Blue Pelican Java

Graphical Labs
Teacher Manual

Copyright ©, 2008 by Charles Cook;
Refugio, Tx

Table of Contents

Topic	BPJ Text Lesson #	Page #

Demonstrating GridWorld

Run the code:

In this chapter we will get a feeling for how GridWorld works. To do this, import the two files in the *BasicBug* folder into a project in your IDE, compile both classes, and then execute the *main* method of the *BasicBugRunner* class. To run *main* in BlueJ, right-click on the *BasicBugRunner* **class icon** and then click on *void main*. The following graphical interface will appear (If not, it may only appear as an item on the task bar. Just click on it to make it display.)
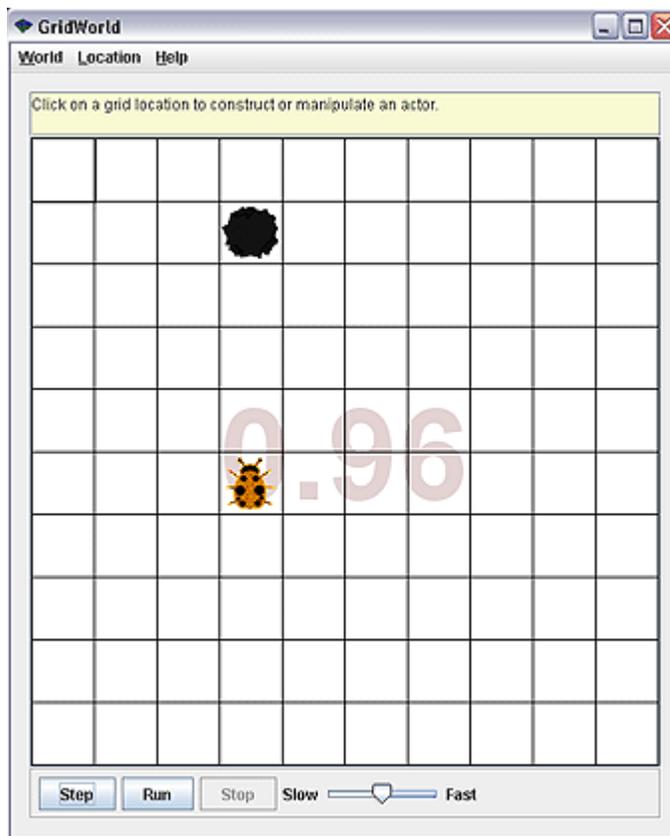
Fig DG-1. GridWorld, *BasicBugRunner* graphical interface

Experiment:

Click on the *Step* button a couple of times and observe the bug moving forward. Now click on any empty cell and get a display similar to the following:
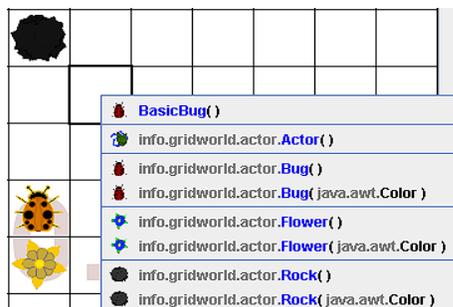
Fig DG-2. The result of clicking on an empty cell.

Click on one of the items in the drop-down menu and add another bug, flower, or rock to the grid at this grid location. Continue experimenting with the controls in this graphic interface and the following facts will soon become apparent:

- Each time the *Step* button is clicked, the *Bug* advances one cell, leaving behind a flower of the same color as the bug. Each time the bug advances, each flower in it's "flower trail" becomes progressively darker, thus showing their age.

- If the bug encounters an obstacle (the black rock in Fig. DG-1) or the edge of the grid, instead of moving forward, it turns 45 degrees clockwise. If it still can't move, it turns another 45 degrees. This turning continues until it can move forward.

- A bug does not consider a flower to be an obstacle. When moving forward into a flower location, the bug "eats" the flower and it disappears.

- Clicking the *Run* button results in the bug continuing to step with a delay between steps determined by the *Slow/Fast* slider.

- The *Stop* button is only active after *Run* is clicked. It stops the *Run* process.

Don't use them:
It is recommended that the *World* and *Location* menu items at the top of the interface not be used with these labs.

Making changes:
Clicking on an object such as a bug, rock, or flower results in the following drop-down menu.
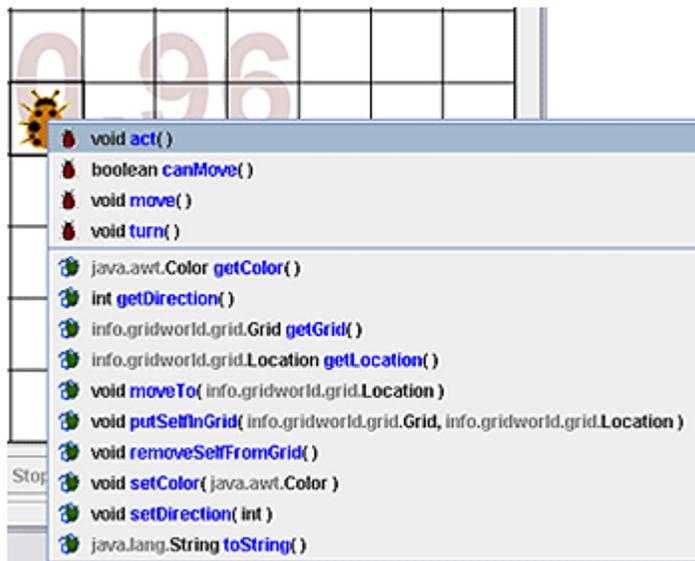
Fig DG-2. The drop-down menu that results from clicking on a *Bug*, *Rock*, or *Flower*.

The drop-down menu:

The items in this object drop-down menu are all **methods** of that object. Clicking on one of these items is, in effect, equivalent to calling that method. The menu items seen here are peculiar to the object selected. The methods available for the *Bug*, *Rock*, and *Flower* are all different. If parameters are required when selecting a method, a dialog box will appear for their entry.

Direction and coordinate conventions:

Some of the methods that can be selected in the above menu samples will call for directions, and GridWorld has its own conventions. Unlike regular mathematics where East is 0 degrees and positive angles rotate counterclockwise, here in GridWorld, **north is 0 degrees and positive angles rotate clockwise**.

Why did the creators of this class do it that way? Supposedly, it is because this is **conventionally how navigation of ships and aircraft is done** (north is 0 degrees with positive angles rotating clockwise.), and here in GridWorld we are "navigating" the grid.

There is yet another difference from conventional mathematics where it is customary to think of the origin as being in the center (sometimes the lower left corner) with the positive vertical axis (the "Y" axis) proceeding upward: **not so in GridWorld**. The **origin (0,0) is the cell in the upper left corner** with the positive direction of the vertical axis proceeding **downward**. The positive direction of the horizontal axis is still to the right as it is conventionally.

Continue to experiment:

Continue to experiment by creating new objects in the grid. Then click on those objects to change their properties. The student should become aware that this is an effective learning technique – experimenting.

BugLab 4 – Using a *for* loop (Blue Pelican Java Lesson 11)

(*Teacher: Although students may not yet be well versed with the meaning of the term "method", it is used below. As a result, hopefully, the student will become accustomed to hearing the term and feel more comfortable when the creation of methods is explored in future lessons.*)

Create the project:

Create a new project (named *BugLab4*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:

Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Look at the source code:

Next, open the source code for the *BasicBug* class (we will not need to modify the *BasicBugRunner* class). Modify the code for this class as follows:

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )  //Executes each click of the Step button.
        {
                if( canMove( ) )
                {
                   move( );
                }
                else
                {
                   turn( );
                }
        }
}
```

Three GridWorld methods:

Just as *main* is a "method", GridWorld has three methods that we will find useful in this lab. We need not understand how they work. All we need to know is that they do work and that they produce the results described below.

canMove( )

All we need to know is that this returns a *boolean*. It will be *true* if the bug sees no obstacle in front of it. A *false* will be returned if the bug can't

move forward; perhaps another bug or rock is in its path, or it's at the edge of the grid.

move( )

This method simply moves the bug forward one cell.

turn( )

The bug faces a new direction by turning 45 degree clockwise.

The task at hand:

Currently, the bug moves forward with one click of the *Step* button and rotates 45 degrees **clockwise** (using the *turn* method) when it can no longer move in its forward direction.

Our job here is to make the bug turn 45 degree **counter clockwise** when it can no longer move in its forward direction. This will be done by replacing the single *turn* in the code above with multiple *turn*s. These **multiple** 45 degree clockwise turns should bring the bug to the eventual direction that is equivalent to a **single** turn of 45 degree counter clockwise.

Instead of replacing the single *turn* in the code above with multiple *turn*'s, create a *for* loop that executes *turn* the appropriate number of times.

BugLab 4 Key:

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )  //Executes each click of the Step button.
        {
                if( canMove( ) )
                {
                   move( );
                }
                else
                {
                   for( int j = 0; j < 7; j++) // 7 X 45 = 315 degrees
                      turn( ); //executes 7 times
                }
        }
}
```

BugLab 5 – Using a *while* loop (Blue Pelican Java Lesson 12)

(*Teacher: Refer to previous labs for a detailed discussion of the various GridWorld methods (act, canMove, move, & turn) used in these labs. If the students have done the previous labs, then they should already be accustomed to the use of these methods.*)

Create the project:
>   Create a new project (named *BugLab5*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:

>>   Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Look at the source code:
>   Next, open the source code for the *BasicBug* class (we will not need to modify the *BasicBugRunner* class). The code for this class should read as follows:

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        //Unique code for each lab to be placed here
}
```

Enter *act* method skeleton:
```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )  // Executes each time the Step button is clicked.
        {

        }
}
```

The task at hand:
>   Currently, the bug moves forward with one click of the *Step* button and rotates 45 degrees clockwise when it can no longer move in its forward direction.

>   Our job here is to redo Lab2 in which we "tried" to make the bug move forward **two** times with each click of the step button. There we discovered that we were unable to guarantee two moves since with each attempted move we might have needed to turn the bug, instead. Now that we have the *while* loop in our arsenal of Java weapons, it is possible to guarantee two moves with each **Step** button click.

Hint: In the *act* method place a *while* loop that stays in the loop as long as *canMove* is false, and inside the loop place the *turn* method. When *canMove* is *true*, the loop is exited and it is safe to *move* the bug. This all ensures that the first *move* will definitely be done **after** any and all necessary turns. Repeat the entire process to ensure the second *move*.

BugLab 5 Key:

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )
        {
                //Stay in this loop and keep turning until canMove is true
                while( !canMove( ) )
                {
                    turn( );
                }
                move( );  //The first move

                //Repeat the entire procedure to produce the second move
                while( !canMove( ) )
                {
                    turn( );
                }
                move( );
        }
}
```

BugLab 6 – Creating new objects (Blue Pelican Java Lesson 15)

(Teacher: Refer to Labs 4 and earlier for a detailed discussion of the various GridWorld methods used in these labs (act, canMove, move, & turn). If the students have done the previous labs, then they should already be accustomed to the use of these methods.)

Create the project:
  Create a new project (named *BugLab6*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:
    Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Modify *BasicBug*:
  Modify the source code for *BasicBug* as shown below. This will cause the bug to exhibit the default behavior (moving forward one cell for each click of the **Step** button or turning clockwise 45 degree if a move is not possible):

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act()
        {
                if( canMove( ) )
                {
                   move( );
                }
                else
                {
                   turn( );
                }
        }
}
```

The task at hand:
  Our job here is to modify the *BasicBugRunner* class so as to produce two bugs and a rock as shown below in Fig.BugLab6–1. A magenta bug should be located in row 1, column 0. A red rock should be in row 3, column 1. A green bug should be in row 3, column 4.

  Notice that rows are numbered from **top to bottom** with 0 being the index of the top row. Columns are numbered from **left to right** with 0 being the index of the far left column.
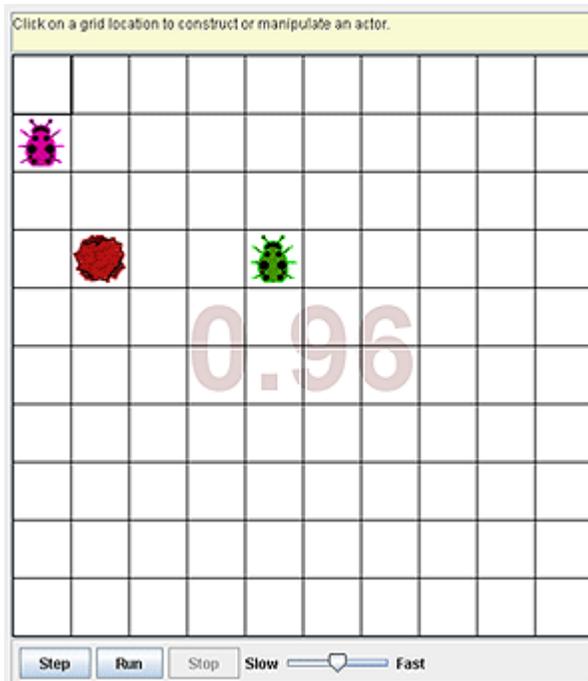
Fig.BugLab6-1
Initial positions of the three
objects.

The existing code for *BasicBugRunner* is shown below. Study it to see how the
bug1 and *rock1* objects are created and placed in the grid at certain locations.
Then in a similar way modify the code to produce the three colored objects in the
locations specified above.

//**This class will not compile until the BasicBug class first compiles.**
```java
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;
import info.gridworld.actor.Rock;
public class BasicBugRunner
{
    public static void main(String args[])
    {
        ActorWorld world = new ActorWorld( );
        BasicBug bug1 = new BasicBug( ); //Create and set color of bug1
        bug1.setColor(Color.ORANGE);

        Rock rock1 = new Rock( ); //Create and set color of rock1
        rock1.setColor(Color.BLACK);

        world.add(new Location(5,3), bug1); //Add to grid and set location
        world.add(new Location(1, 3), rock1);
        world.show( );
    }
}
```

BugLab 6 Key:

```
import java.awt.Color;
import info.gridworld.actor.Rock;
public class BasicBugRunner
{
        public static void main(String args[])
        {
            ActorWorld world = new ActorWorld( );

            BasicBug bug1 = new BasicBug( ); //Create and set color for bug1
            bug1.setColor(Color.MAGENTA);

            BasicBug bug2 = new BasicBug( );  //Create and set color for bug2
            bug2.setColor(Color.GREEN);

            Rock rock1 = new Rock( );
            rock1.setColor(Color.RED);

            //Place objects in the grid and specify locations
            world.add(new Location(1,0), bug1);
            world.add(new Location(3,4), bug2);
            world.add(new Location(3, 1), rock1);
            world.show( );
        }
}
```

BugLab 7 – Passing arguments to a constructor (Blue Pelican Java Lesson 16)

(Teacher: Refer to Labs 4 and earlier for a detailed discussion of the various GridWorld methods used in these labs (act, canMove, move, & turn). If the students have done the previous labs, then they should already be accustomed to the use of these methods.)

Create the project:
> Create a new project (named *BugLab7*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:
>> Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Modify *BasicBug*:
> Modify the source code for *BasicBug* as shown below. This will cause the bug to exhibit the default behavior (moving forward one cell for each click of the **Step** button or turning clockwise 45 degree if a move is not possible):

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )
        {
                if( canMove( ) )
                {
                   move( );
                }
                else
                {
                   turn( );
                }
        }
}
```

The task at hand:
> Our job here is to modify the **both** the *BasicBug* **and** *BasicBugRunner* classes so as to accommodate a constructor in *BasicBug*.

> **Creating the *BasicBug* constructor:**
>> Create a constructor for the *BasicBug* class that will receive as parameters, a *Color* object and an *int* type variable. Then provide the internal code for the constructor as indicated by the following:

```
public BasicBug(Color clr, int numTurns)
{
        //Use clr with the setColor method to set the bug's color.
        //Use numTurns to iterate through a for-loop in which the
        //turn method is executed each time.
}
```

Notice that at the time of creation of a *BasicBug* object, this constructor allows us to set the color of the bug and to orient the bug in a new direction by turning a specified number of times (each turn is 45 degrees).

Add another *import*:
To the *BasicBug* class add the following *import* so as to accommodate the use of the *Color* type:
import java.awt.Color;

Adjusting the *BasicBugRunner* code:
The following line of code in *BasicBugRunner* should be modified to pass to the *BasicBug* constructor, a *Color* object (*Color.GREEN*) and an integer (*3*) specifying the number of times to initially *turn* the newly created *BasicBug* object.

BasicBug bug1 = new BasicBug(?, ?);

Run the *main* method in *BasicBugRunner* and the resulting display should be as follows (notice the **green** bug turned 45 X 3 = 135 degrees):
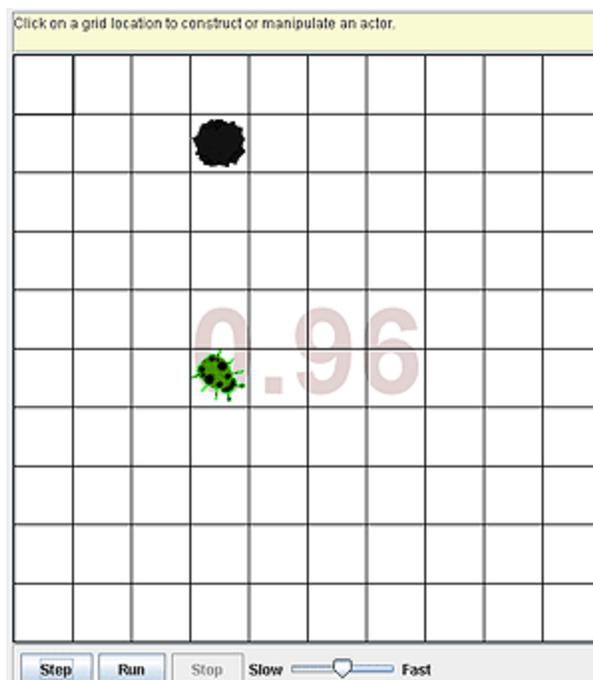
Fig. BugLab7-1 Initial position and orientation of the bug & rock.

BugLab 7 Key:

```
//This class will not compile until the BasicBug class first compiles.
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;
import info.gridworld.actor.Rock;
public class BasicBugRunner
{
      public static void main(String args[])
      {
          ActorWorld world = new ActorWorld( );

          //Make a green bug and turn 45 X 3 = 135 degrees
          BasicBug bug1 = new BasicBug(Color.GREEN, 3);

          Rock rock1 = new Rock( );
          rock1.setColor(Color.BLACK);

          //Add the objects to the grid at the specified locations
          world.add(new Location(5,3), bug1);
          world.add(new Location(1, 3), rock1);
          world.show( );
      }
}
**********************************************************************
import info.gridworld.actor.Bug;
import java.awt.Color;
public class BasicBug extends Bug
{
          public BasicBug(Color clr, int numTurns){  //Constructor
                  setColor(clr); //Color the bug
                  for(int j = 0; j < numTurns; j++)  //Turn numTurns times
                     turn( );
          }

          public void act( ){
                  if( canMove( ) ){
                     move( );
                  }
                  else
                  {
                     turn( );
                  }
          }
}
```

BugLab 23 – Using recursion (Blue Pelican Java Lesson 39)

(Teacher: Refer to Labs 4 and earlier for a detailed discussion of the various GridWorld methods used in these labs (act, canMove, move, & turn). If the students have done the previous labs, then they should already be accustomed to the use of these methods.)

Create the project:
> Create a new project (named *BugLab23*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:
>> Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in the two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Modify *BasicBug*:
> The following code for *BasicBug* produces it's default behavior; the bug moves when it can and when it can't, it turns 45 degrees clockwise.

```
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
        public void act( )
        {
           if(canMove( ))
           {
              move( );
           }
           else
           {
              turn( );
           }
        }
}
```

The task at hand:
> We wish to modify the code above so that ***act* is recursively called** so as to continually move the bug forward. Recursion should stop when the bug needs to turn. As a result, each time the <u>Step</u> button is clicked, the bug should be seen to move forward as far as it can (until a turn is necessary). It will then stop and wait for the next click of the <u>Step</u> button.

While recursion is one of the more sophisticated concepts in computer science, this particular lab is one of the easiest. Actually, only one line of code needs to be added to make the bug perform as described.

BugLab 23 Key:

```java
import info.gridworld.actor.Bug;
public class BasicBug extends Bug
{
  public void act( )
  {
    if(canMove( ))
    {
      move( );
      act( );  //Recursion occurs here.
    }
    else
    {
      //Recursion is terminated here.
      turn( );
    }
  }
}
```

BugLab 28– Sorting with *Comparator* (Blue Pelican Java Lesson 44)

(Teacher: Refer to Labs 4 and earlier for a detailed discussion of the various GridWorld methods used in these labs (act, canMove, move, & turn). If the students have done the previous labs, then they should already be accustomed to the use of these methods.)

Create the project:

Create a new project (named *BugLab28*) with your IDE and into the resulting folder, import the two classes in the *BasicBug* folder. In the IDE BlueJ, it's done as follows:

Create the project with **Project** | **New project**, being careful to create this project within the *C:\GridworldProjects* folder. Then bring in the two classes mentioned above with **Project** | **Import** (Navigate to the *C:\GridWorldProjects\BasicBug* folder and then click **Import**.)

Modify *BasicBugRunner*:

This lab is very similar to *BugLabs 24-27*. Use the exact same *BasicBugRunner* class as in those labs. It creates three bugs and one rock.

Modify *BasicBug*:

The *BasicBug* class will have the following code. Notice the area of "Modifications go here." **That is where a *Comparator* object will be used to sort the *locStr* array**:

```
import info.gridworld.actor.Bug;
//All these imports are necessary for the getLocationArray method
import info.gridworld.grid.*;
import java.util.*;
import info.gridworld.actor.Actor;
public class BasicBug extends Bug
{
   public void act( )
   {
            if(canMove( ))
            {
                    move( );
            }
            else
            {
                    turn( );
            }
            String locStr[] = getLocationArray( );

            //Sort using a comparator

            ….Modifications go here….
```

```
            //Print the sorted array
            for(int j = 0; j < locStr.length; j++)
            {
                    System.out.println(locStr[j]);
            }
            System.out.println( );  //Print a blank line.
    }


    //This method produces a String array in "r,c" format of the
    //coordinatesof all occupied cells.
    public String[] getLocationArray( )
    {
            //Get the coordinates of all occupied cells.
            Grid<Actor> gr = getGrid( );  //Obtain the grid object
            //locList is an ArrayList of all the occupied locations.
            ArrayList<Location> locList = gr.getOccupiedLocations( );
            String locStr[] = new String[locList.size( )];
            for(int j = 0; j < locList.size( ); j++)
            {
                    Location loc = locList.get(j);
                    //Stuff a String array with row & col numbers separated
                    //by a comma.
                    locStr[j] = "" + loc.getRow( ) + "," + loc.getCol( );
            }
            return locStr;

    }
}
```

Again, it is **not necessary** to understand the *getLocationArray* method (just copy and paste): all that is necessary is to know that it returns a *String* array containing the coordinates of occupied cells. Unlike several previous labs, notice that the *String* array returned by *getLocationArray* is in standard row-column form.

The task at hand:
Create a class that *implements Comparator* and then implement the *compare* method inside it. The two *Objects* it receives will be *String*s in the form "3,5". Convert these numbers into a product (15 in this example) and compare those products inside the *compare* method.

Use the class thus produced to create a *Comparator* object and this pass that object along with the *String* array, *locStr*, to the *Arrays.sort* method. Place this code in the code area, "Modifications go here."

Testing the code:
Run *main* in *BasicBugRunner*, advance the bugs by clicking the Step button

several times, and observe the printout on the console screen. It should be recognized that the *act* method executes for **each** bug **each** time the Step button is clicked. As the bugs advance they leave behind a trail of *Flower* objects. Thus, the number of objects on the grid increases with each click of the Step button.

The following output is produced with a single click of the Step button. Notice the order is with respect to the **product** of the row and column numbers.

```
1,3
1,7
2,7
5,3
4,6

1,3
1,7
2,7
5,3
3,6
4,6

1,3
1,7
4,3
2,7
5,3
3,6
4,6
```

BugLab 28 Key:

```
import info.gridworld.actor.Bug;
//All these imports are necessary for the getLocationArray method
import info.gridworld.grid.*;
import java.util.*;
import info.gridworld.actor.Actor;
public class BasicBug extends Bug
{
        public void act( )
        {
                if(canMove( ))
                {
                        move( );
                }
                else
                {
                        turn( );
                }
                String locStr[] = getLocationArray( );

                //Sort using a comparator
                Comparator cmptr = new LocStrComparator( );
                Arrays.sort(locStr, cmptr);

                //Print the sorted array
                for(int j = 0; j < locStr.length; j++)
                {
                        System.out.println(locStr[j]);
                }
                System.out.println( );  //Print a blank line.
        }

        //This method produces a String array in "r,c" format of the coordinates
        //of all occupiued cells.
        public String[] getLocationArray( )
        {
                //Get the coordinates of all occupied cells.
                Grid<Actor> gr = getGrid( );  //Obtain the grid object
                //locList is an ArrayList of all the occupied locations.
                ArrayList<Location> locList = gr.getOccupiedLocations( );
                String locStr[] = new String[locList.size( )];
                for(int j = 0; j < locList.size( ); j++)
                {
                  Location loc = locList.get(j);
                  //Stuff a String array with row & col numbers separated by comma.
```

```java
                        locStr[j] = "" + loc.getRow( ) + "," + loc.getCol( );
                    }
                    return locStr;
            }
}

import java.util.*;
public class LocStrComparator implements Comparator
{
        //Compare Strings in the format "r,c" according to the product r*c
        public int compare(Object firstObj, Object secondObj)
        {
            String s1 = (String)firstObj;  //Typical String, "3,5"
            String s2 = (String)secondObj;

            //product1 will be the product of the row # and column # in s1
            String sp[] = s1.split(",");
            int r = Integer.parseInt(sp[0]);
            int c = Integer.parseInt(sp[1]);
            int product1 = r * c;

            //product2 will be the product of the row # and column # in s2
            sp= s2.split(",");
            c = Integer.parseInt(sp[0]);
            r = Integer.parseInt(sp[1]);
            int product2 = r * c;

            int retValue;
            if(product1 < product2)
            {
                    retValue= -1;
            }
            else
            {
                    if(product1 > product2)
                    {
                            retValue = 1;
                    }
                    else
                    {
                            retValue = 0;
                    }
            }
            return retValue;
        }
}
```