.

Swarm's Java API

Since we will be introducing a number of new Swarm classes in the following enhancements of our SimpleBug model, this might be a good time to talk very briefly about Swarm's Java API. The [Java Reference Guide to Swarm 2.1](#) is available in hypertext form on the Swarm web site. The *Java Reference Guide* lists each Swarm library package and its constituent interfaces and classes. Under each interface or class one can then find the variables and methods defined for that interface or class. As with most Java classes, there is a class hierarchy with variables and methods being passed down from parent to child.

There are basically two ways to construct an instance of one of the classes provided by the Swarm package. The most common in Java applications is to construct an object in one step, calling the class constructor with the arguments required to create the desired instance of the class. Classes which can be created in this way are called "implementation classes" and have class names ending in "Impl". We have already seen two such classes, *SwarmObjectImpl* and *Discrete2dImpl*. Implementation class constructors are often overloaded, that is they can be called with different numbers of arguments. For instance, a *Discrete2dImpl* object can be constructed as

```
foodSpace = new Discrete2dImpl(aZone, X, Y);
```

or as

```
foodSpace = new Discrete2dImpl(aZone);
```

In the first case the lattice is constructed in a specified zone and with the specified dimensions. In the second case, the dimensions of the lattice are left unspecified.

It is also possible to create an object in a manner akin to that used in Objective-C, beginning with *createBegin()*, setting one or more attributes of the object, and ending with *createEnd()*. Constructing objects in this way provides considerable flexibility (for instance, it is especially useful in constructing random number generators), but we will stick to the one-step implementation classes in this tutorial.

The frequent use of the dollar sign in the naming of Swarm class methods may appear a little strange to Java programmers. Method names in Swarm's Java interface are taken directly from Swarm's Objective-C interface where a key/value syntax is used. For instance, the *putValue\$atX\$Y(value, x, y)* method in the *Discrete2d* class in Java would be *putValue: value atX: x Y: y* in Objective-C. The dollar signs in the Java name then mark the points where arguments would occur in the Objective-C implementation. They give the programmer hints as to what arguments are required and in what order.

/SimpleSwarmBug

ADDING A SWARM

One bug is nice, but we're headed for more. In this version of the program, we create a object of the class *Swarm* to take care of creating and managing what will soon be a whole host of bugs in their artificial world. For the moment, though we stick with one bug and concentrate on the *Swarm* object itself.

In a typical Swarm application, we create a top-level *Swarm* object that manages the tasks of creating, running and interacting with our models. The Swarm object is not really a part of the bug's world, it is rather an object in our world. The *Swarm* object encapsulates our model of the bug and its world, making the model itself a kind of object that we can interact with by sending it messages and asking it to do things.

We might think of a *Swarm* object as an container into which we place our simulation agents and one or more lists of activities those agents are to perform. Those lists are, of course, objects in their own right and are defined by Swarm's *Activity* class.

As we don't know (or really even care) at this level what the model is, we have a regular procedure for constructing almost any top-level *Swarm* object - in fact almost any *Swarm* object - that is independent of the specifics of the model. Instances of a Swarm typically have three tasks: to construct the various objects in the model, to arrange for messages to be sent to the objects so that they do what we want them to do, and finally to glue everything together to form a package that we can run. We will examine each of these tasks, in order, in the context of our own top-level *Swarm* class, "ModelSwarm", defined in the file **ModelSwarm.java**.

Our ModelSwarm class is an extension of the *SwarmImpl* class (the implementation class for a *Swarm*) and inherits a number of useful methods from it. The constructor for our ModelSwarm is trivial. Having declared our model's now familiar parameters and their default values as instance variables, we simply call upon the constructor of ModelSwarm's parent class.

The first task of a ModelSwarm is to build the objects that are used in the model. This is done in the `buildObjects()` method and this method builds the same objects that in the SimpleJavaBug2 application were built in `StartSimpleBug.java`: we create a foodspace, seed it with food and create a bug to wander over that foodspace. Before building our objects, however, we first call upon the `buildObjects()` method in our parent class to have Swarm perform a number of preparatory steps behind the scenes. (The Swarm package takes care of a lot of details that we do not have to worry about if we but ask it to do so.) Note that `buildObjects()` requires that we return the ModelSwarm object itself.

The second task, `buildActions()`, creates a group of messages to send to the bug, that is a packaged list of actions for the bug to accomplish. It then puts that packaged list of messages into a schedule that will arrange for them to be sent to the bug at the

appropriate times. All this requires three new classes of Swarm objects, the *ActionGroup*, *Selector* and *Schedule* classes. (We will be using the implementation forms of first and third of these classes, *ActionGroupImpl* and *ScheduleImpl*.) *ActionGroups* and *Schedules* are subclasses of Swarm's *Activity* class and they are basically specialized containers designed to hold messages or groups of messages destined for other objects. Before a message can be inserted into an *ActionGroup* or *Schedule*, however, it must itself be encapsulated in yet another kind of object called a *Selector*. Let's walk through the `buildActions()` method.

As with `buildObjects()`, we first call upon the corresponding method in the parent class to do all the initializations we'd prefer not to worry about. We then create a new *ActionGroup* object, `modelActions`, into which we will put messages to our bug. Note that we create `modelActions` using the *ActionGroup* implementation class, *ActionGroupImpl*.

To specify a message, we need to specify the message name, that is the method that performs the action we desire, and the class in which that message is defined. These two pieces of information are encapsulated in a *Selector*, an object that is specially designed to hold them. The constructor for a *Selector* takes three arguments:

```
Selector sel = new Selector(Class theClass, String theMethodName,  
                           boolean theObjcflag);
```

The second argument is the name of the method. In `buildObjects()` we want to send the "randomWalk" message to our *SimpleBug*, `abug`. I.e., we want to call upon `abug`'s `randomWalk` method. The first argument is the class in which that method is defined. Here that is the *SimpleBug* class, the class to which the object `abug` belongs. The *SimpleBug* class id is returned by the `getClass()` method which all objects in Java (including `abug`) inherit. The third argument allows the use of Objective-C key/value syntax. We do not use that syntax in this tutorial so the flag will always be set to false.

We want to create a *Selector*, `sel`, to encapsulate the `randomWalk` message to a *SimpleBug*, and then add that selector to our *ActionGroup*, `modelActions`. Let's look first at the creation of `sel`.

Because our models may need to create messages and thus *Selectors* on the fly during a simulation, the correspondence between class and method is not checked until runtime. It is therefore possible that there could be a mismatch between the two arguments that will generate an exception (runtime error) when the *Selector*, `sel`, is created. (For instance, we could have mistyped the method name as "RandomWalk", which is not, of course, in the *SimpleBug* class.) The Java compiler knows that such exceptions might occur and insists that *Selectors* be created within try/catch blocks. Moreover, the compiler insists that any use of the resulting *Selector* be within the try block as well, since `sel` would not refer to a valid *Selector* if indeed an exception were to occur in its creation. Therefore, both the creation of the *Selector* and its insertion into `modelActions` are within the try block. If an exception does occur, it is "caught." A message is sent to the console and we bail out

(ungracefully) by calling the Java method `System.exit(1)`. Continuing would serve no purpose.

Now let's look at the addition of `sel` to `modelActions`. We have an action message, but in adding it to `modelActions` we need to specify the object or objects to which that message is to be sent. The method `createActionTo$message(Object abug, Selector sel)` specifies that the message encapsulated in `sel` is to be sent to the particular object, `abug`. We will see later that there are methods to send a message to a whole collection of objects as well.

Typically we want to send messages to the agents and actors in our model, but there are times when we need to send messages to other objects in our application as well, for instance the objects that control the simulation or display its results. Since a message is a message, regardless of its destination, these messages can be inserted in *Activity* containers as well. For reasons which will become clear shortly, we want to have our SimpleSwarmBug application check the time - that is the simulated time - upon each occasion that our bug is told to take a walk. We've provided a method, `checkTime()`, in our own `ModelSwarm` class to accomplish this and so we encapsulate a "checkTime" message to `modelSwarm` in a *Selector* object and insert it too into `modelActions`.

When there is more than one message in an *ActionGroup*, in what order are they sent when the *ActionGroup* is executed? By default, the messages are sent in the order in which they were inserted, but there is an option for randomizing the order if that is desirable.

Having created and filled our *ActionGroup*, it is finally time to insert `modelActions` into a *Schedule*. A *Schedule* is yet another container object into which we can insert either individual messages, or groups of messages that are contained in *ActionGroups*. The *Schedule* then keeps track of the intervals in simulated time at which the messages it contains (directly, or indirectly through an *ActionGroup*) are to be sent. There are several attributes that can be set for a *Schedule*. We will concern ourselves with only one, however. We want the *Schedule* to deliver `modelActions` for execution each and every time period. This interval between actions is the second argument in the constructor for *ScheduleImpl* and we set it to unity. (Had we wished the messages to be sent only every other period we would have specified an interval of 2.) Once the *Schedule* object is created as `modelSchedule`, we insert the `modelActions` *ActionGroup* in the schedule with `modelSchedule`'s `createAction()` method. The first argument specifies that the first kickoff of `modelActions` is to be at period zero, that is in very first period. (You will notice that we have not told `modelSchedule` how long we want it to continue to send off messages. That is not one of it's capabilities. It keeps going until told to stop.)

We've now finished setting up the actions in our model and we return from `buildActions()` with a pointer to `modelSwarm`.

The final task in setting up a `ModelSwarm` is to glue all the pieces together into a package that can be executed. This is done by the `activateIn()` method which returns yet another *Activity* object, an executable object containing all the work we have packaged up

in our *ActionGroup(s)* and *Schedule(s)*, along with some objects that Swarm adds for us. *activateIn()* is given a "context", essentially the *Swarm* in which this *Activity* will be executed. *Swarms* (and our *ModelSwarm* subclass) are hierarchical. (Sub)*Swarms* may operate inside other *Swarms*. For our top-level *ModelSwarm* this context will be null, indicating that *ModelSwarm* is indeed at the top of the hierarchy. As usual, we use the corresponding method for the parent class to initialize everything and then we activate *modelSchedule* in the context of *modelSwarm*. (You can see the *Swarm* hierarchy at work here.) Finally, we return the activity we have built.

Note that we don't actually "start" the model going here. We've used our *ModelSwarm* to create the model and to encapsulate it in an *Activity* object. It will be up to some other part of the *Swarm* package to actually execute the *Activity* object by traversing the data structures it contains.

The file *ModelSwarm.java* ends with the *checkTime()* method. You will remember that we scheduled a message to *modelSwarm* to execute this method at each time step. We did so for two reasons. First, it will periodically send a timestamp to the console to let us know what the simulation time is. Second, it will stop the simulation after a predetermined time given by the *endTime* variable. This latter task is required because *modelSchedule* was told to send its messages every period, forever!

checkTime() first gets the current simulation time by calling a method in *Swarm's* Global environment, *getCurrentTime()*. It then checks to see whether we have reached *endTime*. If we have, *checkTime()* uses the *terminate()* method for the current *Activity* object to stop the simulation. (That object is returned by yet another method in the Global environment, *getCurrentActivity()*.) If we have not reached the end of our simulation and if the time is an integer power of 5, we send a timestamp to the console.

We've set *endTime* to a pretty large number to allow the bug ample opportunity to empty its foodspace. There are other approaches to setting a termination time for a simulation including the creation of a special schedule dedicated to this purpose. *jheatbugs* has an example of this approach. Later on, we will stop the simulation in a much less arbitrary way.

/SimpleSwarmBug/ModelSwarm.java

```
// ModelSwarm.java
// The top-level ModelSwarm.

import swarm.Globals;
import swarm.Selector;
import swarm.defobj.Zone;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;

import swarm.activity.ActionGroupImpl;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;
```

```
public class ModelSwarm extends SwarmImpl
{
    // Declare the model parameters and their default values.
    public int worldXSize = 80, worldYSize = 80;
    public double seedProb = 0.20;
    public int endTime = 20000;

    // Declare some other needed variables.
    FoodSpace foodSpace;
    SimpleBug abug;
    ScheduleImpl modelSchedule;

    // This is the constructor for a new ModelSwarm. All we do is to
    // use the constructor for ModelSwarm's parent class.
    public ModelSwarm(Zone aZone)
    {
        // Use the parent class to create a top-level swarm.
        super(aZone);
    }

    // This is the method for building the model's objects: in this
    // case the food space and the bug.
    public Object buildObjects()
    {
        int xPos, yPos;

        // Use the parent class buildObject() method to initialize the
        // process.
        super.buildObjects();

        // Now create the model's objects.
        // First create the foodspace and seed it with food.
        foodSpace = new FoodSpace(Globals.env.globalZone,
                                   worldXSize, worldYSize);
        foodSpace.seedFoodWithProb(seedProb);

        // Find the middle of the foodspace.
        xPos = (foodSpace.getSizeX())/2;
        yPos = (foodSpace.getSizeY())/2;

        // Create a single SimpleBug, abug, and place it in the
        // foodspace at (xPos, yPos). The bug knows the size of its
        // world from being passed the pointer to the foodspace in
        // which it is to be created.
        abug = new SimpleBug(Globals.env.globalZone, foodSpace,
                              xPos, yPos, 1);

        return this;
    }

    // This is the method a) for building the list of actions for
    // these objects to accomplish and b) for scheduling these actions
    // in simulated time.
    public Object buildActions()
    {

```

```

Selector sel;
ActionGroupImpl modelActions;

// First, use the parent class to initialize the process.
super.buildActions();

// Then create an ActionGroup object and tell it to send a
// randomWalk message to abug. Since the
// createActionTo$message() method requires a selector object to
// "contain" the method, and since the creation of a selector
// can throw an exception, that process is included in a
// try/catch block. The first argument, abug.getClass(),
// retrieves the class to which abug belongs while the second
// argument, "randomWalk", is the name of the particular abug
// method we wish to add to the list of actions. Note that
// randomWalk must have been declared public.
modelActions = new ActionGroupImpl(getZone());
try
{
    sel = new Selector(abug.getClass(), "randomWalk", false);
    modelActions.createActionTo$message(abug, sel);
} catch (Exception e)
{
    System.err.println("Exception randomWalk: " +
        e.getMessage ());
    System.exit(1);
}

// Next we add to modelActions another message, this one to
// modelSwarm itself telling it to check the simulation
// time. We use this to send a timestamp to the console and to
// stop the model after a specified number of periods.
try
{
    sel = new Selector(this.getClass(), "checkTime", false);
    modelActions.createActionTo$message(this, sel);
} catch (Exception e)
{
    System.err.println("Exception checkTime " + e.getMessage ());
    System.exit(1);
}

// Now create the schedule and set the repeat interval to unity.
modelSchedule = new ScheduleImpl(getZone(), 1);

// Finally, insert the action list into the schedule at period zero
modelSchedule.at$createAction(0, modelActions);

return this;
}

// This method specifies the context in which the model is to be run.
public Activity activateIn(Swarm swarmContext)
{
    // Use the parent class to activate ourselves in the context
    // passed to us.
    super.activateIn(swarmContext);
}

```

```

// Then activate the schedule in ourselves.
modelSchedule.activateIn(this);

// Finally, return the activity we have built.
return getActivity();
}

// This is a pretty crude method to end the simulation after an
// arbitrary number of periods given by the endTime parameter. If
// the simulation time returned by getCurrentTime() is greater
// than endTime, we send a message to the console and terminate
// the current activity. We also use this method to send a
// timestamp to the console.
public void checkTime()
{
    int i, t;
    int interval = 5;

    if (Globals.env.getCurrentTime() >= endTime)
    {
        // Terminate the simulation.
        System.out.println("We've reached our endTime at period " +
            Globals.env.getCurrentTime());
        Globals.env.getCurrentSwarmActivity().terminate();
    }
    else
    {
        for (i = 0; (t = (int)Math.pow(interval, (double)i))
            <= endTime; i++)
            if (t == Globals.env.getCurrentTime())
                System.out.println("The time is " + t);
    }

    return;
}
}

```

We've introduced a number new concepts in building the ModelSwarm. Fortunately, the hard work is done. We've made no changes to the nature or capabilities of our SimpleBugs or FoodSpace, and therefore there are no changes in these files at all. We need only to change **StartSimpleBug.java** to reflect the new ModelSwarm. After the usual `initSwarm()` step, we create a ModelSwarm object as `modelSwarm` and run through the three tasks we discussed above, `buildObjects()`, `buildActions()` and `activateIn()`. Note that we activate `modelSwarm` in the null context, indicating that it is the top-level Swarm. Finally, we are ready to run the *Activity* object constructed in our `modelSwarm` by first getting the *Activity* using `modelSwarm's getActivity()` method and then running it with the `run()` method of the *Activity* class.

```
(modelSwarm.getActivity()).run()
```

(Note that we could have combined the activation of `modelSwarm` with the running of its *Activity* since `activateIn()` returns the *Activity* that was built.

```
(modelSwarm.activateIn(null)).run() )
```

/SimpleSwarmBug/StartSimpleBug.java

```
// StartSimpleBug.java
// Java SimpleBug application.

import swarm.Globals;
import swarm.defobj.Zone;

public class StartSimpleBug
{
    public static void main (String[] args)
    {
        ModelSwarm modelSwarm;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("SimpleBug", "2.1",
                               "bug-swarm@santafe.edu", args);

        // Create a top-level Swarm object and build its internal
        // objects and activities.
        modelSwarm = new ModelSwarm(Globals.env.globalZone);
        modelSwarm.buildObjects();
        modelSwarm.buildActions();
        modelSwarm.activateIn(null);

        // Now activate the swarm.
        (modelSwarm.getActivity()).run();
    }
}
```

The **Makefile** for our application changes only by the addition of `ModelSwarm.java` to the list of source files. When the application is compiled and run, our single bug will wander through its foodspace until `endTime`, eating food as it comes upon it and reporting its feasts to the console. After a while, the bug will have exhausted all the food in the foodspace and we will hear from it no more. (Think about the impact of different values of `seedProb` on how long we continue to hear from the bug.)

/SimpleSwarmBug2

MANAGING MORE AGENTS

It is now time to move to a multi-agent world, creating an arbitrary number of bugs and having them compete for food. Since we have encapsulated our model in **ModelSwarm.java**, almost all of the changes we need to make will be made there.

In order to keep track of our collection of bugs we will need two new Swarm objects. The first is a *List* object to encapsulate our many bugs and allow us to treat them as a unit. For instance, instead of having to communicate with each bug separately, we can send a message to the *List* object and have it forward that message to every bug it contains. The second is a *Grid2d* object, a lattice much like *Discrete2d* except that it holds the ids of other objects rather than integers. Our *Grid2d* object, our "bugspace," will record the location of each of our bugs and, in the process, make sure that no two bugs are at the location at the same time.

It is useful to think of the bugspace as being superimposed on top of the foodspace. Our bugs walk around on the former and eat any food they find on the foodspace "underneath" them.

The changes we make to **ModelSwarm.java** are largely in the `buildObjects()` method. After creating our foodspace, we create `bugSpace`, a *Grid2d* object, by using its implementation class, *Grid2dImpl*. (Note that we have added `swarm.space.Grid2dImpl` to our import list.) It is, of course, the same size as `foodSpace`. We then fill `bugSpace` with nulls to indicate that there are as yet no bugs in it.

The next step is to create a *List* object, `bugList`, to hold our bugs. *List* objects can expand as they are filled so we don't have to know in advance how many bugs we will have. Again we use the implementation class, *ListImpl*, and include it in our list of imports. We then traverse all the points in `bugSpace`, creating a bug at each point with a probability given by `bugDensity`, a new parameter for our model, using the same `getDoubleWithMin$withMax()` method we have been using in seeding the foodspace. As each bug is created, it is placed on `bugSpace` using `putObjectatXY(abug, x, y)`, a method defined for the *Grid2d* class. The bug is then added to the end of `bugList` by using `addLast(abug)`, a method defined on *List* class.

Because we don't want each of our potentially large number of bugs to report its finding of food to the console, we choose one bug to be a "reporter bug." We could choose any bug, but it is easy to use the first bug created, that is the first bug in `bugList`. A *List* object works something like a two-ended stack. We can on either end push things on (`addFirst()`, `addLast()`) or pop things off (`removeFirst()`, `removeLast()`) the list. We therefore get the identity of our reporter bug by removing the first bug on `bugList`, saving its id in the variable `reportBug`, and then pushing it back on. (We might also have used a single non-destructive method such as `getFirst()` or `atOffset(0)`, which return the object without removing it from the list.)

In the `buildActions()` section of `ModelSwarm.java` there is one change and one addition. The change comes in how we send the "randomWalk" message to our bugs. We now have many bugs to communicate with and it would be tedious to insert into `modelActions` a message to each bug individually. Luckily, our `bugList` object comes to the rescue. We can send the message to `bugList` and let `bugList` take care of forwarding the message to all the objects it contains. To do this we use a different method for inserting an action in an *ActionGroup*, `createActionForEach$message()`, which takes as its arguments the *List* to which the message is being sent and the *Selector* containing the message.

The addition to `buildActions()` is that after every walk, we want to ask the reporter bug to tell us if it has eaten. We've added a method to the `SimpleBug` class to do just that, `reportIfEaten()`. In the now familiar way, we package up the message to reportBug in a *Selector* and insert it in `modelActions`, using the `createActionTo$message()` method since the destination of the message is a particular bug.

The rest of `ModelSwarm.java` remains unchanged from the previous version.

/SimpleSwarmBug2/ModelSwarm.java

```
// ModelSwarm.java
// The top-level ModelSwarm

import swarm.Globals;
import swarm.Selector;
import swarm.defobj.Zone;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;

import swarm.activity.ActionGroupImpl;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.space.Grid2dImpl;
import swarm.collections.ListImpl;

public class ModelSwarm extends SwarmImpl
{
    // Declare the model parameters and their default values.
    public int worldXSize = 80, worldYSize = 80;
    public double seedProb = 0.80;
    public double bugDensity = 0.10;
    public int endTime = 125;

    // Declare some other needed variables.
    public FoodSpace foodSpace;
    public Grid2dImpl bugSpace;
    public ListImpl bugList;
    public ScheduleImpl modelSchedule;
    public SimpleBug reportBug;
```

```

// This is the constructor for a new ModelSwarm. All we do is to
// use the constructor for ModelSwarm's parent class.
public ModelSwarm(Zone azone)
{
    // Use the parent class to create a top-level swarm.
    super(azone);
}

// This is the method for building the model's objects: the food
// space, the two-dimensional positioning grid, and the host of
// bugs.
public Object buildObjects()
{
    int x, y, num;
    SimpleBug abug;

    // use the parent class buildObject() method to initialize the
    // process
    super.buildObjects();

    // Now create the model's objects.
    // First create the foodspace and seed it with food.
    foodSpace = new FoodSpace(Globals.env.globalZone,
                               worldXSize, worldYSize);
    foodSpace.seedFoodWithProb( seedProb );

    // Then create a 2-D grid that will be used to keep track of
    // each bug's position, insuring that no two bugs will ever be
    // on the same cell. Initialize the grid to be empty.
    bugSpace = new Grid2dImpl(Globals.env.globalZone,
                               worldXSize, worldYSize);
    bugSpace.fastFillWithObject( null );

    // Now create a List object to manage all the bugs we are
    // about to create.
    bugList = new ListImpl(Globals.env.globalZone);

    // Iterate over the grid with a certain probability of
    // creating a bug at each site. If a bug is created, put it
    // on the grid and add it to the end of the bug list. Note
    // that we increment the bug number, num, each time a bug is
    // created.
    num = 0;
    for (y = 0; y < worldYSize; y++)
        for (x = 0; x < worldXSize; x++)
            if ( Globals.env.uniformDblRand.getDoubleWithMin$withMax(
                    0.0, 1.0) <= bugDensity)
                {
                    abug = new SimpleBug(Globals.env.globalZone, foodSpace,
                                         bugSpace, x, y, ++num);
                    bugSpace.putObject$atX$Y(abug, x, y);
                    bugList.addLast(abug);
                }

    // Finally, enlist a "reporter" bug to let us know how things
    // are going. We just pop the first bug off the list, record
    // its id, and return it to the list.

```

```

reportBug = (SimpleBug)bugList.removeFirst();
bugList.addFirst(reportBug);
System.out.println("The lucky reporter bug is bug number " +
    reportBug.bugNumber + ".");

// We're done.
return this;
}

// This is the method a) for building the list of actions for
// these objects to accomplish and b) for scheduling these actions
// in simulated time.
public Object buildActions()
{
    Selector sel;
    ActionGroupImpl modelActions;

    // First, use the parent class to initialize the process.
    super.buildActions();

    // Then create an ActionGroup object and tell it to send a
    // randomWalk message to every bug in the bug list. Note the
    // change here from using the method createActionTo$message()
    // to send a message to a single object, to using the method
    // createActionForEach$message() to send the same message to
    // all the objects in a list.
    modelActions = new ActionGroupImpl(getZone());
    try
    {
        // Note the use here of the forName() method of the
        // "Class" class to pass the first argument to the
        // creation of the selector. The first argument to the
        // creation of a selector requires an object of type Class
        // that identifies the class of the object to which the
        // message should be sent, in this case a SimpleBug. If we
        // had an instance of a SimpleBug available, say abug,
        // that first argument could use the getClass() method
        // derived for all classes from the superclass, Object.
        // The argument would be "abug.getClass()". Indeed, we
        // can use getClass() when we create below the selector
        // for the reportIfEaten message to the reportBug since
        // reportBug is a created object. (Of course, reportBug
        // is an instance of a SimpleBug and we could use
        // reportBug.getClass() instead of
        // Class.forName("SimpleBug") in creating our randomWalk
        // selector, but an instance such as reportBug might not
        // always be available and it's good to know that we don't
        // really need one.)
        sel = new Selector(Class.forName("SimpleBug"),
            "randomWalk", false);
        modelActions.createActionForEach$message(bugList, sel);
    } catch (Exception e)
    {
        System.err.println("Exception randomWalk: " +
            e.getMessage ());
        System.exit(1);
    }
}

```

```

// Next we will create a message to the report bug to tell us
// what it is doing, and add that message to the ActionGroup.
try
{
    sel = new Selector(reportBug.getClass(),
        "reportIfEaten", false);
    modelActions.createActionTo$message(reportBug, sel);
} catch (Exception e)
{
    System.err.println("Exception reportIfEaten: " +
        e.getMessage ());
    System.exit(1);
}

// Our last addition to the ActionGroup is the message to
// modelSwarm itself to check the simulation time. We use this
// to send a timestamp to the console and to stop the model
// after a specified number of periods.
try
{
    sel = new Selector(this.getClass(), "checkTime", false);
    modelActions.createActionTo$message(this, sel);
} catch (Exception e)
{
    System.err.println("Exception checkTime " + e.getMessage ());
    System.exit(1);
}

// Now create the schedule and set the repeat interval to unity.
modelSchedule = new ScheduleImpl(getZone(), 1);

// Finally, insert the action list into the schedule at period zero
modelSchedule.at$createAction(0, modelActions);

return this;
}

// This method specifies the context in which the model is to be run.
public Activity activateIn(Swarm swarmContext)
{
    // Use the parent class to activate ourselves in the context
    // passed to us.
    super.activateIn(swarmContext);

    // Then activate the schedule in ourselves.
    modelSchedule.activateIn(this);

    // Finally, return the activity we have built.
    return getActivity();
}

// This is a pretty crude method to end the simulation after an
// arbitrary number of periods given by the endTime parameter. If
// the simulation time returned by getCurrentTime() is greater
// than endTime, we send a message to the console and terminate
// the current activity. We also use this method to send a

```

```

// timestamp to the console.
public void checkTime()
{
    int i, t;
    int interval = 5;

    if (Globals.env.getCurrentTime() >= endTime)
    {
        // Terminate the simulation.
        System.out.println("We've reached our endTime at period " +
            Globals.env.getCurrentTime());
        Globals.env.getCurrentSwarmActivity().terminate();
    }
    else
    {
        for (i = 0; (t = (int)Math.pow(interval, (double)i))
            <= endTime; i++)
            if (t == Globals.env.getCurrentTime())
                System.out.println("The time is " + t);
    }

    return;
}
}

```

Our SimpleBug class has seen some changes as well. First, each bug must be made aware of the bugspace through which it wanders and we pass the bugspace to each bug through its constructor. Second, we need to make sure that one bug does not try to occupy the same space as another. When contemplating its randomWalk, each bug decides (randomly) where it would like to go, but checks before it moves to see that its destination is empty (that is if `myBugSpace.getObjectAtX$Y(newX, newY) == null`). If there is no bug at that location, the bug puts itself there. If there is, the bug does not move. (Remember that at any time step in the simulation, our bugs are told to move in the order in which they were created. Therefore, one bug might be frustrated in its move by another bug that would have moved out of the first bug's way had second bug moved first.)

Finally, we've added a capability to our SimpleBug class, the ability to respond to a `reportIfEaten()` message, a query as to whether or not the bug has eaten. The boolean variable `haveEaten` is defined as an instance variable for the SimpleBug class. Each time the bug takes a `randomWalk()`, it sets `haveEaten` to true if it finds food and false otherwise. (Previously the bug reported its find to the console without prompting.) The new `reportIfEaten()` method sends a report to the console if `haveEaten` is true, and does nothing if `haveEaten` is false.

`/SimpleSwarmBug2/SimpleBug.java`

```

// SimpleBug.java
// Defines the class for our SimpleBug agents/

```

```

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.space.Grid2dImpl;

public class SimpleBug extends SwarmObjectImpl
{
    // These instance variables keep track of a given bug's foodspace,
    // position and identity. We also save the dimensions of the
    // foodspace so that we can make fewer calls to the getSizeX() and
    // getSizeY() methods in the bug's randomWalk().
    FoodSpace myFoodSpace;
    Grid2dImpl myBugSpace;
    int xPos;
    int yPos;
    int bugNumber;

    int worldXSize;
    int worldYSize;

    // haveEaten keeps track of whether the bug has eaten on its most
    // recent walk.
    boolean haveEaten;

    // Constructor to create a SimpleBug object in Zone aZone and to
    // place it in the foodspace and bugspace, fSpace and bSpace, at
    // the specified X,Y location. The bug is also given a numeric id,
    // bNum.
    public SimpleBug(Zone aZone, FoodSpace fSpace, Grid2dImpl bSpace,
                    int X, int Y, int bNum)
    {
        // Call the constructor for the bug's parent class.
        super(aZone);

        // Record the bug's foodspace, bugspace, initial position and
        // id number.
        myFoodSpace = fSpace;
        myBugSpace = bSpace;
        worldXSize = myFoodSpace.getSizeX();
        worldYSize = myFoodSpace.getSizeY();
        xPos = X;
        yPos = Y;
        bugNumber = bNum;

        // Announce the bug's presence to the console.
        System.out.println("SimpleBug number " + bugNumber +
                           " has been created at " + xPos + ", " + yPos);
    }

    // This is the method to have the bug take a random walk backward
    // (-1), forward (+1), or not at all (0) in first the X and then
    // the Y direction. The randomWalk method uses
    // getIntegerWithMin$withMax() to return an integer between a
    // minimum and maximum value, here between -1 and +1.
    // Globals.env.uniformRand is an instance of the class
    // UniformIntegerDistImpl, instantiated by the call to
    // Globals.env.initSwarm in StartSimpleBug. Note that the bug's

```

```

// world is a torus.  If the bug walks off the edge of its
// rectangular world, it is magically transported (via the modulus
// operator) to the opposite edge.  If on its walk the bug finds
// food, it eats it and turns on the haveEaten flag so it can
// report its feast if asked.  Note that before the bug actually
// moves, we must check to see that there is no other bug at the
// destination cell.  If there is, the this bug just stays put.
public void randomWalk()
{
    int newX, newY;

    // Turn off the haveEaten flag.
    haveEaten = false;

    // Decide where to move.
    newX = xPos +
        Globals.env.uniformIntRand.getIntegerWithMin$withMax(-1, 1);
    newY = yPos +
        Globals.env.uniformIntRand.getIntegerWithMin$withMax(-1, 1);
    newX = (newX + worldXSize) % worldXSize;
    newY = (newY + worldYSize) % worldYSize;

    // Is there a bug at the new position already?  If not, put a
    // null at this bug's current position and put this bug at the
    // new position.
    if (myBugSpace.getObjectAtX$Y(newX, newY) == null)
    {
        myBugSpace.putObject$atX$Y(null, xPos, yPos);
        xPos = newX;
        yPos = newY;
        myBugSpace.putObject$atX$Y(this, xPos, yPos);
    }

    // If there is food at this cell, eat it and set the haveEaten
    // flag.
    if (myFoodSpace.getValueAtX$Y(xPos, yPos) == 1)
    {
        myFoodSpace.putValue$atX$Y(0, xPos, yPos);
        haveEaten = true;
    }
}

// Method to report the bug's position to the console.
public void reportPosition()
{
    System.out.println("Bug " + bugNumber + " is at " + xPos +
        ", " + yPos);
}

// Method to report if the bug has eaten.
public boolean reportIfEaten()
{
    if ( haveEaten )
        System.out.println("Bug " + bugNumber + " has found food at " +
            xPos + ", " + yPos);

    return haveEaten;
}

```

```
}
```

There are no changes to `FoodSpace.java`, `StartSimpleBug.java` or the `Makefile`. The changes we have made to the model have no impact on the way in which the application is constructed in `StartSimpleBug.java`

When the application is built and run, we see the creation messages for a lot of bugs. (You may want to comment out the creation message in the `SimpleBug` constructor.) We then see messages from the reporter bug when it finds food, fewer and fewer messages as time goes on and the foodspace is emptied. We have reduced the value of `endTime` dramatically. With a `bugDensity` of 0.1 in an 80 x 80 world, there are now about 640 bugs in our model rather than one, and each simulated time step takes much longer.

/SimpleSwarmBug3

READING PARAMETERS FROM A FILE

Up to now, we have "hard coded" the values of our model parameters: worldXSize, worldYSize, seedProb, bugDensity and endTime. We could, of course, use Java's file and/or console I/O routines to read those values at run time, but Swarm provides its own I/O facilities, one set for file I/O and the other for console I/O through a graphical user interface. We'll look at the former here and the latter in the next application.

The initialization of the model parameters occurs in the creation of our ModelSwarm. The created modelSwarm object then communicates the parameters to whatever other objects need to know them. You might think, then, that in order to read the parameters from a file we would need to make changes to the ModelSwarm creator in ModelSwarm.java. In fact, we don't change the ModelSwarm creator, but rather the way in which it is called in **StartSimpleBug.java**.

The *lispArchiver* object in the Swarm *Globals* environment contains a method called *getWithZone\$key()*. This method can be used to call the constructor for an class and to read and fill in one or more of the instance variables for that class in creating a class instance. Here, instead of calling the ModelSwarm constructor directly, we allow *lispArchiver* to call the constructor and, at the same time, to read the model parameters from a file and to insert them into the newly created modelSwarm. *getWithZone\$key()* takes two arguments, the *Zone* in which we want our new object to be created and a "key." *getWithZone\$key()* then looks for that key in a file and creates an object according to the instructions associated with that key. By default, the name of the file is the name of our application (the name we passed to *initSwarm()*) with the file type ".scm". **SimpleBug.scm** is shown below.

The format of the .scm file is a bit fussy. The key passed to *getWithZone\$key()* comes after the "cons" keyword. This is followed by the name of the class whose constructor is to be called, ModelSwarm, and by the names and values of the instance variables we want to set upon creation. (Any instance variable not set in the .scm file retains the value set in its ModelSwarm declaration.) Now, when we want to change the value of a model parameter, we need only change the value in SimpleBug.scm. We do not need to rebuild our application. (More information on the *ListArchiver* interface can be found in the Users' Guide.)

/SimpleSwarmBug3/StartSimpleBug.java

```
// StartSimpleBug.java
// Java SimpleBug application.

import swarm.Globals;
import swarm.defobj.Zone;

public class StartSimpleBug
{
```

```

public static void main (String[] args)
{
    ModelSwarm modelSwarm;

    // Swarm initialization: all Swarm apps must call this first.
    Globals.env.initSwarm ("SimpleBug", "2.1",
        "bug-swarm@santafe.edu", args);

    // Create a top-level Swarm object and build its internal
    // objects and activities. Note that we now use the Lisp
    // Archiver to create modelSwarm and to load the model's
    // parameters from a file. The default filename is the
    // name given by the first argument to initSwarm(), above,
    // with the file extension .scm. In this case that is
    // SimpleBug.scm.

    // The second argument to getWithZone$key(), "modelSwarm", is
    // the key in the .scm file which contains the class of the
    // object to be created, ModelSwarm, and the values of the
    // model parameters in the ModelSwarm class that we wish to
    // set when the new object is created: worldXSize, worldYSize,
    // seedProb, bugDensity and endTime.
    modelSwarm =
        (ModelSwarm)Globals.env.lispAppArchiver.getWithZone$key(
            Globals.env.globalZone, "modelSwarm");
    modelSwarm.buildObjects();
    modelSwarm.buildActions();
    modelSwarm.activateIn(null);

    // Now activate the swarm.
    (modelSwarm.getActivity()).run();
}
}

```

/SimpleSwarmBug3/SimpleBug.scm

```

(list
  (
    cons 'displaySwarm
      (
        make-instance 'ObserverSwarm
          #:displayFrequency 1
          #:zoomFactor 4
        )
      )
  (
    cons 'modelSwarm
      (
        make-instance 'ModelSwarm
          #:worldXSize 80
          #:worldYSize 80
          #:seedProb 0.90
          #:bugDensity 0.01
        )
      )
  )
)

```

```
        #:bugHardiness 20
      )
    )
```

When we build and run this version of the application we see the same type of output as before, but with fewer bugs and more food. (The SimpleBug.scm value of seedProb is higher and bugDensity lower than the values hard coded in ModelSwarm.java.) We also increased the value of endTime in SimpleBug.scm. With fewer bugs we can let the simulation run a bit longer. You might want to run the application several times with different parameter values to convince yourself that the parameters are indeed being read from SimpleBug.scm.

/SimpleObserverBug

GOING GUI

Now that we have an operational model, it is time to work in its presentation. Swarm provides a host of tools to implement a graphical user interface for our model, allowing us to visualize the bugs in their world and to interact with the model through a control panel. Both of these capabilities are provided by Swarm's *GUISwarm* interface. Essentially we create a top-level *GUISwarm* object that will encapsulate, control and display our *ModelSwarm*. Since our *ModelSwarm* is an object that stands largely on its own, most of the work we need to do lies in creating the *GUISwarm* itself.

Our particular GUI object will belong to a subclass of *GUISwarm*, the *ObserverSwarm* class, defined in **ObserverSwarm.java**. Like any other *Swarm* object, an *ObserverSwarm* object is built in four stages with a constructor, *buildObjects()*, *buildActions()* and *activateIn()*. The constructor is trivial. *ObserverSwarm* extends the *GUISwarm* implementation class, *GUISwarmImpl*, and it uses the constructor of its parent. Note that there are two display parameters, *displayFrequency* and *zoomFactor*. We'll see their use shortly.

The first thing we need to do in an *ObserverSwarm*'s *buildObjects()* method is to initialize the *ObserverSwarm* through its parent's *buildObjects()* method. Aside from the necessary initializations, the *GUISwarm* *buildObjects()* method will also display a control panel on the user's screen. A *GUISwarm* object has its own built-in GUI control panel with five buttons: START, STOP, NEXT, SAVE, and QUIT. These buttons control the progress of our simulation and are normally pressed by the user with the mouse. However, the buttons can also be "pressed" internally, as we shall see below.

Since the *ObserverSwarm* object is our application's top-level *Swarm*, it must take over the construction of the simulation model itself. *buildObjects()* creates the *ModelSwarm* object, *modelSwarm*, using the now-familiar *LispArchiver*. Note that we create *modelSwarm* in a sub-*Zone* of the *Zone* in which the *ObserverSwarm* itself is located. (*getZone()* returns the *Zone* of the current object, and calling on the implementation class constructor for a *Zone* creates a sub-*Zone* of its argument.) We now have a model object that the *ObserverSwarm* can monitor and control.

Moving right along, *getControlPanel().setStateStopped()* returns the control panel associated with this *ObserverSwarm* and "presses" its STOP button. (Both these methods are inherited from *GUISwarm*.) Execution of the application will wait here until the user presses START, signaling that she is ready to begin. (As we'll see in the next application, this can provide the user an opportunity to set some of the model's parameters.)

Once the user gives the go-ahead, the *ObserverSwarm* calls upon *modelSwarm* to build its objects with *modelSwarm.buildObjects()*. *ObserverSwarm* then gets down to the business of building its own display objects.

The first of these is a *Colormap* object, an object that contains the user-defined correspondence between the names of colors and a numeric code. The *Colormap* object is created with *Colormap*'s implementation class and the codes are set with the *setColor\$ToName()* method. Note that the numeric codes are byte integers and the color names are strings. As we will see below, the correspondences we have chosen are intentional. (Our foodspace will be displayed in black and red, and our bugs in green. The yellow is "reserved for future use.")

Indeed, the next step is to give our bugs their display color by sending each bug in *bugList* a 2 (cast as a byte integer) for green as its color code. We first ask *modelSwarm* to give us its *bugList* with the *modelSwarm.getBugList()* message. We then iterate through the bugs in the list, sending a *setBugColor()* message to each. (Note that we use the *getCount()* method inherited from the *List* class to retrieve the number of bugs in *bugList*.)

The second display object to be created is a "raster widget." A raster is a display window onto which we will project our foodspace and our bugspace. The raster knows all about the graphics environment of our computer and it will do all the hard graphics work for us. We create a particular kind of raster here, a *ZoomRaster*. A *ZoomRaster* object can be sized to pretty much any physical size on our computer screen by setting its "zoomFactor" or by resizing it with the mouse. After calling upon the constructor of the *ZoomRaster* implementation class, we give *worldRaster* our *Colormap* object, our desired zoomFactor (an integer between 4 and 8 works well), the size of our model world, and a window title. (Note that we ask *modelSwarm* to return its world to us with the *getWorld()* message and then use *getSizeX()* and *getSizeY()* methods to find the dimensions of that world. The *ModelSwarm* class contains several new methods like this as we shall see below.)

We have defined *zoomFactor* in a group of variables we've called display parameters. The value of *zoomFactor* defaults to 8, but we'll allow it to be changed in the *SimpleBug.scm* file and we'll see in the next application how it can be changed interactively as well.

worldRaster also needs to know what to do if for any reason it suffers an untimely death. Toward the end of the *ObserverSwarm* class we have a method called *_worldRasterDeath_()* which simply calls on *worldRaster* to "drop" itself. As we have seen, in order to pass this message to *worldRaster* we need to encapsulate the message in a *Selector* object. Selectors, in turn, need to be created within *try/catch* blocks. As the need for *Selector* objects grows in our applications, it is useful to routinize their creation. We do this by creating a new class of Swarm utilities, *SwarmUtils*. The first (and for now only) method in *SwarmUtils* is the static method *getSelector()*. The operation of this method is detailed in *SwarmUtils.java*. For now, suffice it to say that *getSelector()* takes two arguments and returns a *Selector* object based upon them. The first argument is either a class id or the name of a class as a String. (Java allows this kind of method overloading.) The second argument is the name of a method from that class as a String, that is, the message itself. We pack up the *_worldRasterDeath_()* method in a selector

and give it to worldRaster using *enableDestroyNotification\$notificationMethod()*, specifying too that the message is to be sent to the ObserverSwarm object itself.

Finally we call on worldRaster to "pack" up all the information we have given it and to ready itself for display on the screen by sending worldRaster the *pack()* message.

We want to superimpose our foodspace on the raster and we do that by creating a third display object, a *Value2dDisplay* object. A *Value2dDisplay* object takes a two-dimensional integer lattice such as our foodspace (and its *Grid2d* parent) and sends it to the raster for display. The integers in each cell of the lattice are translated to a color using the codes defined in the *Colormap* object. Given our foodspace and colormap, cells with no food (the cells with 0's) will display as black while cells with food (the cells with 1's) will display as red. As usual, we create the *Value2dDisplay* object using the implementation class and we pass to its constructor the worldRaster, the colormap, and the foodspace returned to us by the *getFood()* message to modelSwarm.

Our fourth and final display object will be used to display the positions of our bugs in their bugworld. This is an *Object2dDisplay* object which takes a two dimensional lattice of objects, a *Discrete2d* object such as modelSwarm's bugSpace, and asks each object on the lattice to draw itself on the raster. Our *Object2dDisplay* object then needs for its constructor the raster object (worldRaster), the *Discrete2d* object (bugSpace), and the message that prompts each object on the lattice to draw itself. Our SimpleBugs will soon respond to the *drawSelfOn()* message (we'll add this method to the SimpleBug class below) and we pack up the message in a *Selector* to pass on to the *Object2dDisplay*'s implementation class constructor. (Note that we ask modelSwarm to return the bugSpace to us by sending it the *getWorld()* message.)

When asked to display its objects, an *Object2dDisplay* object loops through the lattice it was given and sends to each object it finds the message it was given. If the lattice is sparsely populated, however, it can take a long time for the *Object2dDisplay* object to find the relatively few bugs on the lattice. A more efficient method, therefore, is to actually pass the *Object2dDisplay* object a list of the objects on the lattice and have the *Object2dDisplay* object use the list to send its message. We choose that option here and, immediately after creating bugDisplay we give it our modelSwarm's bugList, using the *setObjectCollection()* method. We ask modelSwarm to give us its bugList by sending it the *getBugList()* message.

By the end of the ObserverSwarm's *buildObjects()* method we have created both the objects needed to run our model and the objects needed to display it.

The ObserverSwarm's *buildActions()* phase begins by initializing itself with the parent's *buildActions()*. It then calls upon modelSwarm to build the actions for the simulation model itself. The next step is to create an *ActionGroup* to hold the actions needed to control modelSwarm and to generate the display of the model.

The guts of `buildActions()` are the display actions. They are pretty straightforward. We create and insert into `displayActions` two "*display*" messages, one to `foodDisplay` and another to `bugDisplay`, telling each one to check the current state of its lattice and to send its display information to `worldRaster`. We then insert a message to `worldRaster` to draw itself to a screen buffer. Finally we insert a "*doTkEvents*" message to the *ActionCache* object. (The *ActionCache* object is retrieved from the *Globals* environment using the `getActionCache()` method.) This last message is needed to "flush" `worldRaster`'s buffer to the screen. It should always be the last action in the list of display actions. Without it, the screen is not updated nor is the control panel checked for a button press.

We have finished building the display actions, but there is one more message to be inserted into `displayActions`. The "*checkForDone*" message instructs the *ObserverSwarm* object itself to see if `modelSwarm` has finished with its work. We will explain the `checkForDone()` method below.

Having packed up the desired actions into `displayActions`, we insert `displayActions` in a newly created schedule, `displaySchedule`. Note that we have defined the schedule interval, `displayFrequency`, in a group of variables we've called display parameters. The value of `displayFrequency` defaults to one, but we'll allow it to be changed in the `SimpleBug.scm` file and we'll see in the next application how it can be changed interactively as well.

The final stage in building our *ObserverSwarm* object is in the `activateIn()` method. Here the *ObserverSwarm* object takes the context passed to it and sends it on to its parent's `activateIn()` method. Since the *ObserverSwarm* object is going to be the top-level *Swarm*, it will be passed a null as its context. Note that the next step, however is to send an `activateIn()` message to `modelSwarm`. Since `modelSwarm` is a *subSwarm* of the *ObserverSwarm*, the latter tells the former to activate itself in the latter's context. The same is true of `displaySchedule`. It too is activated in the context of the *ObserverSwarm* object. Having finished with the activations, we return the finished *Activity* object which is now ready to run.

In our previous applications, we set an `endTime` to tell `modelSwarm` when to stop. In some sense that is no longer necessary. The control panel provided by the *GUISwarm* parent of *ObserverSwarm* has a STOP button that the user can press at any time to stop the simulation. Still, we will want in a later application to allow the simulation to stop itself and so we have retained `endTime` to provide a maximum runtime for our model. (If the user falls asleep, the model will eventually terminate without intervention.)

The decision about when to stop is made by the model itself, that is by the `modelSwarm` activity. As we shall see when we look at `ModelSwarm.java`, `modelSwarm` tests the simulation time against `endTime` and simply terminates itself if `endTime` has been reached. You will remember that a "*checkForDone*" message was inserted as the last action in *ObserverSwarm*'s *ActionGroup*. Let's assume that when `checkForDone()` is entered, `simulationFinished` still has its initialized value of false. We would then go on to the "else if" block which obtains the status of `modelSwarm` by getting its activity with

getActivity() and then getting the status of that activity with *getStatus()*. If *modelSwarm* has terminated itself, the status of its activity should be "Completed". We can check for that by matching the *Symbol* returned by *getStatus()* with the appropriate *Symbol* in *Swarm's Globals* environment. If the *Symbols* match, *simulationFinished* is set to true, a message is sent to the console, and *checkForDone()* presses the STOP button on the control panel. Everything stops, allowing the user to look at the raster window (and whatever other windows we've displayed on the screen), and to push the QUIT button when she's ready. What if the user presses the START or NEXT button, however? Execution of *Swarm* will continue when we might not want it to.

Normally, once the *Swarm* engine has processed *observerSwarm's* last message ("checkForDone") the simulation time is moved forward one period and the *modelSwarm* activity is called upon to do its thing for the next period. In this case, however, the *modelSwarm* activity has terminated itself and, if the user presses START, the *Swarm* engine passes over it, going straight on to the *ObserverSwarm* activity. The display messages are sent, updating the display with the same information as before since *modelSwarm's* activity was not executed, and the *checkForDone()* method is entered again. This time, *simulationFinished* is true, a smart-aleck message is sent to the console, *modelSwarm* is terminated (again) and dropped, and the QUIT button is pressed.

The last method in *ObserverSwarm.java* was given to *worldRaster* when it was created. It simply gives instructions about what to do when *worldRaster* dies. The *worldRaster* object is dropped and its pointer set to null.

/SimpleObserverBug/ObserverSwarm.java

```
// ObserverSwarm.java The observer swarm is collection of objects that
// are used to run and observe the ModelSwarm that actually comprises
// the simulation.

import swarm.Globals;
import swarm.Selector;

import swarm.defobj.Zone;
import swarm.defobj.ZoneImpl;
import swarm.defobj.Symbol;

import swarm.gui.Colormap;
import swarm.gui.ColormapImpl;
import swarm.gui.ZoomRaster;
import swarm.gui.ZoomRasterImpl;

import swarm.space.Value2dDisplay;
import swarm.space.Value2dDisplayImpl;
import swarm.space.Object2dDisplay;
import swarm.space.Object2dDisplayImpl;

import swarm.simtoolsgui.GUISwarm;
import swarm.simtoolsgui.GUISwarmImpl;
```

```

import swarm.activity.ActionGroup;
import swarm.activity.ActionGroupImpl;
import swarm.activity.Schedule;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;

import swarm.collections.ListImpl;

// ObserverSwarm is a subclass of GUISwarm implementation class.
public class ObserverSwarm extends GUISwarmImpl
{
    // Declare the display parameters and their default values.
    public int displayFrequency = 1;
    public int zoomFactor = 8;

    // A flag to signal the end of the simulation.
    public boolean simulationFinished = false;

    // Declare other variables local to ObserverSwarm.
    ModelSwarm modelSwarm;
    ZoomRaster worldRaster;
    Value2dDisplay foodDisplay;
    Object2dDisplay bugDisplay;
    ScheduleImpl displaySchedule;

    // This is the constructor for a new ObserverSwarm.
    public ObserverSwarm(Zone azone)
    {
        // Use the parent class to create an observer swarm.
        super(azone);
    }

    // The buildObjects method.
    public Object buildObjects()
    {
        Zone modelZone;
        Colormap colormap;
        Selector sel;

        // Use the parent class to initialize the process.
        super.buildObjects();

        // First we create the model that we're actually observing, by
        // creating an instance of the ModelSwarm class, modelSwarm.
        // modelSwarm will now be a subSwarm of this top-level
        // ObserverSwarm rather than the top-level Swarm in its own
        // right. We create modelSwarm in its own newly-created Zone
        // so that modelSwarm's storage is segregated from the rest of
        // the application. Note that as in SimpleSwarmBug3, we are
        // reading the modelSwarm parameters from a file and so use
        // the List Archiver to create modelSwarm.

        modelZone = new ZoneImpl(getZone());
        modelSwarm =

```

```

        (ModelSwarm)Globals.env.lispAppArchiver.getWithZone$key(
            modelZone, "modelSwarm");

// Instruct the control panel to wait for a button event: we
// halt here until someone hits a control panel button.
// Eventually this will allow the user a chance to fill in
// parameters before the simulation runs.
getControlPanel().setStateStopped();

// OK - the user has pressed a button. Now we're ready to
// start.

// Allow the model swarm to build its objects.
modelSwarm.buildObjects();

// Now build the GUI display objects.

// First, create a colormap, the correspondence between a
// color and a byte integer code. This is a global resource
// which is used by lots of different objects. Then set the
// three colors we will be using. Since the FoodSpace grid
// uses one to indicate a cell with food and zero to indicate
// a cell without food, FoodSpace cells will be displayed in
// red or black depending on whether they contain food or not.
// We'll use green to indicate the location of our bugs.
// (Yellow is reserved for future use.)
colormap = new ColormapImpl(getZone());
colormap.setColor$ToName((byte)0, "black");
colormap.setColor$ToName((byte)1, "red");
colormap.setColor$ToName((byte)2, "green");
colormap.setColor$ToName((byte)3, "yellow");

// Now tell each of the bugs in the model to set its default
// display color to green (2). We do this by getting the list
// of bugs created in modelSwarm and iterating through it.
ListImpl bugList = modelSwarm.getBugList();
for (int i = 0; i < bugList.getCount(); i++)
{
    SimpleBug bug = (SimpleBug)bugList.atOffset(i);
    bug.setBugColor((byte)2);
}

// Next, create a "raster widget", a 2-dimensional display
// window. We tell the raster what to do if it dies, give it
// its colormap, set its zoom factor (its actual size on the
// display screen), set its virtual dimensions to the size of
// our world, and give it its title.
worldRaster = new ZoomRasterImpl (getZone(), "worldRaster");
sel = SwarmUtils.getSelector(this, "_worldRasterDeath_");
worldRaster.enableDestroyNotification$notificationMethod
    (this, sel);
worldRaster.setColormap (colormap);
worldRaster.setZoomFactor (zoomFactor);
worldRaster.setWidth$Height ((modelSwarm.getWorld()).getSizeX(),
    (modelSwarm.getWorld()).getSizeY());
worldRaster.setWindowTitle ("Food World");

```

```
// This instructs the raster to digest all the information we
// have just given it and to initialize itself for display.
worldRaster.pack();

// Now create a Value2dDisplay, an object that will display an
// arbitrary 2-dimensional value array, in this case our
// foodspace, on the raster widget. Think of the foodspace
// lattice overlaying the raster. Remember that we have set
// the colormap such that cells with no food (0's) will be
// displayed in black and cells with food (1's) will be
// displayed in red. We use the Value2dDisplay implementation
// class and give it a zone, the raster, the colormap and the
// foodspace.
foodDisplay = new Value2dDisplayImpl(getZone(), worldRaster,
                                     colormap,
                                     modelSwarm.getFood());

// Also create an Object2dDisplay that will display the bugs
// on the raster, giving it the raster, the grid on which the
// objects (bugs) are located, and the draw message to the
// bugs. (The Object2dDisplay relies on the objects to send
// their own draw messages to the raster.) Once the
// Object2dDisplay is created, we give it the list of bugs to
// which the draw message needs to be sent. Again, think of
// the bugspace as overlaying the raster.)
sel = SwarmUtils.getSelector("SimpleBug", "drawSelfOn");
bugDisplay = new Object2dDisplayImpl(getZone(), worldRaster,
                                     modelSwarm.getWorld(), sel);
bugDisplay.setObjectCollection(modelSwarm.getBugList());

    return this;
}

public Object buildActions()
{
    Selector sel;
    ActionGroupImpl displayActions;

    // Use the parent class to begin the process.
    super.buildActions();

    // Call on the model swarm to build and schedule its actions.
    modelSwarm.buildActions();

    // Create an ActionGroup for display. This is a list of
    // display actions that we want to occur at each step in
    // simulation time. First we tell the foodDisplay and the
    // bugDisplay to display themselves on the raster widget, and
    // then tell the raster widget to display itself on the
    // screen. "doTkEvents", is required at the end to make
    // everything happen. We then check to see if modelSwarm has
    // told us to stop the simulation.
    displayActions = new ActionGroupImpl(getZone());
```

```

sel = SwarmUtils.getSelector(foodDisplay, "display");
displayActions.createActionTo$message(foodDisplay, sel);

sel = SwarmUtils.getSelector(bugDisplay, "display");
displayActions.createActionTo$message(bugDisplay, sel);

sel = SwarmUtils.getSelector(worldRaster, "drawSelf");
displayActions.createActionTo$message(worldRaster, sel);

sel = SwarmUtils.getSelector(getActionCache(), "doTkEvents");
displayActions.createActionTo$message(getActionCache(), sel);

sel = SwarmUtils.getSelector(this, "checkForDone");
displayActions.createActionTo$message(this, sel);

// Finally, put the ActionGroup into a display schedule.
displaySchedule = new ScheduleImpl(getZone(), displayFrequency);
displaySchedule.at$createAction(0, displayActions);

return this;
}

// Activate the schedules so that they are ready to run. The
// swarmContext argument is the zone in which the ObserverSwarm is
// activated. Typically the ObserverSwarm is the top-level swarm,
// so it is activated in "null". The other (sub)swarms and
// schedules will be activated inside of the ObserverSwarm
// context.
public Activity activateIn(Swarm swarmContext)
{
// Use the parent class to activate ourselves in the context
// passed to us.
super.activateIn(swarmContext);

// Now activate the model swarm in the ObserverSwarm context.
modelSwarm.activateIn(this);

// Then activate the ObserverSwarm schedule in the
// ObserverSwarm context.
displaySchedule.activateIn(this);

// Finally, return the activity we have built - the thing that
// is ready to run.
return getActivity();
}

// This method checks each period to see if the simulation is
// done. It is called at the end of each period.
public void checkForDone()
{
if (simulationFinished)
{
// The simulation is over. Presumably we got here because
// the user did not QUIT when told to do so after the
// modelSwarm activity was terminated. We therefore chide
// her and press QUIT ourselves.
System.out.println("I said to QUIT!");
}
}

```

```

        modelSwarm.getActivity().terminate();
        modelSwarm.drop();
        getControlPanel().setStateQuit();
    }

    else if (modelSwarm.getActivity().getStatus() ==
        Globals.env.Completed)
    {
        // modelSwarm has signaled us that the simulation is
        // finished by terminating itself. (ObserverSwarm sees
        // this as Completed.) Press the STOP button on the
        // control panel. Pressing STOP rather than QUIT leaves
        // the raster window and the control panel on the display
        // so that the user can look at the results of the
        // simulation. (Those windows will disappear when the
        // user presses QUIT.) We also set a flag to indicate
        // that the simulation is over, in case the user presses
        // START or NEXT instead of QUIT.
        simulationFinished = true;
        System.out.println("The simulation ended after "
            + Globals.env.getCurrentTime()
            + " periods.");
        System.out.println("Press QUIT when ready.");
        getControlPanel().setStateStopped();
    }
}

// This is a method given to the raster object to tell it what to
// do in the event of an untimely death.
public Object _worldRasterDeath_ (Object caller)
{
    worldRaster.drop ();
    worldRaster = null;
    return this;
}
}

```

We have introduced a lot of new material in creating the ObserverSwarm. Fortunately, very little more is needed to make it all work. Since we have not made any changes to our model itself, **ModelSwarm.java** needs very little change. You will remember that we used three new methods in the ObserverSwarm that returned bugList, bugSpace and foodSpace from modelSwarm. They are getBugList(), getWorld() and getFood(), respectively, and they have been added to the methods in the ModelSwarm class. We no longer need a reporter bug and so we have removed it from the code in buildObjects(). Instead, we simply report to the console the number of bugs created by using the *getCount()* method on bugList. *getCount()* returns the number of objects currently in a *List* object.

In buildActions() we have taken advantage of our new getSelector() utility, but have made no substantive changes. Nor have we made any changes in activateIn(). However, as noted above, we have made a change to checkTime(). Now when endTime is reached,

checkTime gets the modelSwarm activity and terminates it, signaling to the ObserverSwarm that modelSwarm has had enough.

Because these changes are so minor, we reproduce below only the new methods.

/SimpleObserverBug/ModelSwarm.java (new elements only)

```

// The next three methods return some useful information about the
// created ModelSwarm to the caller. They will be used by the
// Observer Swarm.
public ListImpl getBugList()
{
    return bugList;
}

public Grid2dImpl getWorld()
{
    return bugSpace;
}

public FoodSpace getFood()
{
    return foodSpace;
}

// This is a pretty crude method to end the simulation after an
// arbitrary number of periods given by the endTime parameter. If
// the simulation time returned by getCurrentTime() is greater
// than endTime, we terminate the modelSwarm activity. The
// ObserverSwarm will pick this up and handle it. Note that this
// is no longer strictly necessary since the user can now stop the
// simulation at any time using the control panel. Still, this is
// a placeholder for a more sophisticated end-of-simulation
// routine to be introduced later.
public void checkTime()
{
    if (Globals.env.getCurrentTime() >= endTime)
        getActivity().terminate();

    return;
}
}

```

The changes to the SimpleBug class are likewise few. We no longer ask a bug to report its birth to the console and we add two methods. The first, setBugColor(), is to allow the bug to set its bugColor with a message from the ObserverSwarm. The second, drawSelfOn(), instructs the bug to draw itself on the raster widget (which is passed to it by the *Object2dDisplay* object), at the bug's current position in bugColor. drawSelfOn() uses the *ZoomRaster* class method *drawPointX\$Y\$Color()* to accomplish this task.

Again, given the paucity of changes to SimpleBug.java, we reproduce here only the new methods.

/SimpleObserverBug/ModelSwarm.java (new elements only)

```

// These are methods that allow the bug to draw itself on the GIU
// raster display object.  The first tells the bug what color it
// should use, the second draws the bug at its current location.
public Object setBugColor(byte c)
{
    bugColor = c;
    return this;
}

public Object drawSelfOn (Raster r)
{
    r.drawPointX$Y$Color (xPos, yPos, bugColor);
    return this;
}

```

There are no changes to `FoodSpace.java` and only the addition of `SwarmUtils.java` to the `Makefile`. The code for the static `getSelector()` method in the new `SwarmUtils` class is documented in `SwarmUtils.java`.

That leave us only with **StartSimpleBug.java**. The major change here is that `StartSimpleBug` builds and activates an `ObserverSwarm` as the top-level *Swarm*, rather than a `ModelSwarm`. It uses the same procedure we are familiar with: creating `displaySwarm` with the *LispArchiver*, building its objects and actions, and then activating and running it. Note that we have introduced a second key, "displaySwarm", that will distinguish the parameters for the `ObserverSwarm` from those for the `ModelSwarm` in the `SimpleBug.scm` file.

/SimpleObserverBug/StartSimpleBug.java

```

// StartSimpleBug.java
// Java SimpleBug application.

import swarm.Globals;
import swarm.defobj.Zone;

public class StartSimpleBug
{
    public static void main (String[] args)
    {
        ObserverSwarm displaySwarm;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("SimpleBug", "2.1",
                               "bug-swarm@santafe.edu", args);

        // Create a top-level Swarm object, now DisplaySwarm, and
        // build its internal objects and activities. Note that we use
        // the Lisp Archiver to create displaySwarm and to load the
        // model's display parameters from the a file. The default

```

```

// filename is the name given by the first argument to
// initSwarm, above, with the file extension .scm. In this
// case that is SimpleBug.scm.

// "displaySwarm", the second argument to getWithZone$key(), is
// the key in the .scm file which contains the values of the
// display parameters in the ObserverSwarm class. For now
// there is only one parameter, displayFrequency.

displaySwarm =
  (ObserverSwarm)Globals.env.lispAppArchiver.getWithZone$key(
    Globals.env.globalZone, "displaySwarm");

displaySwarm.buildObjects();
displaySwarm.buildActions();
displaySwarm.activateIn(null);

// Now start the displaySwarm and the control panel it
// provides.
displaySwarm.go();

// The user has pressed Quit. Drop everything and return.
displaySwarm.drop();
}
}

```

/SimpleObserverBug/SimpleBug.scm

```

(list
  (
    cons 'displaySwarm
      (
        make-instance 'ObserverSwarm
          #:displayFrequency 1
          #:zoomFactor 4
        )
      )
  (
    cons 'modelSwarm
      (
        make-instance 'ModelSwarm
          #:worldXSize 80
          #:worldYSize 80
          #:seedProb 0.90
          #:bugDensity 0.01
          #:endTime 625
        )
      )
  )
)

```

What then happens when we Make and run our new application? We first see a control panel appear on the screen with buttons we can click on. Clicking on START results in

the appearance of the display raster with our "world" displayed on it. Cells without food are in black, cells with food are in red and, as the green bugs move around in the world on their random walks, the food begins to disappear as it is eaten. We can stop the action at any time by pressing the STOP button, and then restart it using START. Pressing NEXT takes the model through a single time step and stops it again. We can press QUIT at any time, and are more-or-less required by our checkForDone() method to press QUIT when endTime is reached.

/SimpleObserverBug2

ADDING PROBES AND PROBE DISPLAYS

Now we add a very useful observation object. Probes are just what their name implies - probe-like access to the internals of any object in Swarm. The *Probe Library* allows one to effectively see inside objects: to observe the values of an object's internal variables, to change those values on the fly, and even to invoke the object's methods. This is both incredibly useful and incredibly dangerous. However, in spite of their dangers, probes are essential. We can never anticipate in advance everything that we might want to know about an object's state and it would be both tedious and inefficient to provide a "get" method for every internal variable on every object. Hence, probes.

Probing is based upon *ProbeMap* objects. There is a *ProbeMap* object for every class in the system, although the object doesn't really exist until someone asks for it. *ProbeMap* objects establish the mapping from an object's class to the location of the variables in the internal data structure of a particular instance of that class.

Creating a *ProbeMap* object turns out to be relatively easy and very few changes to our application are required. Indeed, the only changes will come in the *ObserverSwarm* and *ModelSwarm* classes. Let's take the latter first.

We begin by adding two imports, *EmptyProbeMap* and its implementation class, *EmptyProbeMapImpl*. All the substantive changes occur in the constructor for a *ModelSwarm* object.

After our usual call upon the parent-class constructor, we create a new *EmptyProbeMap* object and give it the class of the object we want to probe. Here, of course, that is the *ModelSwarm* class itself. Then we add to the new *probeMap* object, probes for those variables we want to track and/or change: *woldXSize*, *worldYSize*, *seedProb*, *bugDensity* and *endTime*. The *getProbeForVariable()* method returns a probe for the given variable and *addProbe()* inserts the probe into *probeMap*. (Note that if the variable we specify is not actually in the class with which *probeMap* is associated, we will get a runtime error. The compiler does not check this for us.) Once we have created our *ProbeMap* object, we insert it into the *ProbeLibrary* in Swarm's *Globals* environment. That's it! The *ProbeMap* and *ModelSwarm* are ready to go.

(We have chosen here to create an "empty" *ProbeMap* object. Had we wished to do so, we could have created instead a "complete" *ProbeMap* object (using *CompleteProbeMapImpl*) that would have contained by default all the variables and methods in the given class and its subclasses. For our purposes, that would be overkill. However, this class can be useful, particularly in debugging.)

Because so little of **ModelSwarm.java** has changed, we reproduce here only the constructor.

/SimpleObserverBug2/ModelSwarm.java (the new constructor only)

```
// This is the constructor for a new ModelSwarm.
public ModelSwarm(Zone azone)
{
    // Use the parent class to create a top-level swarm.
    super(azone);

    // Build a customized probe map. Without a custom probe map
    // the default is to show all variables and messages. Here we
    // choose to customize the appearance of the probe, giving a
    // nicer interface.

    // Create the probe map and give it the ModelSwarm class.
    probeMap = new EmptyProbeMapImpl(azone, getClass());

    // Now add probes for the variables we wish to probe.
    probeMap.addProbe(getProbeForVariable("worldXSize"));
    probeMap.addProbe(getProbeForVariable("worldYSize"));
    probeMap.addProbe(getProbeForVariable("seedProb"));
    probeMap.addProbe(getProbeForVariable("bugDensity"));
    probeMap.addProbe(getProbeForVariable("endTime"));

    // And finally install our probe map into the probeLibrary.
    // Note that this library was created by initSwarm().
    Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
}
```

We also want to probe the ObserverSwarm, in particular displayFrequency, zoomFactor, and a new variable, simulatedTime. The procedure is exactly the same and the new constructor for the ObserverSwarm class in **ObserverSwarm.java** is reproduced below.

/SimpleObserverBug2/ObserverSwarm.java (the new constructor only)

```
// This is the constructor for a new ObserverSwarm.
public ObserverSwarm(Zone azone)
{
    // Use the parent class to create an observer swarm.
    super(azone);

    // Build a custom probe map. Without a probe map, the default
    // is to show all variables and messages. Here we choose to
    // customize the appearance of the probe, giving a nicer
    // interface.

    // Create the probe map and give it the ObserverSwarm class.
    EmptyProbeMapImpl probeMap =
        new EmptyProbeMapImpl(azone, getClass());

    // Now add probes for the variables we wish to probe, using
    // the method in our SwarmUtils class.
    probeMap.addProbe(getProbeForVariable("displayFrequency"));
    probeMap.addProbe(getProbeForVariable("zoomFactor"));
    probeMap.addProbe(getProbeForVariable("simulatedTime"));
}
```

```
// And finally install our probe map into the probeLibrary.
// Note that this library object was automatically created by
// initSwarm.
Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
}
```

However, the ObserverSwarm needs to worry about two additional items. The first item is to make sure that the windows displaying the probes are updated at each time step. The ObserverSwarm does this by adding to its list of actions an "update" message to the *probeDisplayManager* in Swarm's *Globals* environment. Therefore in `buildActions()` we find just before the "doTkEvents" message

```
sel = SwarmUtils.getSelector(Globals.env.probeDisplayManager,
                             "update");
displayActions.createActionTo$message(Globals.env.probeDisplayManager,
                                       sel);
```

The second item is to make sure that our new `simulatedTime` variable is updated at each time step. This is done by adding yet another message to `buildActions()`, an "updateSimulatedTime" message to the ObserverSwarm object itself. The new code looks like

```
sel = SwarmUtils.getSelector(this, "updateSimulatedTime");
displayActions.createActionTo$message(this, sel);
```

and the `updateSimulatedTime()` method is trivial.

```
public void updateSimulatedTime()
{
    simulatedTime = Globals.env.getCurrentTime();
}
```

Happily, there are no changes to any of our other application files.

When this new version of the application is run, two probe map windows appear on the screen along with the control panel, one for the ModelSwarm object and the other for the ObserverSwarm object. (You may have to drag them around as they tend to overlap each other when first displayed.) Before pressing START on the control panel, you can view the values of the probed variables and change them if you wish. To change a variable, simply place the cursor in the variable's field, erase the current value, type in a new value, and press Enter on the keyboard. (That last step is required!) Note that it will take a few seconds for the new value to be "set". Change as many values as you like and then press the START button to begin the simulation. You will see `simulationTime` incremented as the simulation proceeds, showing that the probes are indeed being updated each period. Of course, in this application, it would make no sense to change any of the probed variables once the simulation has begun. All the variables except for `simulationTime` are used only when the model is initialized and `simulationTime` is reset to the current simulation time each period. However, in other applications it might indeed be useful to

change the value of a probed variable during the simulation, and we shall see an example of this in the next version of SimpleObserverBug.

/SimpleObserverBug3

A MATTER OF LIFE AND DEATH

Now that we have worked on the display aspects of our simulation, it is time to return our attention to the model itself. In particular, we want to allow our bugs to die. How then do we remove agents from the model in a graceful fashion? While we work on this task, we will also improve our ability to probe individual bugs and, in so doing, to extend or shorten their lives if we wish.

In the new version of our model, bugs die if they go hungry for a specified period of time. This functionality is added to the SimpleBug class in **SimpleBug.java**. We define two new variables, bugHardiness, which tells the bug how long it can survive without food and which is passed to the individual bug upon its creation, and periodsSinceEaten, which will keep track of how long the bug has gone hungry. We also pass to each new bug a pointer to the ModelSwarm that created it since bugs will have to send a message to their creator when they die. The remaining changes to the SimpleBug class are found in the randomWalk() method, which is reproduced below.

/SimpleObserverBug3/SimpleBug.java (the randomWalk() method only)

```
public void randomWalk()
{
    int newX, newY;

    // Decide where to move.
    newX = xPos +
        Globals.env.uniformIntRand.getIntegerWithMin$withMax(-1, 1);
    newY = yPos +
        Globals.env.uniformIntRand.getIntegerWithMin$withMax(-1, 1);
    newX = (newX + worldXSize) % worldXSize;
    newY = (newY + worldYSize) % worldYSize;

    // Is there a bug at the new position already? If not, put a
    // null at this bug's current position and put this bug at the
    // new position.
    if (myBugSpace.getObjectAtX$Y(newX, newY) == null)
    {
        myBugSpace.putObject$atX$Y(null, xPos, yPos);
        xPos = newX;
        yPos = newY;
        myBugSpace.putObject$atX$Y(this, xPos, yPos);
    }

    // If there is food at this cell, eat it and record the
    // fact. Otherwise, increment periodsSinceEaten and change the
    // bugColor to yellow (3).
    if (myFoodSpace.getValueAtX$Y(xPos, yPos) == 1)
    {
        myFoodSpace.putValue$atX$Y(0, xPos, yPos);
        haveEaten = true;
        periodsSinceEaten = 0;
    }
}
```

```

        setBugColor((byte)2);
    }
    else
    {
        haveEaten = false;
        ++periodsSinceEaten;
        if (periodsSinceEaten >= bugHardiness/2)
            setBugColor((byte)3);
    }

    // Now check to see if we're still alive!  If not, we tell
    // modelSwarm that we've died.
    if (periodsSinceEaten >= bugHardiness)
        modelSwarm.bugDeath(this);
}

```

If a bug encounters food on its walk, it "eats" it, sets `haveEaten` to true and `periodsSinceEaten` to zero. If, however, it does not find food, it sets `haveEaten` to false, increments `periodsSinceEaten` by one, and set its `bugColor` to yellow (code 3 in our `colorMap`) if it has reached half of its endurance. Finally, we check to see if the bug has reached the end of the line by going without food for `bugHardiness` periods. If it has, it sends a message to `modelSwarm` that it has died.

The `ModelSwarm` class must know what to do with this message and we have added a `bugDeath()` method to **ModelSwarm.java**. Before we get to that method, note that we have added `bugHardiness` to our list of model parameters. It can be set in the `SimpleBug.scm` file or through the probe display before the START button is pressed. We have also added a bug count variable, `numberOfBugs`, and deleted `endTime` since we no longer need it to stop our simulation. The `buildObjects()` method sports a new *List* object, `reaperQueue`, and it reflects the new constructor for a `SimpleBug`. We have also added a new action to `buildActions()`, a "reapBugs" message to the `ModelSwarm` itself. Let's then look at the `bugDeath()` and `reapBugs()` methods.

/SimpleObserverBug3/ModelSwarm.java (the new methods only)

```

public void bugDeath(SimpleBug abug)
{
    if ((SimpleBug)bugSpace.getObjectAtX$Y(abug.xPos, abug.yPos) == abug)
    {
        bugSpace.putObject$atX$Y(null, abug.xPos, abug.yPos);
        reaperQueue.addLast(abug);
    }
}

public void reapBugs()
{
    SimpleBug abug;

    while (reaperQueue.getCount() != 0)
    {
        abug = (SimpleBug)reaperQueue.removeFirst();
    }
}

```

```

        bugList.remove(abug);
        abug.drop();
        System.out.println("Bug " + abug.bugNumber +
            " has died of hunger.");
        numberOfBugs -= 1;
    }

    // Check the number of remaining bugs and quit the simulation
    // if there are none left, by terminating the ModelSwarm
    // activity.
    if (bugList.getCount() == 0)
        getActivity().terminate();
}

```

When a bug announces its death via the `bugDeath()` message, we check to make sure that the bug is where it thinks it is in our `bugSpace`. If it is, we remove the bug from `bugSpace` by putting a null at the bug's former location and then add the bug to the reaperQueue *List*. Then we return to continue with the simulation. Note that we do not at this point truly kill off the bug object by dropping it. There still may be other bugs that have yet to take their random walks and the Swarm engine may therefore still be traversing the `bugList`. It would be very bad form to change one of the `bugList` entries while the list was being traversed.

Only after the "randomWalk" message to all the bugs has been processed do we deal with the dead bugs by calling upon the `reapQueue()` method. We loop through the reaperQueue, popping each dead bug off the list, removing it from `bugList`, dropping it, reporting its death to the console, and decrementing the number of bugs. Once we have cleared the reaperQueue, we check `bugList` to make sure there are still some bugs left alive. If not, we *terminate()* the `ModelSwarm` activity. (We might have simply checked `numberOfBugs` rather than using `bugList.getCount()`, but the latter is safer.)

We have only a few more changes to make to our application, all in **ObserverSwarm.java**. The *ZoomRaster* object is capable of sending a message when it detects a mouse click on the raster display. In `ObserverSwarm`'s `buildObjects()` method we have added

```

sel = SwarmUtils.getSelector(bugDisplay, "makeProbeAtX$Y");
worldRaster.setButton$Client$Message(3, bugDisplay, sel);

```

Button 3 is the right button on the mouse. When the user right-clicks on the raster display, `worldRaster` sends the *makeProbeAtX\$Y* message to the *Object2dDisplay* object, `bugDisplay`, along with the X,Y coordinates of the point clicked upon. This instructs `bugDisplay` to create a complete probe map for the object at that location and to insert it in the *ProbeLibrary*.

It would be nice if the `ObserverSwarm` kept track of the simulation time and the number of bugs, and reported their current values in `ObserverSwarm`'s probe display. We have therefore introduced two new variables, `simulatedTime` and `numberOfBugs`, which are

kept current by the new `updateSimulatedTime()` method. A message to call upon this method each period has been added to the *ActionGroup* (in `buildActions()`) before the message to update the probe displays. Finally, we warn the user if she tries to set either of these variables before the simulation begins. It would be fruitless for her to do so.

When we build and run this new application, we see the control panel and probe displays. Change any of the display or model parameters (except `simulationTime` or `numberOfBugs`) and press START. Let the simulation run for a few periods and then press STOP. Let's look at all we can do with the probe displays.

In the upper left corner of a probe map window you will see the class name of the object being probed, highlighted in blue. If you right click on that name, you will create another kind of probe display. It looks similar to the previous one, except that there is a new button in the upper right corner with some connected green boxes. If you click on this with the left mouse-button, you will extend the probe display to show the super class of the object. It too will have the green-box button, which you can click on again, and so on, to see the object's inheritance hierarchy all the way up to "*Object*", the root of all Swarm objects.

For instance, a "*SimpleBug*" is a subclass of "*SwarmObject*," which is a subclass of "*CreateDrop*," which is a subclass of "*CreateDrop_s*," which is a subclass of "*Customize_s*," which is a subclass of "*Object_s*," which is a subclass of "*Object*." You can see what methods and variables an object has by virtue of inheritance from all of its super classes.

To get rid of any probe display, or any frame in a probe display, just click the red circle-and-X button in the upper right-hand corner of the display.

The probe displays will also allow you to alter the internal contents of an object. For instance you can alter the X and Y coordinates of a bug. When the simulation is stopped, click with the right mouse-button on a green or yellow bug on the raster window. You will remember that this is a signal for `bugDisplay` to create a complete probe map for a bug and to display it. Since this is a complete probe map, it will contain all of a *SimpleBug*'s public instance variables. Click with the left mouse-button in the field for the `xPos` in the probe display for a bug. The entry becomes highlighted, and you can type in some other (legal) value. Press the "Enter" key to enter the value. You can do the same for the `yPos` value. Be sure to hit the "Enter" key after you have changed an entry. When you restart the simulation, the bug will appear at the new coordinate values. (You can even make a bug "immortal" by setting its `bugHardiness` to a very large number!) The probe display for the bug will remain on the screen as long as the bug stays alive. You can use the display to track the bug's movements and its luck in finding food.

You should not attempt to change entries that do not change during the simulation, like a bug's pointer to its `foodSpace`, or its idea of the size of its world.

With the `probeDisplays`, you can also invoke methods on objects. Try this:

Click with the right mouse-button on the string "FoodSpace" in the bug's probe display. You will get a probe display on the foodSpace. This probe display has one entry: a method probe on the "seedFoodWithProb" method of FoodSpace. If you enter 1.0 into the argument slot to the right of the button and then click the button, you will have reseeded the foodSpace with food at every site when you start the simulation.

You can try pulling up probe displays on any object in the simulation this way, altering its variables and invoking its methods.

Thus, the probe mechanism is a very powerful tool for observing the detailed state of a simulation, and for interacting with any object in the artificial world of the simulation.