Blue Pelican GridWorld

Student Manual

AP Computer Science
Case Study

Table of contents

Chapter 1--Getting Started

The new case study, GridWorld

The College Board has prepared a case study that is part of the Computer Science AP test. This case study is new to the 2007-2008 school year and replaces the older Marine Biology case study. This particular case study is graphically oriented and can be tested with a graphical interface that is part of the provided code.

It should be emphasized that **the graphical part** is only for the convenience of the student for testing/proving code and is **not actually part of the AP test**. The graphical nature of this project, of course, makes it much more engaging/interesting for the student.

Why have a case study?

The reasoning behind this part of the test is to give students an opportunity to:

- Become familiar with a lengthy project written by another person.
- Observe top quality, rigorous code design and documentation.
- Work with a significant, non-trivial program.
- Become familiar with a master/apprentice relationship.

Download from the College Board

The first step will be to go to the College Board web page and download three things: (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html)

- The Student Manual (630kb .pdf) This document provides the following:
  - A brief overview of the project, especially the graphical interface
  - A discussion of the major classes in the project along with questions/exercises
  - *Grid* data structures
  - Appendices containing source code for the classes
  - A quick reference document

  This Blue Pelican booklet provides all of the essentials contained in the Student Manual from the College Board, and much more; however, you may still find the AP Student Manual useful.
- The installation guide (172kb, .pdf)
- The source code for GridWorld (238kb, .zip)

Installing GridWorld

Unzip the source code and place it in a folder called *C:\GridWorldCode* on your computer. In this booklet, the assumption is made that you have placed the GridWorld code in this folder.

Make sure your version of Java is 5.0 or higher. A good place to find the correct version to download and install Java on your computer is the BlueJ web site (www.bluej.org)

The next step will be to install the JAR file. This is a collection of files that provides the "behind the scenes" classes necessary for the graphical interface and other features of GridWorld. The GridWorld Installation Guide (one of the downloads mentioned above) gives detailed instructions on how to install the JAR file for the following IDE's:

- Eclipse 3.1
- BlueJ 2
- JCreator (Windows only)
- JJ (Windows only)
- TextPad 5.0 (Windows only)

Demonstrating the *BugRunner* project

GridWorld has several ready made classes that provide demonstrations of what can be done with this case study. The first and simplest is *BugRunner* and is initially discussed in the Student Handbook from the College Board mentioned above. Here is how to build a *BugRunner* project using the IDE, BlueJ:

- Load BlueJ and create a new project. It is suggested that a folder different from *C:\GridWorldCode* be used to store your projects. *C:\GridWorld_projects* would be a good name. Call this new project *BugRunner*.
- Go to the BlueJ menu item *Project* | *Import*… and navigate to the *C:\GridWorldCode\projects\firstProject* folder. Leave the file name as *C:\GridWorldCode\projects\firstProject* and click on the *Import* button.
- Right click on the *BugRunner* class icon that now appears and compile it.
- Right click on the *BugRunner* class icon and select *void main* to run the program. 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.)

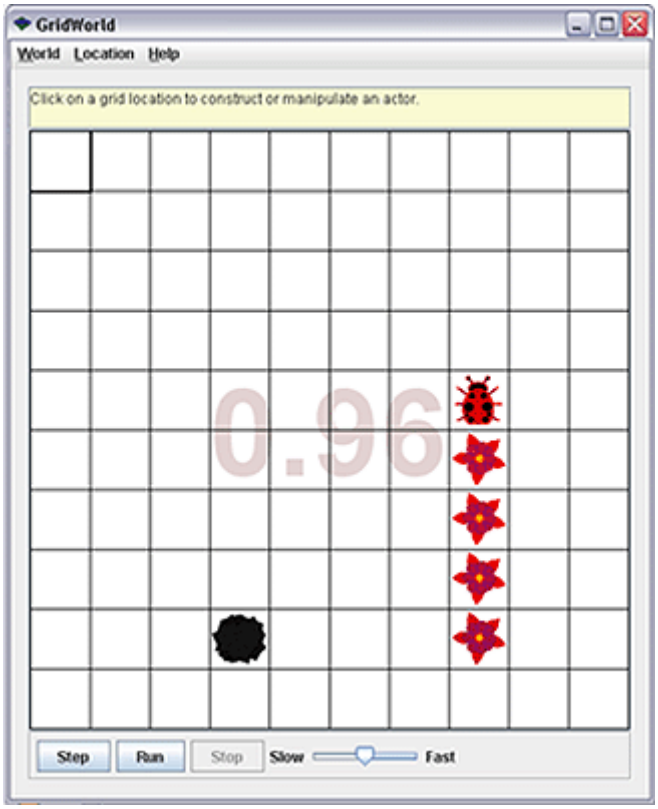

Fig 1-1. GridWorld, *BugRunner* graphical interface

To get an idea of what GridWorld is all about, experiment 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 object in Fig. 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.
- 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.

The menu items

There are two significant menu items: *World* and *Location*. These give a hint of code that will be written later because they allow methods to be accessed that create objects or cause things to happen.

World menu item

Fig 1-2. World menu item

The menu item *BoundedGrid(int rows, int cols)* allows a new grid to be created with a set number of rows and columns. *UnboundedGrid( )* produces an infinite grid for which scroll bars are provided as a navigation aide.

Location menu item

Click on any cell, and then press the *Esc* key to dismiss the resulting menu. Notice a heavy square now highlighting the cell. Click on the *Location* menu and navigate to other cells with the items *Up*, *Down*, etc. (The four arrow keys on the keyboard will do the same thing.) When a cell is selected having an object, use the menu items *Edit* and *Delete* to perform functions as their names suggest. Fig 3.

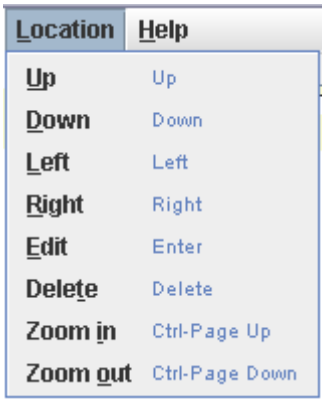

Fig 1-3. The *Location* menu item

Clicking on an empty cell

Fig 1-4. The drop-down menu that results from clicking on an empty cell.

This drop-down menu provides a list of constructors that, when selected, will cause an object of that type to be created and inserted in the highlighted cell. If, for example, *Bug(java.awt.Color)* is selected, notice that a parameter is required. A dialog box will appear in which the desired color can be entered.

Clicking on an object (bug, rock, flower)

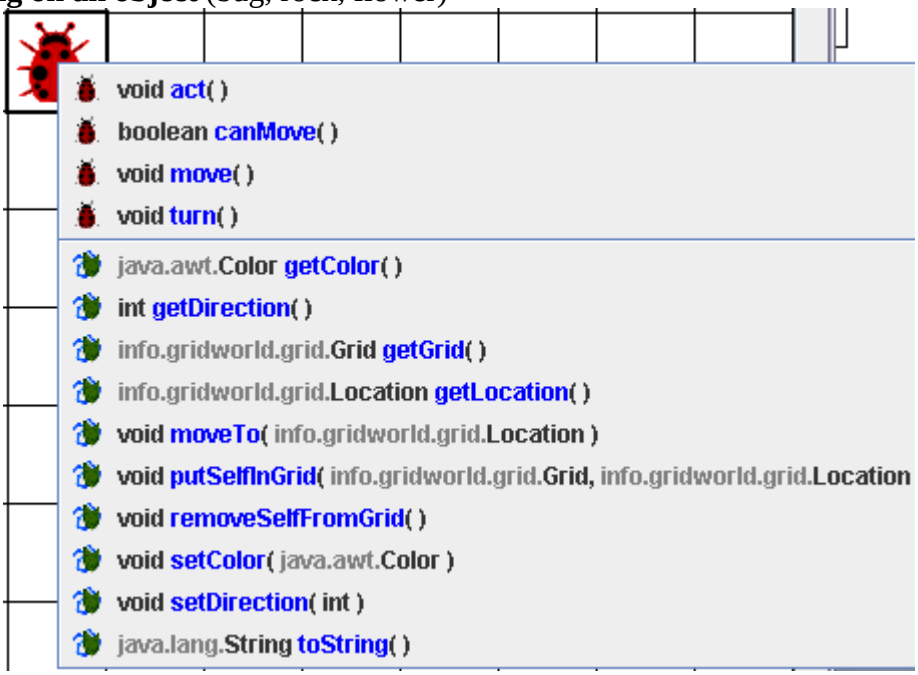

Fig 1-5. Object drop-down menu

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. Once again, if parameters are called for 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 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.

These coordinates help locate the positions of the various objects. As will be presented later, each object has a *Location* object associated with it that specifies a row and column.

It will be emphasized again that the **graphical part** of GridWorld is **not testable** on the AP Computer Science test. Its real value is that it provides an easy way to **prove and debug code** that **is** testable.

Exercise, Chapter 1

1. In which direction does a *Bug* always move?
2. Under what circumstances might the *Bug* be unable to move?
3. When *Step* is clicked, what happens when the *Bug* is unable to move?
4. When a *Bug* moves, what will occupy the cell that it just vacated?
5. How many objects can occupy a *Grid* cell simultaneously?
6. Can a *Rock* object be moved? If so, what method would move it?
7. What method sets a new direction for the *Bug*?
8. Are the *Rock* objects as displayed, graphically symmetrical in shape?
9. Based on you answer in #7, would it make visual sense for it to be possible to set a direction for a *Rock* object? Why?
10. Does a *Flower* object move with each *Step*? If so, where? If not, is it possible to move a *Flower* object?
11. What does a *Bug* do when it tries to move and it encounters the edge of the grid or a *Rock*?
12. What happens when a *Bug* encounters a *Flower*?
13. What happens when a *Rock* is moved on top of a *Bug*?
14. To change the direction of a *Bug* to west, what parameter would you send to the *setDirection* method?
15. What would happen when, instead of turning, an attempt is made to move a *Bug* outside the *Grid*?

Chapter 2--*BoxBug* & *SpiralBug*

Modifying the methods of *Bug*

The *Bug* class is a very fundamental part of GridWorld. It should **not be modified**; rather, a new class is created **extending** the *Bug* class, and modifications are made in it by overriding the methods in the *Bug* superclass. One method that is very commonly overridden is the *act( )* method.

Cleaning up our *act( )*

Recall from the last chapter (Getting Started), the *Step* button on the graphical interface to GridWorld. Each time it is clicked (and also on each iteration of *Run*), the *act* method of each object in the *Grid* is called. Below is the source code for the *act* method of the *Bug* class:

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

Notice how very simple this method is. It, in turn, uses three other methods of the *Bug* class:

- *canMove( )* … returns a *boolean* telling if it's safe to move in the direction set for this object.
- *move( )* …move one space to the nearest of this object's direction to horizontal, vertical, or at a 45 degree diagonal.
- *turn( )* …sets a new direction of 45 degrees clockwise from the current direction.

Notice that this code explains why when a *Bug* wants to move into the position of a *Rock*, another *Bug*, or is trying to move off the grid, it turns, instead. Also notice that with just a few changes, this is very fertile ground for **modifying the behavior** of the *Bug*.

BoxBug

The *Bug* class will now be extended to produce the *BoxBug* class. As its name suggests, *BoxBug* will travel in the shape of a box (square). The *BoxBug* will move along in its initial direction for a distance specified by the state variable (instance field) *sideLength*. It will then turn 90 clockwise and continue doing this unless it encounters an obstacle in which case it also turns 90 degrees clockwise and begins a new box.

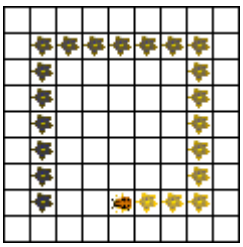

Fig 2-1. When testing the *BoxBug* class, the graphics should produce something like this for each *BoxBug* object on the Grid.

It has already been suggested that we will have an integer state variable called *sideLength* that determines the lengths of the sides of the square traced out by *BoxBug*. A good feature for this new class to have would be for its constructor to initialize *sideLength* as follows:

```
public BoxBug(int length)
{
        sideLength = length;
        steps = 0;
}
```

Notice that there is now evidence of a second state variable, *int steps*. For the sake of knowing when to turn 90 degrees, this variable keeps a tally of how many steps through which the *BoxBug* has progressed. Also, notice that this constructor specifies how *BoxBug* objects should be created:

```
BoxBug myBoxBug = new BoxBug( len );  //int len specifies side length
```

So far, the new *BoxBug* class appears as follows (notice ***extends Bug***):

```
import info.gridworld.actor.Bug;

public class BoxBug extends Bug
{
        //state variables
        private int sideLength;
        private int steps;

        //constructor
        public BoxBug(int length)
        {
                sideLength = length;
                steps = 0;
        }

        //…more code to come…
 }
```

Finally, and most important of all, a modified *act* method must be provided that overrides the *act* method of the *Bug* superclass. The requirements are that it keeps up with how far the *BoxBug* has moved and then turns it 90 degrees clockwise.

Project… *BoxBug*

As a project, complete the *BoxBug* class by providing code for the *act* method so that the behavior of *BoxBug* is as described: after turning 90 degrees be sure to reset *steps* to 0 so the count can start over. To test this class, see the next section titled, **Testing with a new *Runner* class.**

Testing with a new *Runner* class

(This discussion applies to testing a *BoxBug* class. A *Runner* class could be similarly created for any other modified type of *Bug*.)

Now that a *BoxBug* class has been created, how is it to be tested? First, create a new project: call it *BoxBug* and create the *BoxBug* class within it. The actual visual testing must be done with a *BoxBugRunner* class. This is **not an AP tested class**, but is necessary for the testing of *BoxBug* and to see it perform. Enter a second class into the project called *BoxBugRunner* as follows:

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;

public class BoxBugRunner
{
	public static void main( String args[] )
	{
		ActorWorld world = new ActorWorld( );
		BoxBug bug1 = new BoxBug(6);  //side of box = 6
		bug1.setColor(Color.ORANGE);

		BoxBug bug2 = new BoxBug(3);  //side of box = 3
		bug2.setColor(Color.GREEN);

		world.add (new Location(7, 8), bug1 );
		world.add (new Location(7, 5), bug2 );
		world.show( );
	}
}
```

Again this code is **not part of the AP test**. This is just a class we need to provide in order to test our *BoxBug* class with a graphical interface. One thing is; however, of importance if we wish to create other extensions of the *Bug* class. If for example, a spiral bug is created with a *SpiralBug* class, then the following two lines of code would replace the corresponding two lines in the *BoxBugRunner* class:

```
SpiralBug bug1 = new SpiralBug(6);
SpiralBug bug2 = new SpiralBug(6);
```

This new class could be called the *SpiralBugRunner* class.

It should be noted that this runner class (either *BoxBugRunner* or *SpiralBugRunner*) will not compile unless the class (*BoxBug* or *SpiralBug*), upon which it is dependent, has already been compiled.

Project… *SpiralBug*

As a project, create a *SpiralBug* class by providing code for the *act* method so that it moves in a spiral. A key feature is to use most of the *BoxBug* class and increase the value of *sideLength* at the end of each turn. To test this class, see the previous section titled, **Testing with a new *Runner* class.** When testing, set an unbounded grid.

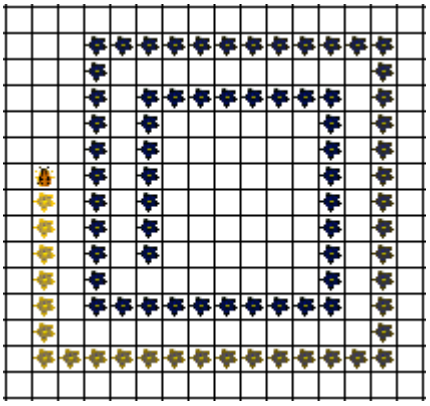

Fig 2-1. When testing the *SpiralBug* class, the graphics should produce something like this for each *SpiralBug* object on the grid

Chapter 3… The *Location* Class

Rows first, then columns

The official source code for the *Location* class is given in Appendix A. This class is able to specify *Location*s in the grid (a row and column). As is conventional throughout computer science and mathematics, a position is typically given as R,C with the **row always being specified first**. (Just think of RC Cola.) Rows are numbered from top-to-bottom starting with *0* at the top. Columns are more conventionally numbered left-to-right starting with *0* at the far left.

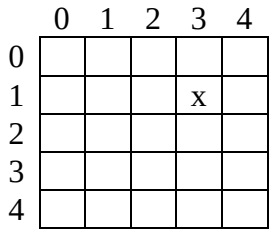

Fig 3-1. Numbering scheme for the cells in a grid. The x is in row = 1 and column = 3, commonly abbreviated as (1,3).

Direction constants

Another useful feature of the *Location* class is the list of *static* constants that specify direction. Before discussing this list of constants, a review of what is meant by "direction" is in order. Direction is always given as an integer degree value referenced from north (up on the grid). Hence, east is 90, south is 180, etc. The *Location* class provides the following constants that are subdivided into two groups, compass directions and turn angles:

Compass directions (given in increments of 45 degrees)

```
public static final int NORTH = 0;
public static final int NORTHEAST = 45;
public static final int EAST = 90;
public static final int SOUTHEAST = 135;
public static final int SOUTH = 180;
public static final int SOUTHWEST = 225;
public static final int WEST = 270;
public static final int NORTHWEST = 315;
```

Turn angles

```
public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;
```

The *Location* constructor

The signature of the constructor is:

```
public Location(int r, int c)
```

Example:

The appropriate way to declare and instantiate a *Location* object (at row 1 and column 3) named *loc* would be:

```
Location loc = new Location(1, 3);
```

Two accessor methods

Method signatures:

```
public int getRow( );
public int getCol( );
```

Example:

The following code returns the row and column indices for the *loc* object:

```
int theRow = loc.getRow( );
int theCol = loc.getCol( );
```

The *getAdjacentLocation* method

Method signature:

```
public Location getAdjacentLocation(int direction)
```

Example (using Fig 3-1. above where *loc* is pointing to the cell given by (1,3)):

```
Location loc1 = loc.getAdjacentLocation(Location.SOUTHEAST);
System.out.println( "(" + loc1.getRow( ) + "," + loc1.getCol( ) + ")" );
//prints (2, 4)
```

The *getDirectionToward* method

Method signature:

```
public int getDirectionToward(Location target)
```

Example (using Fig 3-1. above where *loc* is pointing to the cell given by (1,3)):

```
Location loc2 = new Location(3,1);
int dir = loc.getDirectionToward(loc2);
System.out.println(dir);  //prints 225
```

The *equals* method

This method returns a *true* if the two methods being compared have **identical** row and column positions; otherwise, *false*.

Method signature:

```
public boolean equals(Object other)
```

Example:

```
Location loc1 = new Location(2,3);
Location loc2 = new Location(4,5);
Location loc3 = new Location(2,3);

boolean b1 = loc1.equals(loc2); //false
boolean b2 = loc1.equals(loc3); //true
```

The *compareTo* method

Method signature:

public int compareTo(Object other)

Example:

```
Location loc1 = new Location(2,3);
Location loc2 = new Location(2,5);

int cmp =  loc1.compareTo(loc2);  //returns a negative number
```

This method first compares the rows of *loc1* and *loc2* using the traditional rules of *compareTo*. What it really does is return this: *loc1.getRow( ) – loc2.getRow( )*. If this is *0*, as it is in our example above, then it compares the columns and returns this: *loc1.getCol( ) – loc2.getCol( ).* For our example this is *3 – 5 = -2.*

Exercise, Chapter 3

For the problems below, use the following code:

```
Location loc1 = new Location(12, 3);
Location loc2 = new Location(8, 3);
Location loc3 = new Location(12, 1);
Location loc4 = new Location( loc3.getRow( ), loc3.getCol( ) + 2 );
```

1. What is returned by *loc1.equals(loc2)*?

2. What is returned by *loc1.compareTo(loc2)*?

3. What is returned by *loc1.compareTo(loc3)*?

4. What is returned by *loc1.equals(loc4)*?

5. What is returned by *loc1==loc4*?

6. Which of the following are legitimate ways to specify a direction of 45 degrees toward the upper left? (There may be more than one answer.)

   A. 45 B. -45 C. Location.NORTHEAST D. Location.NORTHWEST
   E. NORTHWEST F. Location.LEFT + Location.HALF_RIGHT

7. Using *Location* methods, write code that will store in *int i* the product of the rows in *loc1* and the number of columns in *loc4*.

8. What is the (row, column) position of *loc* after the following statement is executed?

   ```
   Location loc = loc1.getAdjacentLocation(Location.East);
   ```

9. What is the (row, column) position of *loc* after the following statement is executed?

```
Location loc = loc1.getAdjacentLocation(Location.West + Location.RIGHT);
```

10. How would you determine if the row position of loc1 is even or odd?

11. What is stored in *dir* by the following?

```
int dir = loc1.getDirectionToward(loc3);
```

12. What is stored in *dir* by the following?

```
int dir = loc1.getDirectionToward( new Location(13, 4) );
```

Project… the *compareTo* method

Try to imagine how the creators of GridWorld created the code that implement the *compareTo* method within the *Location* class. Write this code so as to make this method perform as described on page 3-3.

Project… *ZorroBug*

Create a *ZorroBug* that will make a Z and then stop. Use *static* constants (compass directions) from the *Location* class to turn the bug to a specific direction using the *setDirection* method of the *Bug* class (signature: *public void setDirection(int dir)*). If an obstacle is encountered, just immediately try to do the next segment of the Z. Start the *ZorroBug* on the upper left portion of the Z.

Using the *Location* class will necessitate the following import:

```
import info.gridworld.grid.Location;
```

Chapter 4… The *Grid* Interface

Since *Grid* is an interface, it does not implement any of its methods. Implementation is done in the two classes, *BoundedGrid<E>* and *UnboundedGrid<E>* where the generic type parameter *E* specifies the type of the objects contained within these *Grids*: the parameter type is always *Actor* here in GridWorld. See Appendix AF in the Blue Pelican Java textbook for an explanation of generic classes.

Preconditions

Most *Grid* methods have preconditions; however, an implied precondition of all is that passed object parameters not be *null*. Trying to specify a *Grid Location* that is invalid is particularly troublesome; therefore, the first method (*isValid*) in the list below is useful in avoiding this pitfall.

The *isValid* method

This method returns *true* if *loc* is valid in this *Grid*, *false* otherwise.

Method signature:

```
boolean isValid(Location loc)
```

Example:

```
if( theGrid.isValid( new Location(118, 46) )
{    …some code …   }
```

The *put* method

This method places the object *obj* at the location specified by *Location loc*. It returns the object that was previously at this location. A *null* is returned if nothing was previously there.

Method signature:

```
E put(Location loc, E obj)  //E is a generic object type
```

Example:

```
Rock myRock = new Rock ( );
Location loc = new Location(5, 23);
Actor prevObj = theGrid.put(loc, myRock);
```

The *remove* method

This method removes the object at the specified *Location* and returns the object that occupied that *Location*. A *null* is returned if there was no occupant.

Method signature:

```
E remove(Location loc)  //E is a generic object type
```

Example:

```
Actor actr = theGrid.remove( new Location(5,0) );
```

Warning: When adding or removing *Actor* objects, do not use either *put* or *get* since these methods do not update the *location* and *grid* state variables of *Actor* objects (see Chapter 5).

The *get* method

This method does exactly the same thing as the *remove* method, except it is non-destructive (nothing is removed).

Method signature:

```
E get(Location loc)  //E is a generic object type
```

Example:

```
Actor actr = theGrid.get( new Location(5, 0) );
```

The *getOccupiedLocations* method

This method returns an *ArrayList* object containing *Location* objects representing **all** occupied locations within a *Grid*.

Method signature:

```
ArrayList<Location> getOccupiedLocations( )
```

Example:

```
ArrayList<Location> aryLst = theGrid.getOccupiedLocations( );
```

The *Grid* interface has four convenient methods that gather information about the immediate surroundings of a particular *Location*. Either the adjacent *Location*s or objects in those *Location*s can be returned.

The *getValidAdjacentLocations* method

This method returns (as an *ArrayList*) all valid locations directly adjacent to the specified *Location* in this grid. There could be as many as eight valid, adjacent *Location* objects.

Method signature:

```
ArrayList<Location> getValidAdjacentLocations(Location loc)
```

Example:

```
ArrayList<Location> aryLst = theGrid.getValidAdjacentLocations( loc );
```

The *getEmptyAdjacentLocations* method

This method returns (as an *ArrayList*) all *Location*s directly adjacent to the specified *Location*, containing **no** objects. There could be as many as eight such adjacent cells.

Method signature:

```
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
```

Example:

```
ArrayList<Location> aryLst = theGrid.getEmptyAdjacentLocations( loc );
```

The *getOccupiedAdjacentLocations* method

This method returns (as an *ArrayList*) all *Location*s directly adjacent to the specified *Location*, containing objects. There could be as many as eight such adjacent cells.

Method signature:

```
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
```

Example:

```
ArrayList<Location> aryLst = theGrid.getOccupiedAdjacentLocations( loc );
```

The *getNeighbors* method

This method returns (as an *ArrayList*) all objects in the occupied *Locations* adjacent to the specified *Location*.

Method signature:

```
ArrayList<E> getNeighbors(Location loc)
```

Example:

```
ArrayList<Actor> aryLst = theGrid.getNeighbors( new Location(23, 5) );
```

The *getNumRows* method:

This method returns the number of rows in a *BoundedGrid* (-1 for an *UnboundedGrid*).

Method signature:

```
int getNumRows( )
```

Example:

```
int n = theGrid.numRows( );
```

The *getNumCols* method:

This method returns the number of columns in a *BoundedGrid* (-1 if an *UnboundedGrid*).

Method signature:

```
int getNumCols( )
```

Example:

```
int n = theGrid.numCols( );
```

Exercise, Chapter 4

1. What code will determine if *Location* (5,13) is on the *Grid*?

2. Write code that will print the total number of objects in a *Grid*.

3. What is the difference between the two methods, *getOccupiedLocations* and *getOccupiedAdjacentLocations*?

4. In a *Grid* of 20 rows by 30 columns, what would be the maximum number of objects in the *ArrayList* that *getOccupiedLocations* could return?

5. How could the number of empty cells in a *Grid* be determined?

6. Why does *Grid* not have any code?

7. What classes have full implementation of *Grid*?

8. What do *getRows( )* and *getCols( )* return for an *UnboundedGrid*?

9. The *validAdjacentLocations* method can have a many as eight elements in its returned *ArrayList* object. Using *Location* constants, list the directions of these eight cells relative to the specified *Location*.

10. Several of the methods of the *Grid Interface* return an *ArrayList* object. What advantage does this have over an ordinary array?

11. Suppose an *ArrayList* object called *aryLst* is returned by *getValidAdjacentLocations( )*. Using an enhanced *for*-loop, show how to iterate through the *Location* objects in *aryLst* and determine if any objects are in the list. Print *true* if any objects are present; otherwise, print *false*.

Project… creation of *getEmptyAdjacentLocations*

The method *getEmptyAdjacentLocations* is actually implemented by using the *getValidAdjacentLocations* method and applying the *get* method to the *Locations* objects within its returned *ArrayList*. Each object returned by the *get* method is tested to see if it is *null*. If not, the *Location* object is added to an *ArrayList* object to be returned by this method.

Implement *getEmptyAdjacentLocations* **without** using *getValidAdjacentLocations* as described above.

> Hint: Use the *getAdjacentLocation* method by repeatedly (eight times) passing it increments of 45 degrees. Pass the returned *Location* object to the *get* method, and then test to see if its returned value is *null*.

Chapter 5… The *Actor* Class

By extending the *Actor* class, the *Rock*, *Flower*, *Bug*, and *Critter* classes are created; therefore, it is important to become familiar with the *Actor* class before examining these extending classes.

******** Retrieving properties of an *Actor* class using accessor methods********

The *getColor* method

This method returns the color of the *Actor* object. These colors are *Color* constants of the *java.awt.Color* class: for example, *Color.BLUE*, *Color.RED*, etc. At the time of creation of an *Actor* object, the constructor assigns a default color of blue.

Method Signature:

```
Color getColor( )
```

Example:

```
if( theRock.getColor( ) = = Color.Yellow )
        System.out.println("Yes, it's yellow!");
```

The *getDirection* method

This method returns an integer that represents the north referenced (positive, clockwise) direction assigned to this object. At the time of creation of an *Actor* object, the constructor assigns a default direction of 0 (north).

Method signature:

```
int getDirection( )
```

Example:

```
int dir = theBug.getDirection( );
```

The *getGrid* method

This method returns the *Grid* in which the *Actor* is assigned. A *null* is returned if the *Actor* is not contained in a *Grid*.

Method signature:

```
Grid<Actor> getGrid( )  //notice the required type parameter
```

Example:

```
Grid<Actor> gr = theFlower.getGrid( );
```

The *getLocation* method

This method returns the *Location* object specifying the position of the *Actor*. A *null* is returned if the *Actor* is not contained in a *Grid*.

Method signature:

```
Location getLocation( )
```

Example:

```
Location loc = getLocation( );
```

******** Changing the color and direction of an *Actor* object ********

The *setColor* method

This *void* method sets the color of the *Actor*. Arguments passed to this method must be of type *Color*: for example, *Color.BLUE, Color.RED*, etc.

Method signature:

```
void setColor(Color newColor)
```

Example:

```
theRock.setColor(Color.YELLOW);
```

The *setDirection* method

This *void* method lets us specify an integer that is the direction in which the *Actor* object is "pointed." It is normal to specify a direction using a class constant of the *Location* class: Location.NORTHEAST, Location.RIGHT, etc.

Method signature:

```
void setDirection(int newDirection)
```

Example:

```
spiralBug.setDirection( Location.SOUTHWEST );
```

********* Adding/removing an object from a *Grid* *********

The *putSelfInGrid* method

This void method places the *Actor* in the specified *Grid* at the specified *Location*; however, as a precondition, the *Actor* must **not** already by in a *Grid*. This method is *void* and returns nothing. If another *Actor* is at the specified location, it will be removed.

Warning: Do not directly use the *put* method of the *Grid* class to try to place an *Actor* in a *Grid* since this action this **does not** update the *Location* and *Grid* state variables of the *Actor* object.

Method signature:

```
void putSelfInGrid( Grid<Actor> gr, Location loc )
```

Example:

```
redRock.putSelfInGrid( myGrid, new Location(22, 14) );
```

The *removeSelfFromGrid( )* method

This *void* method removes the *Actor* from its *Grid* and sets its *grid* and *location* state variables both to *null*. A precondition is that the *Actor* be contained in a *Grid*.

Warning: Do not use directly the *remove* method of the *Grid* class to try to remove an *Actor* from a *Grid* since this action this **does not** update the *location* and *grid* state variables of the *Actor* object.

Method signature:

```
void removeSelfFromGrid( )
```

Example:

```
myFlower.removeSelfFromGrid( );
```

********* Moving an *Actor* to a new Location*********

The *moveTo* method

This *void* method allows an *Actor* to be moved to a different *Location*. If another *Actor* occupies that *Location*, it is removed. There are two preconditions: the *Actor* must already be in a *Grid*, and the *Location* must be valid.

Method signature:

```
void moveTo(Location loc)
```

Example:

```
Grid<Actor> grd = theActor.getGrid( );
Location loc = new Location(22,19);
if( grd.isValid(loc) )
        theActor.moveTo(loc);
```

******** Controlling the action of an *Actor* ********

The *act* method

Recall the *Runner* graphical interfaces discussed in Chapters 1 and 2. This *void* method determines the action taken each time the *Step* button is clicked there (and also on each iteration of *Run*). All this method does is to reverse the direction of the *Actor*. It is in inherited classes like *Bug*, *Flower*, and *Rock* where this *act* method is overridden and real, meaningful actions are implemented. See the next section in which extending classes are discussed.

Method signature:

```
void act( )
```

Example:

```
act( );
```

Exercise A, Chapter 5

1. When an *Actor* object is instantiated, what color and direction are automatically assigned by the constructor?

2. What would be wrong with using the *put* method of the *Grid* class to place an *Actor* object in a *Grid*?

3. What properties does an *Actor* object have?

4. What is returned if *getGrid* is called and the *Actor* object is contained in multiple grids? (This is a trick question.)

5. What action is taken in the *act* method of the *Actor* class?

6. What are three classes that inherit the *Actor* class?

7. Show code that will move *Actor actr* to *Location* (3, 15) only if that *Location* does not contain a *Rock* object. Assume that *actr* is already in a *Grid*.

8. After an *Actor* is removed from a *Grid* with *removeSelfFromGrid*, what will *getLocation* return?

9. Is it possible to place an *Actor* at a *Grid* location already containing another *Actor*? If not, will this produce an exception? If possible, what would be the result?

10. Suppose we want to remove an *Actor* from a *Grid*. Knowing that a precondition for doing so is for the *Actor* to already be in a *Grid*, demonstrate a safe way to remove *Rock rk*.

******** Extending the *Actor* class ********

The *Bug*, *Flower*, and *Rock* classes all *extend* the *Actor* class. Each of these classes overrides the *act* method, thus causing each to behave differently.

The *act* method in the *Bug* class

The *act* method causes the *Bug* to move forward in its "direction" setting and to leave a *Flower* behind of the same color as the *Bug*. If a *Bug* encounters another *Bug*, *Rock*, or the edge of the *Grid* so that a forward move would be off the *Grid*, a 45 degree clockwise turn is done. If a move forward would encounter a *Flower*, the *Flower* is removed and the move is executed.

The overriding *act* code of the *Bug* class is as follows:

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

Notice this method depends on three other methods of the *Bug* class: *canMove*, *move*, and *turn*. These methods are fully described in <u>Appendix D</u>.

The *act* method in the *Rock* class

This *act* method does absolutely nothing. The code body is empty.

The *act* method in the *Flower* class

All the *act* method does is to darken its color.

Exercise B, Chapter 5

Appendix D will need to be consulted since the following questions concern code there.

1. Which line of code in the *canMove* method of the *Bug* class in Appendix D is most responsible for not allowing a move into a cell that contains a *Rock* or another *Bug*.

2. In the *canMove* method of the *Bug* class in Appendix D, what is the significance of *Location loc* given by?

   Location loc = getLocation( );

3. In which method of the *Bug* class in Appendix D is a *Flower* removed when moving a object to the *Location* occupied by the *Flower*?

4. Show that portion of the code in the *canMove* method of the *Bug* class in Appendix D used to determine the object occupying the position to which we wish to move.

5. What portion of the code in the *canMove* method of the *Bug* class in Appendix D is responsible for not allowing a move to an invalid *Location*?

6. In the *move* method of the *Bug* class in Appendix D, what happens if the *move* method is called **without** previously "getting permission" by calling *canMove*, and the resulting move would be to an invalid *Location* on the *Grid*?

7. For a *Bug* originally pointed north, how many times must the *turn* method be executed in order for the *Bug* to be pointed West?

8. What code evidence could you offer that the color of a *Flower* dropped by a *Bug* is the same color as the *Bug*?

9. When a *Bug* moves to a new cell it leaves behind a *Flower* in the cell it is vacating. Which method of the *Bug* class in Appendix D accomplishes this, and with what code?

10. Modify the code in the *move* method of the *Bug* class in Appendix D, eliminating the line: *Location loc = getLocation( );* Rewrite the remaining two lines of code that used *loc* by calling *getLocation* instead.

11. What does the *act* method of the *Rock* class do?

12. If a *Bug* removes itself from the *Grid* in its *move* method, does it still leave a *Flower* behind?

13. What does the *act* method of the *Flower* class do?

Project… *BugBeGone*

Create a *BugBeGone* class by extending the *Bug* class. Modify the *act* method by removing the *canMove* test. Do not react in any way to what lies in front of the *Bug*: just let it move forward in all cases.

Provide a *BugBeGoneRunner* class, and place a *Rock* in the path of the *Bug* so an observation can be made when the *Bug* encounters the *Rock*. Also, observe what happens when the *Bug* gets to the edge of the *Grid*. Write down these two observations.

Project… *JumpingBug*

Extend the *Bug* class to create a *JumpingBug* class that observes the following rules:

- When a *Bug* encounters another *Bug* or a *Rock*, it jumps over that obstacle in the direction in which the *Bug* is headed and into the next cell immediately on the other side of the obstacle.
- If no obstacle is encountered, just move forward one cell. If a *Flower* is encountered, remove it and let the *Bug* move to that position.
- If a jump over an obstacle would encounter another *Bug* or a *Rock*, then no move is executed; rather, a clockwise turn of 45 degrees is made.
- If a jump over an obstacle would carry the *Bug* off the *Grid*, don't jump. Rather, turn the *Bug* 135 degrees clockwise.
- Any other move that would take the *Bug* off the *Grid* should result in the *Bug* turned 225 degrees to the right.

JumpingBug will have two state variables (also called instance fields):

- *int numTurnDegrees* … this specifies how many degrees to turn clockwise in case *canMove* returns a *false*. This variable is set within the *canMove* method.

- *Location moveHere* … this specifies the *Location* to which we should move. This variable is set within the *canMove* method and is then used in the *move* method.

Override three methods as follows:

- Override the *canMove* method so as to implement the above rules and set the two state variables. You may want to use *instanceof* in your code (it's not part of the AP CS Java subset). See page 35-5 in the Blue Pelican Java textbook for an explanation of *instanceof*.

- Override the *move* method so as to make use of the *moveHere* variable.

- Override the *turn* method so as to turn *numTurnDegrees* degrees.

For testing, provide a *JumpingBugRunner* class in which a *JumpingBug* object is placed at (7, 8) and three *Rock* objects placed at (4, 8), (3, 5), and (4, 4).

The result of testing the *JumpingBugRunner* should look something like the following:

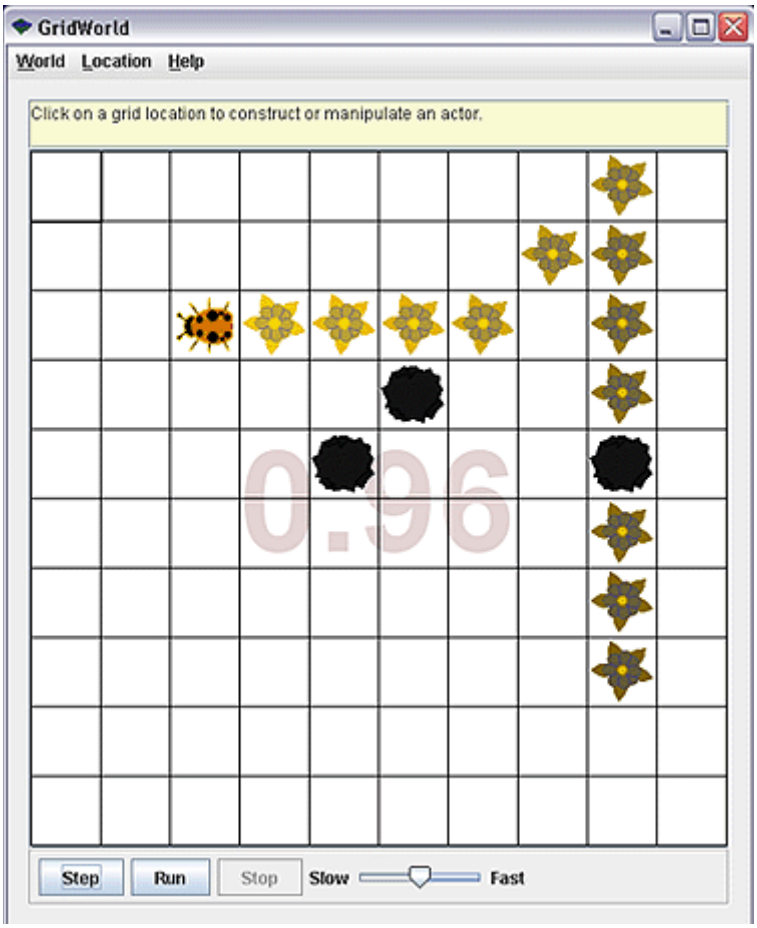

Fig 5-1. Testing the *JumpingBug* class

Chapter 6… The *Critter* Class

The *Critter* class extends the *Actor* class; however, it does **not** behave like a *Rock* or *Bug*. Here are the five fundamental things a *Critter* does via its *act* method.

1. Using the *getActors* method, a list of *Actor*s is gathered. These *Actors* are often in the immediate neighborhood.

2. It then processes these *Actor*s (causing them to do something) with the *processActors* method.

3. It generates a set of *Location*s (again, usually in its vicinity) to which it may later move. This is accomplished with the *getMoveLocations* method.

4. It selects one of these *Location*s with the *selectMoveLocation* method.

5. It moves the *Critter* to this new *Location* with the *makeMove* method.

As can be seen, there is considerable flexibility in the creation of a new class by extending *Critter*. For example, we have the freedom in determining:

- which *Actor*s to choose with *getActors*,
- what actions are taken as we process those *Actor*s with *processActors*,
- which *Location*s to choose with *getMoveLocations*,
- which one to eventually choose with *selectMoveLocation*,
- and the details of the actual move with *makeMove*.

It is in the subclasses of *Critter* that these methods are overridden.

**** The *act* method of *Critter* uses all five of these methods as follows ****

```
public void act( )
{
        if( getGrid( ) = = null )
                return;
        ArrayList<Actor> actors = getActors( );
        processActors(actors);
        ArrayList<Location> moveLocs = getMoveLocations( );
        Location loc = selectMoveLocation(moveLocs);
        makeMove(loc);
}
```

**** The five "overrideable" methods of the *Critter* class ****

It is recommended to **not** override the *act* method in classes that *extend Critter*. This is because any needed changes can be made by **overriding the methods used by *act***. The **default** action of those five methods is described below:

The *getActors* method

This method returns an *ArrayList* object containing the neighboring objects. It uses the *getNeighbors* method of the *Grid* class that returns all objects in the occupied *Locations* adjacent to a specified *Location*.

Method signature:

ArrayList<Actor> getActors( )

The *processActors* method

This method would typically receive the *ArrayList* created by *getActors*. It then proceeds to eat (i.e., to remove) all *Actor*s in that list that are **not** *Rock*s or *Critter*s.

Method signature:

public void processActors(ArrayList<Actor> actors)

The *getMoveLocations* method

This method uses the *getEmptyAdjacentLocations* of the *Grid* class to obtain and return an *ArrayList* of *Locations* that contain no objects. This is the list from which a *Location* will be selected for an eventual move.

Method signature:

ArrayList<Location> getMoveLocation( )

The *selectMoveLocation* method

This method ***randomly*** selects the *Location* to which this *Critter* will be later moved with *makeMove* and returns that *Location*.

Method signature:

Location selectMoveLocation(ArrayList<Location> locs)

The *makeMove* method

This method receives as a parameter the *Location* to which it is to move. The *Critter* object first has its direction set toward the *Location* to which it is to move, then the move is made.

Method signature:

void makeMove(Location loc)

For these five "overridable" methods, it should be noted that a **postcondition** of each is that the state of all *Actor*s is unchanged. An exception is the *makeMove* method where, of course, the *Location* property of the *Critter* would be changed. Another exception is that the direction of the *Actor* is set toward the new *Location*. When extending the *Critter* class, make sure that the new methods maintain all these post conditions.

Exercise, Chapter 6

1. What is the largest number of objects possible in the *ArrayList* object returned by *getActors*?

2. What are the five other methods called within the *act* method of the *Critter* class?

3. What type of objects are contained in the *ArrayList* object returned by *getMoveLocations*?

4. What type objects are **not** eaten by the *processActors* method?

5. When creating your own class and overriding the *processActors* method, must you necessarily make it "eat" something?

6. *Critter* has no constructor. What happens in the absence of a constructor when a new *Critter* object is instantiated?

7. Is it possible for *getEmptyAdjacentLocations* to return an invalid *Location* in its list?

8. Why is the *Location* selected by *selectMoveLocation* not predictable?

9. What is returned by *selectMoveLocation* if there are no *Locations* in the *ArrayList* that it receives?

10. Suppose the *ArrayList* of possible move *Location*s (from which an eventual *Location* is selected) is obtained in the *Critter* class using the *getMoveLocations*. Is the following code in the *makeMove* class really necessary? Why?

```
if(loc = = null)
        removeSelfFromGrid( );
```

11. In the *selectMoveLocation* method, why is it necessary to cast (Math.random( ) * n) as an *int* type?

12. Suppose you create a new class by extending the *Critter* class. Some of the methods that you might override in this new class make use of *ArrayList*. If you override one of those methods and make use of *ArrayList*, will it be necessary to do an import? If so, what should you import?

Chapter 7… Extending the *Critter* Class

The *Critter* class can be extended to produce *Critter*s with different characteristics. This is done by overriding methods of the *Critter* superclass and supplying code in the new subclasses to produce the necessary changes. Remember, as we produce these new *Critter* subclasses that **we do not override *act***. It is **the five methods** used in *act* that will be overridden with new code.

Project *ChameleonCritter*… changing color (**testable on the AP CS, A & AB exams**)

Create a class called *ChameleonCritter* by extending the *Critter* class. Behavior is exactly the same as in *Critter* with the exception of two overridden methods as follows:

- When processing *Actor*s, randomly select one and change the *ChameleonCritter*'s color to the selected *Actor*'s color. Do this by overriding the *processActors* method.
- If there are no neighboring *Actor*s, take no action with regard to a color change.
- Change the moved *ChamelonCritter*'s direction to the direction given by: "from its original *Location* toward its new *Location*". Do this by overriding the *makeMove* method.

Provide a *ChamelonCritterRunner* for testing and create an orange *ChamelonCritter* at (5, 6). Place a red *Rock* at (4, 4), a green *Rock* at (4, 8), and a blue *Rock* at (3, 5). Use the *Step* button to see the *Critter* randomly move and observe it obey its rules. Before the stepping process begins, the arrangement of the *ChameleonCritter* and the *Rock*s is as shown in Fig 7-1 below.

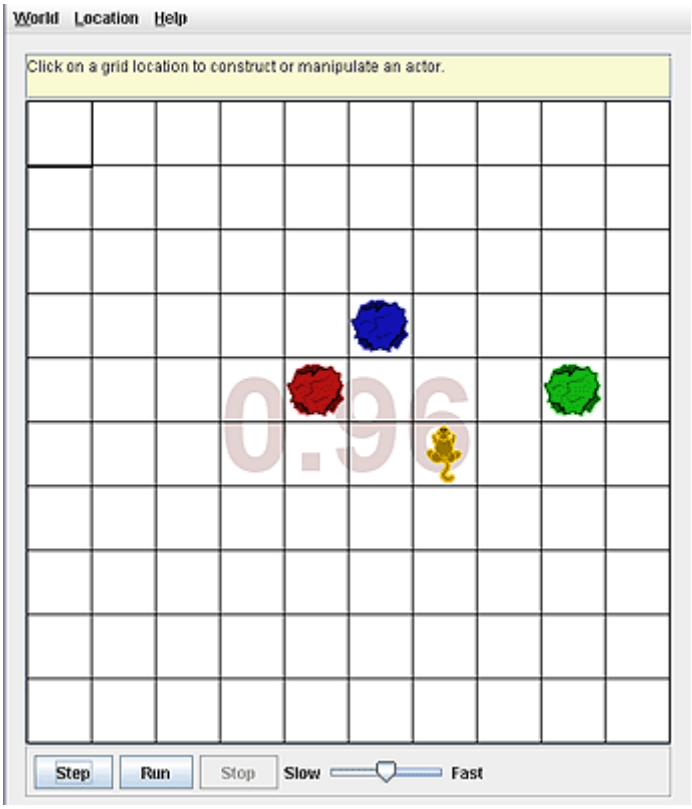

Fig 7-1. Relative positioning of a *ChameleonCritter* and three colored *Rock*s.

Exercise A, Chapter 7 (*ChameleonCritter*)

In answering the following questions, look at the code provided for the *ChameleonCritter* project near the end of this chapter:

1. How would the code in *makeMove* be modified so as to make the new direction exactly the opposite from that presently set in this method of *ChameleonCritter*?

2. Why isn't the *getActors* method overridden in *ChameleonCritter*?

3. In the *processActors* method, the following line of code is found:

   ```
   int r = (int) (Math.random( ) * n);
   ```

   Would it be permissible to leave off the parenthesis surrounding all of *Math.random( ) * n*? Why?

4. Even though the *act* method of the *Critter* superclass is not overridden in *ChameleonCritter*, it does behave differently. Why?

5. Show code for making a *ChameleonCritter* object access its own *Location* and store the result in a variable called *myLoc*.

6. What modifications are needed to make *ChameleonCritter* drop a flower in the position it vacates after making a move? Be sure to guard against replacing the *ChameleonCritter* object with a *Flower* in case no move is made.

7. Suppose instead of calling *super.makeMove(loc)* in the *makeMove* method of *ChameleonCritter*, *makeMove* was called. What would be the result?

8. The method *getGrid* can be legally called within any overriding method of a subclass of *Critter* even though the subclass has no *getGrid* method. Why is this legal?

9. Create a method called *darken* that is to be within *ChameleonCritter* and that darkens the color of the displayed object to 90% of its original intensity.

   Hints:

   The *Color* constructor is overloaded and also has a version with the following signature:

   ```
   Color(int red, int green, int blue)
   ```

   This constructor uses the standard RGB color model where the maximum values for red, green and blue are 255 (maximum intensity). The minimum values are 0 (zero intensity).

   You should also be aware that once a *Color* object has been retrieved for a *Critter* (for example, with *Color c = getColor( );*), it is possible to

obtain the individual color component values (integers) with *c.getRed( )*, *c.getGreen( )*, and *c.getBlue( )*.

10. Create a class called *ChameleonJunior* that extends *ChameleonCritter*. Have it change color to the color of one of the randomly selected *Actor*s directly in **front** or **behind**. If no *Actor* is present in either of these *Location*s, darken the displayed *ChameleonJunior* using the *darken* method developed in #9.

Project *CrabCritter*… moving sideways (**not** testable on the AP test)

This *Critter* is considerably more complex than was the *ChameleonCritter*. In addition to its inherited characteristics from the *Critter* class, it follows these additional rules:

- A *CrabCritter* eats whatever lies immediately to the **left-front**, **right-front**, and in **front** of it. Notice, it will not be necessary to modify *processActors*. All that needs to be done is to send the superclass version of *processActors* an *ArrayList* containing the appropriate *Actor*s.

- As with *Critter*, it will not eat a *Rock* or another *Critter*. Again, this is exactly the behavior of the *processActors* method within the *Critter* superclass.

- It can only move to the **left** or **right** of its current direction into an **empty** cell.
  - If both *Location*s are empty, it randomly selects one.

  - If neither *Location* is available, it randomly turns 90 degrees left or right.

Create a *CrabCritterRunner* class for testing. In it, create an orange *CrabCritter* at (4, 5), a green *Rock* at (3, 6), a blue *Bug* at (3, 5), and a red *Rock* at (4, 4). This will result in the arrangement as shown below in Fig 7-2.

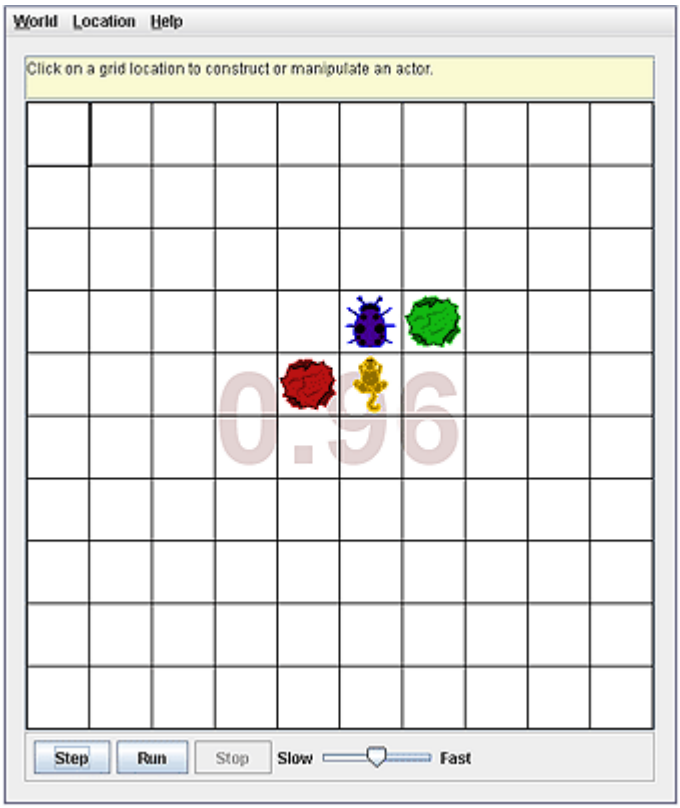

Fig 7-2. Relative initial positioning of a *CrabCritter*, two colored *Rock*s and a *Bug*.

Exercise B, Chapter 7 (*CrabCritter*)

1. What code should be added to the *CrabCritter* class so that it will construct a green *CrabCritter*?

2. How does *CrabCritter* manage without having its own *selectMoveLocation* method?

3. Under what circumstances does a *CrabCritter* turn rather than move?

4. Study the following line of code in *makeMove*. What are the possible integer values it can generate?

```
int r = (int)( 2 * Math.random( ) );
```

5. Why doesn't *CrabCritter* eat *Rock*s?

6. What code could be added to *CrabCritter* so that behavior is exactly as before except that *Rock*s are also eaten?

7. Looking at Fig 7-2, it would appear that from the initial position of the *CrabCritter* that it would eat the *Bug* immediately in front of it. After clicking *Step*, it is observed to **not** have eaten the *Bug*. Assuming there are no errors in the code, what possible explanation could there be for this apparent violation of the rules?

8. Run *CrabCritterRunner* several times using the *Step* button to manually increment through the steps. Each time the sequence of moves of the *CrabCritter* will be observed to move through the **very same** sequence of steps. Why is this since the moves are randomly selected?

9. If a *CrabCritter* at Location (5, 9) faces southwest and it has no adjacent neighbors, what are the possible move *Locations*?

Chapter 8… *Grid* Data Structures

In this chapter we will examine three specific *Grid* classes: *AbstractGrid*, *BoundedGrid*, and *UnboundedGrid*. All inherit the *Grid* interface (some indirectly via a "chain" of inheritance).

****** AbstractGrid ******

The two *Grid* classes that are ultimately used in GridWorld are ***BoundedGrid*** and ***UnboundedGrid***; however, they do not directly extend the *Grid* interface. Rather they both extend the *AbstractGrid* class which, in turn, implements the *Grid* interface.

So, why do we have *AbstractGrid* in the middle of the inheritance chain? Very simply, it is for the purpose of **not having repeated code** in *BoundedGrid* and *UnboundedGrid*. The signatures of these common methods in *AbstractGrid* that would otherwise be repeated are:

- public ArrayList<E> getNeighbors(Location loc)
- public ArrayList<Location> getValidAdjacentLocations(Location loc)
- public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
- public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
- public String toString( )

The code for these five methods is given in Appendix F. There, it will be observed the class signature for *AbstractGrid* is as follows:

public **abstract** class AbstractGrid<E> implements Grid<E>

Notice this class is *abstract* because it extends the *Grid* interface, yet it does not implement all of the methods specified in *Grid*. It is left up to the classes (*BoundedGrid* and *UnboundedGrid*) that extend *AbstractGrid* to finally implement those remaining methods.

Exercise A… Chapter 8 (*AbstractGrid*)

To answer some of the following questions, you will need to consult the source code for *AbstractGrid* in Appendix F.

1. Why is the *AbstractGrid* class *abstract*?
2. In the *getValidAdjacentLocations* method, the following expression is found:

   Location.FULL_CIRCLE / Location.HALF_RIGHT

   Rewrite the entire line of code containing this expression, replacing the above with its equivalent integer value.

3. In the *getValidAdjacentLocations* method, the *isValid* method is invoked; however, this method is not in the *AbstractGrid* class. Why is this legal and where is *isValid* actually implemented?

4. Rewrite *getValidAdjacentLocations* so that it will only return *Locations* that are immediately north, south, east, and west of *loc*.

5. The *isValid* method is asking for the validity of what (a *Location*, an *Actor*, a *Grid*, etc….)?

6. What is the purpose of *AbstractGrid* since *UnboundedGrid* and *BoundedGrid* are in the chain of inheritance and would be available to implement code?

7. What is the method that can return an object at a particular *Location*? Show how to perform a test with this method to determine if the *Location* is empty. Which method of *AbstractGrid* uses these techniques to fill its returned *ArrayList*?

8. Show the inheritance chain for *BoundedGrid*.

9. Describe what *getNeighbors* returns.

10. Look at the following line of code in the *toString* method. How does something meaningful get printed for the two items in bold?

    s += **loc** + "=" + **get(loc)**;

****** *BoundedGrid* ******

A *BoundedGrid* is actually implemented with a *two dimensional array*, and has a fixed number of rows and columns. Any direct attempt to access cells outside these fixed boundaries will result in an exception.

The code for *BoundedGrid* is given in Appendix F. The class signature there uses the generic type *E*; however, the following creation of its only instance field is **not** done as might be expected:

```
private E[][] occupantArray;
```

Rather, it is declared in this way:

```
private Object[][] occupantArray;
```

The reason for this is that an array **cannot** be declared as a generic type. Thus, it would appear that the "type safety" is comprised as is generally guaranteed by generics. However, type safety is still assured since the *put* method of this class is the only way to add an element to the *Grid* as can be see in its signature, and only objects of type *E* can be added.

Exercise B… Chapter 8 (*BoundedGrid*)

1. Why is the generic type *E* not used in the declaration of the instance field, *occupantArray*?

2. What is the underlying data structure of *BoundedGrid*?

3. What restrictions does the constructor of *BoundedGrid* place on the instantiation of *BoundedGrid* objects.

4. The method *getNumCols* returns *occupantArray[0].length* . Could a subscript of *1* have worked equally well in all cases?

5. Which of the following is the mathematical equivalent of what is returned by the *isValid* method where *r* is the row number and *c* is the column number being tested (assume that the dimensions of the *Grid* are *numOfRows* X *numOfCols*)?

    a. $0 < r < numOfRows - 1$ and $0 < c < numOfCols - 1$
    b. $0 < r < numOfRows$ and $0 < c < numOfCols$
    c. $0 < numOfRows < r - 1$ and $0 < numOfCols < c - 1$
    d. $0 < numOfRows < r$ and $0 < numOfCols < c$

6. What are the time complexity (Big O) values for the following methods?
    a. get
    b. getOccupiedLocations
    c. put
    d. remove

7. What is returned by the *put* method?

8. What is returned by *put* if nothing presently occupies the specified *Location*? What line of code is responsible for this?

9. What might possibly cause *put* to throw and exception?

10. Does a *BoundedGrid* require any memory usage for “unused” cells? Why?

11. Could a *BoundedGrid* be created in which the *Location* given by (-22, 0) be legal? What about (0,0)?

******* Unbounded Grid *******

Unlike the *BoundedGrid*, the *UnboundedGrid* does not use a fixed sized two-dimensional array. Therefore, all *Location* coordinates are valid, even negative or extremely large values. Occupants are stored in a *Map<Location, E>* object with the following properties:

- The **keys** are the *Location*s objects
- The **values** are occupants of type *E* (normally *Actors*)

Naturally, since there are no fixed bounds, the methods *numRows* and *numCols* both return -1 to indicate that these are both meaningless quantities for the *UnboundedGrid*. Also, since all *Location*s are valid, *isValid* always returns a *true*. The following methods are all implemented by invoking the corresponding methods of the *Map* class.

- *get*
- *put*
- *remove*

Finally, the *getOccupiedLocations* method returns the same *Location*s as does the *keySet* method of the *Map* class as follows:

```
ArrayList<Location> a = new ArrayList( );
for( Location loc: occupantMap.keySet( ) )
        a.add(loc);
return a;
```

Exercise C… Chapter 8 (*UnbounbdedGrid*)

1. What type of loop is used in the implementing code for *getOccupiedLocations*?

2. What methods of the *UnboundedGrid* use corresponding methods of the very same name within the *Map* class.

3. Which method of the *UnboundedGrid* uses the *keySet* method of the *Map* class?

4. In the code for *getOccupiedLocations*, why isn't *get* used to verify that a *Location* is occupied by an object as is done in *BoundedGrid*.

5. Is the *Location* given by (-3, 0) valid in an *UnboundedGrid*? What about (5.3, 6)?

6. Notice in the constructor for *UnboundedGrid* that a *HashMap* is used. Since *Location*s are the keys, what method(s) does the *Location* class have that are used as a consequence of the data structure being a *HashMap*?

7. Suppose the constructor for *UnboundedGrid* used a *TreeMap* instead. Since *Location*s are the keys, what method(s) does the *Location* class have that are used as a consequence of the data structure being a *TreeMap*?

8. What will be the difference in the *ArrayList* returned by *getOccupiedLocations* from a *UnbounderGrid* with an underlying *HashMap* as compared to one with an underlying *TreeMap*?

9. What is the Big O value for *get, put, and remove* for a *HashMap* implementation of *UnboundedGrid*?

10. What is the Big O value for *get, put, and remove* for a *TreeMap* implementation of *UnboundedGrid*?

11. Why are *get*, *put*, and *remove* tested with *if(loc= = null)* rather than with *isValid*?

Appendices A-H

The source code shown in these appendices is exactly the same as that supplied by the College Board for Computer Science AP.

Appendix A… *Location* Class

`info.gridworld.grid.Location` class (`implements Comparable`)

`public Location(int r, int c)`
constructs a location with given row and column coordinates

`public int getRow()`
returns the row of this location

`public int getCol()`
returns the column of this location

`public Location getAdjacentLocation(int direction)`
returns the adjacent location in the direction that is closest to `direction`

`public int getDirectionToward(Location target)`
returns the closest compass direction from this location toward `target`

`public boolean equals(Object other)`
returns `true` if `other` is a `Location` with the same row and column as this location; `false` otherwise

`public int hashCode()`
returns a hash code for this location

`public int compareTo(Object other)`
returns a negative integer if this location is less than `other`, zero if the two locations are equal, or a positive integer if this location is greater than `other`. Locations are ordered in row-major order.
Precondition: `other` is a `Location` object.

`public String toString()`
returns a string with the row and column of this location, in the format (row, col)

Compass directions:

```
public static final int NORTH = 0;
public static final int EAST = 90;
public static final int SOUTH = 180;
public static final int WEST = 270;
public static final int NORTHEAST = 45;
public static final int SOUTHEAST = 135;
public static final int SOUTHWEST = 225;
public static final int NORTHWEST = 315;
```

Turn angles:

```
public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;
```

Appendix B… *Grid* Interface

`info.gridworld.grid.Grid<E>` interface

`int getNumRows()`
: returns the number of rows, or -1 if this grid is unbounded

`int getNumCols()`
: returns the number of columns, or -1 if this grid is unbounded

`boolean isValid(Location loc)`
: returns `true` if `loc` is valid in this grid, `false` otherwise
Precondition: `loc` is not `null`

`E put(Location loc, E obj)`
: puts `obj` at location `loc` in this grid and returns the object previously at that location (or `null` if the location was previously unoccupied).
Precondition: (1) `loc` is valid in this grid (2) `obj` is not `null`

`E remove(Location loc)`
: removes the object at location `loc` from this grid and returns the object that was removed (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`E get(Location loc)`
: returns the object at location `loc` (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedLocations()`
: returns an array list of all occupied locations in this grid

`ArrayList<Location> getValidAdjacentLocations(Location loc)`
: returns an array list of the valid locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getEmptyAdjacentLocations(Location loc)`
: returns an array list of the valid empty locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedAdjacentLocations(Location loc)`
: returns an array list of the valid occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<E> getNeighbors(Location loc)`
: returns an array list of the objects in the occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

Appendix C… *Actor, Rock, Flower*

`info.gridworld.actor.Actor` class

`public Actor()`
constructs a blue actor that is facing north

`public Color getColor()`
returns the color of this actor

`public void setColor(Color newColor)`
sets the color of this actor to `newColor`

`public int getDirection()`
returns the direction of this actor, an angle between 0 and 359 degrees

`public void setDirection(int newDirection)`
sets the direction of this actor to the angle between 0 and 359 degrees that is equivalent to `newDirection`

`public Grid<Actor> getGrid()`
returns the grid of this actor, or `null` if this actor is not contained in a grid

`public Location getLocation()`
returns the location of this actor, or `null` if this actor is not contained in a grid
Precondition: this actor is contained in a grid

`public void putSelfInGrid(Grid<Actor> gr, Location loc)`
puts this actor into location `loc` of grid `gr`. If there is another actor at `loc`, it is removed.
Precondition: (1) This actor is not contained in a grid (2) `loc` is valid in `gr`

`public void removeSelfFromGrid()`
removes this actor from its grid.
Precondition: this actor is contained in a grid

`public void moveTo(Location newLocation)`
moves this actor to `newLocation`. If there is another actor at `newLocation`, it is removed.
Precondition: (1) This actor is contained in a grid (2) `newLocation` is valid in the grid of this actor

`public void act()`
reverses the direction of this actor. Override this method in subclasses of `Actor` to define types of actors with different behavior

`public String toString()`
returns a string with the location, direction, and color of this actor

`info.gridworld.actor.Rock` class (`extends Actor`)

`public Rock()`
constructs a black rock

`public Rock(Color rockColor)`
constructs a rock with color `rockColor`

`public void act()`
overrides the `act` method in the `Actor` class to do nothing

`info.gridworld.actor.Flower` class (`extends Actor`)

`public Flower()`
constructs a pink flower

`public Flower(Color initialColor)`
constructs a flower with color `initialColor`

`public void act()`
causes the color of this flower to darken

Appendix D… *Bug, BoxBug*

Bug.java

```
package info.gridworld.actor;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
/**
* A Bug is an actor that can move and turn. It drops flowers as it moves.
* The implementation of this class is testable on the AP CS A and AB Exams.
*/
public class Bug extends Actor
{
	/**
	* Constructs a red bug.
	*/
	public Bug()
	{
		setColor(Color.RED);
	}


	/**
	* Constructs a bug of a given color.
	* @param bugColor the color for this bug
	*/
	public Bug(Color bugColor)
	{
		setColor(bugColor);
	}


	/**
	* Moves if it can move, turns otherwise.
	*/
	public void act()
	{
		if (canMove())
			move();
		else
			turn();
	}


	/**
	* Turns the bug 45 degrees to the right without changing its location.
	*/
	public void turn()
	{
		setDirection(getDirection() + Location.HALF_RIGHT);
	}


	/**
	* Moves the bug forward, putting a flower into the location it previously occupied.
	*/
```

```
	public void move()
	{
		Grid<Actor> gr = getGrid();
		if (gr == null)
			return;
		Location loc = getLocation();

		Location next = loc.getAdjacentLocation(getDirection());
		if (gr.isValid(next))
			moveTo(next);
		else
			removeSelfFromGrid();
		Flower flower = new Flower(getColor());
		flower.putSelfInGrid(gr, loc);
	}


	/**
	* Tests whether this bug can move forward into a location that is empty or contains a flower.
	* @return true if this bug can move.
	*/
	public boolean canMove()
	{
		Grid<Actor> gr = getGrid();
		if (gr == null)
			return false;
		Location loc = getLocation();
		Location next = loc.getAdjacentLocation(getDirection());
		if (!gr.isValid(next))
			return false;
		Actor neighbor = gr.get(next);
		return (neighbor == null) || (neighbor instanceof Flower);
		// ok to move into empty location or onto flower
		// not ok to move onto any other actor
	}
}
```

BoxBug.java

```
import info.gridworld.actor.Bug;
/**
* A BoxBug traces out a square “box” of a given size.
* The implementation of this class is testable on the AP CS A and AB Exams.
*/
public class BoxBug extends Bug
{
	private int steps;
	private int sideLength;
	/**
	* Constructs a box bug that traces a square of a given side length
	* @param length the side length
	*/
	public BoxBug(int length)
	{
		steps = 0;
		sideLength = length;
	}

	/**
	* Moves to the next location of the square.
	*/
	public void act()
	{
		if (steps < sideLength && canMove())
		{
			move();
			steps++;
		}
		else
		{
			turn();
			turn();
			steps = 0;
		}
	}
}
```

Appendix E… *Critter, ChameleonCritter*

Critter.java

```
package info.gridworld.actor;
import info.gridworld.grid.Location;
import java.util.ArrayList;
/**
 * A Critter is an actor that moves through its world, processing
 * other actors in some way and then moving to a new location.
 * Define your own critters by extending this class and overriding any methods of this class except for act.
 * When you override these methods, be sure to preserve the postconditions.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Critter extends Actor
{
	/**
	 * A critter acts by getting a list of other actors, processing that list, getting locations to move to,
	 * selecting one of them, and moving to the selected location.
	 */
	public void act()
	{
		if (getGrid() == null)
			return;
		ArrayList<Actor> actors = getActors();
		processActors(actors);
		ArrayList<Location> moveLocs = getMoveLocations();
		Location loc = selectMoveLocation(moveLocs);
		makeMove(loc);
	}


	/**
	 * Gets the actors for processing. Implemented to return the actors that occupy neighboring grid locations.
	 * Override this method in subclasses to look elsewhere for actors to process.
	 * Postcondition: The state of all actors is unchanged.
	 * @return a list of actors that this critter wishes to process.
	 */
	public ArrayList<Actor> getActors()
	{
		return getGrid().getNeighbors(getLocation());
	}


	/**
	 * Processes the elements of actors. New actors may be added to empty locations.
	 * Implemented to "eat" (i.e., remove) selected actors that are not rocks or critters.
	 * Override this method in subclasses to process actors in a different way.
	 * Postcondition: (1) The state of all actors in the grid other than this critter and the
	 * elements of actors is unchanged. (2) The location of this critter is unchanged.
	 * @param actors the actors to be processed
	 */
	public void processActors(ArrayList<Actor> actors)
	{
		for (Actor a : actors)
		{
			if (!(a instanceof Rock) && !(a instanceof Critter))
				a.removeSelfFromGrid();
		}
	}

	/**
	 * Gets a list of possible locations for the next move. These locations must be valid in the grid of this
	critter.
	 * Implemented to return the empty neighboring locations. Override this method in subclasses to look
	 * elsewhere for move locations.
```

```
	* Postcondition: The state of all actors is unchanged.
	* @return a list of possible locations for the next move
	*/
	public ArrayList<Location> getMoveLocations()
	{
		return getGrid().getEmptyAdjacentLocations(getLocation());
	}


	/**
	* Selects the location for the next move. Implemented to randomly pick one of the possible locations,
	* or to return the current location if locs has size 0. Override this method in subclasses that
	* have another mechanism for selecting the next move location.
	* Postcondition: (1) The returned location is an element of locs, this critter's current location, or null.
	* (2) The state of all actors is unchanged.
	* @param locs the possible locations for the next move
	* @return the location that was selected for the next move.
	*/
	public Location selectMoveLocation(ArrayList<Location> locs)
	{
		int n = locs.size();
		if (n == 0)
			return getLocation();
		int r = (int) (Math.random() * n);
		return locs.get(r);
	}


	/**
	* Moves this critter to the given location loc, or removes this critter from its grid if loc is null.
	* An actor may be added to the old location. If there is a different actor at location loc, that actor is
	* removed from the grid. Override this method in subclasses that want to carry out other actions
	* (for example, turning this critter or adding an occupant in its previous location).
	* Postcondition: (1) getLocation() == loc.
	* (2) The state of all actors other than those at the old and new locations is unchanged.
	* @param loc the location to move to
	*/
	public void makeMove(Location loc)
	{
		if (loc == null)
			removeSelfFromGrid();
		else
			moveTo(loc);
	}
}
```

ChameleonCritter.java

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;
import java.util.ArrayList;
/**
* A ChameleonCritter takes on the color of neighboring actors as it moves through the grid.
* The implementation of this class is testable on the AP CS A and AB Exams.
*/
public class ChameleonCritter extends Critter
{
	/**
	* Randomly selects a neighbor and changes this critter's color to be the same as that neighbor's.
	* If there are no neighbors, no action is taken.
	*/
	public void processActors(ArrayList<Actor> actors)
	{
		int n = actors.size();
		if (n == 0)
			return;
		int r = (int) (Math.random() * n);
		Actor other = actors.get(r);
		setColor(other.getColor());
	}


	/**
	* Turns towards the new location as it moves.
	*/
	public void makeMove(Location loc)
	{
		setDirection(getLocation().getDirectionToward(loc));
		super.makeMove(loc);
	}
}
```

Appendix F… *Grid* Structures

AbstractGrid.java

```
package info.gridworld.grid;
import java.util.ArrayList;
/**
* AbstractGrid contains the methods that are common to grid implementations.
* The implementation of this class is testable on the AP CS AB Exam.
*/
public abstract class AbstractGrid<E> implements Grid<E>
{
	public ArrayList<E> getNeighbors(Location loc)
	{
		ArrayList<E> neighbors = new ArrayList<E>();
		for (Location neighborLoc : getOccupiedAdjacentLocations(loc))
			neighbors.add(get(neighborLoc));
		return neighbors;
	}


	public ArrayList<Location> getValidAdjacentLocations(Location loc)
	{
		ArrayList<Location> locs = new ArrayList<Location>();
		int d = Location.NORTH;
		for (int i = 0; i < Location.FULL_CIRCLE / Location.HALF_RIGHT;
		i++)
		{
			Location neighborLoc = loc.getAdjacentLocation(d);
			if (isValid(neighborLoc))
				locs.add(neighborLoc);
			d = d + Location.HALF_RIGHT;
		}
		return locs;
	}


	public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
	{
		ArrayList<Location> locs = new ArrayList<Location>();
		for (Location neighborLoc : getValidAdjacentLocations(loc))
		{
			if (get(neighborLoc) == null)
				locs.add(neighborLoc);
		}
		return locs;
	}



	public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
	{
		ArrayList<Location> locs = new ArrayList<Location>();
		for (Location neighborLoc : getValidAdjacentLocations(loc))
		{
			if (get(neighborLoc) != null)
				locs.add(neighborLoc);
		}
		return locs;
	}
```

```
	/**
	* Creates a string that describes this grid.
	* @return a string with descriptions of all objects in this grid (not
	* necessarily in any particular order), in the format {loc=obj, loc=obj, ...}
	*/
	public String toString()
	{
		String s = "{";
		for (Location loc : getOccupiedLocations())
		{
			if (s.length() > 1)
				s += ", ";
			s += loc + "=" + get(loc);
		}
		return s + "}";
	}
}
```

BoundedGrid.java

```
package info.gridworld.grid;
import java.util.ArrayList;
/**
* A BoundedGrid is a rectangular grid with a finite number of rows and columns.
* The implementation of this class is testable on the AP CS AB Exam.
*/
public class BoundedGrid<E> extends AbstractGrid<E>
{
	private Object[][] occupantArray; // the array storing the grid elements


	/**
	* Constructs an empty bounded grid with the given dimensions.
	* (Precondition: rows > 0 and cols > 0.)
	* @param rows number of rows in BoundedGrid
	* @param cols number of columns in BoundedGrid
	*/
	public BoundedGrid(int rows, int cols)
	{
		if (rows <= 0)
			throw new IllegalArgumentException("rows <= 0");
		if (cols <= 0)
			throw new IllegalArgumentException("cols <= 0");
		occupantArray = new Object[rows][cols];
	}


	public int getNumRows()
	{
		return occupantArray.length;
	}


	public int getNumCols()
	{
		// Note: according to the constructor precondition, numRows() > 0, so
		// theGrid[0] is non-null.
		return occupantArray[0].length;
	}
```

```
	public boolean isValid(Location loc)
	{
		return 0 <= loc.getRow() && loc.getRow() < getNumRows()
			&& 0 <= loc.getCol() && loc.getCol() < getNumCols();
	}


	public ArrayList<Location> getOccupiedLocations()
	{
		ArrayList<Location> theLocations = new ArrayList<Location>();
		// Look at all grid locations.
		for (int r = 0; r < getNumRows(); r++)
		{
			for (int c = 0; c < getNumCols(); c++)
			{
				// If there's an object at this location, put it in the array.
				Location loc = new Location(r, c);
				if (get(loc) != null)
					theLocations.add(loc);
			}
		}
		return theLocations;
	}


	public E get(Location loc)
	{
		if (!isValid(loc))
			throw new IllegalArgumentException("Location " + loc + " is
			not valid");
		return (E) occupantArray[loc.getRow()][loc.getCol()]; // unavoidable
                                                                      //warning
	}


	public E put(Location loc, E obj)
	{
		if (!isValid(loc))
			throw new IllegalArgumentException("Location " + loc+ " is
			not valid");
		if (obj == null)
			throw new NullPointerException("obj == null");
		// Add the object to the grid.
		E oldOccupant = get(loc);
		occupantArray[loc.getRow()][loc.getCol()] = obj;
		return oldOccupant;
	}


	public E remove(Location loc)
	{
		if (!isValid(loc))
			throw new IllegalArgumentException("Location " + loc + " is
			not valid");
		// Remove the object from the grid.
		E r = get(loc);
		occupantArray[loc.getRow()][loc.getCol()] = null;
		return r;
	}
}
```

UnboundedGrid.java

```
package info.gridworld.grid;
import java.util.ArrayList;
import java.util.*;

/**
* An UnboundedGrid is a rectangular grid with an unbounded number of rows and columns.
* The implementation of this class is testable on the AP CS AB Exam.
*/
public class UnboundedGrid<E> extends AbstractGrid<E>
{
      private Map<Location, E> occupantMap;


      /**
      * Constructs an empty unbounded grid.
      */
      public UnboundedGrid()
      {
            occupantMap = new HashMap<Location, E>();
      }


      public int getNumRows()
      {
            return -1;
      }


      public int getNumCols()
      {
            return -1;
      }


      public boolean isValid(Location loc)
      {
            return true;
      }


      public ArrayList<Location> getOccupiedLocations()
      {
            ArrayList<Location> a = new ArrayList<Location>();
            for (Location loc : occupantMap.keySet())
                  a.add(loc);
            return a;
      }


      public E get(Location loc)
      {
            if (loc == null)
                  throw new NullPointerException("loc == null");
            return occupantMap.get(loc);
      }


      public E put(Location loc, E obj)
      {
            if (loc == null)
                  throw new NullPointerException("loc == null");
            if (obj == null)
                  throw new NullPointerException("obj == null");
            return occupantMap.put(loc, obj);
      }
```

```
	public E remove(Location loc)
	{
		if (loc == null)
			throw new NullPointerException("loc == null");
		return occupantMap.remove(loc);
	}
}
```

Appendix G… Quick Reference, A/AB

Location Class (implements Comparable)

```
public Location(int r, int c)
public int getRow()
public int getCol()
public Location getAdjacentLocation(int direction)
public int getDirectionToward(Location target)
public boolean equals(Object other)
public int hashCode()
public int compareTo(Object other)
public String toString()
NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST
LEFT, RIGHT, HALF_LEFT, HALF_RIGHT, FULL_CIRCLE, HALF_CIRCLE, AHEAD
```

Grid<E> Interface

```
int getNumRows()
int getNumCols()
boolean isValid(Location loc)
E put(Location loc, E obj)
E remove(Location loc)
E get(Location loc)
ArrayList<Location> getOccupiedLocations()
ArrayList<Location> getValidAdjacentLocations(Location loc)
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
ArrayList<E> getNeighbors(Location loc)
```

Actor Class

```
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
```

Rock Class (extends Actor)

```
public Rock()
public Rock(Color rockColor)
public void act()
```

Flower Class (extends Actor)

```
public Flower()
public Flower(Color initialColor)
public void act()
```

Bug Class (extends Actor)

```
public Bug()
public Bug(Color bugColor)
public void act()
public void turn()
public void move()
public boolean canMove()
```

BoxBug Class (extends Bug)

```
public BoxBug(int n)
public void act()
```

Critter Class (extends Actor)

```
public void act()
public ArrayList<Actor> getActors()
public void processActors(ArrayList<Actor> actors)
public ArrayList<Location> getMoveLocations()
public Location selectMoveLocation(ArrayList<Location> locs)
public void makeMove(Location loc)
```

ChameleonCritter Class (extends Critter)

```
public void processActors(ArrayList<Actor> actors)
public void makeMove(Location loc)
```

Appendix H… Quick Reference, AB Only

AbstractGrid Class (implements Grid)

```
public ArrayList<E> getNeighbors(Location loc)
public ArrayList<Location> getValidAdjacentLocations(Location loc)
public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
public String toString()
```

BoundedGrid Class (extends AbstractGrid)

```
public BoundedGrid(int rows, int cols) / public UnboundedGrid()
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```

UnboundedGrid Class (extends AbstractGrid)

```
public UnboundedGrid()
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```