# Blue Pelican GridWorld

## Student Manual

**AP Computer Science**
**Case Study**

# Table of contents

# 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.

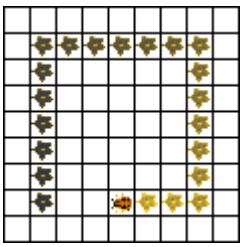

Fig 2-1. When testing the *BoxBug* class, the graphics should produce something like this for each *BoxBug* object on the Grid.

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.

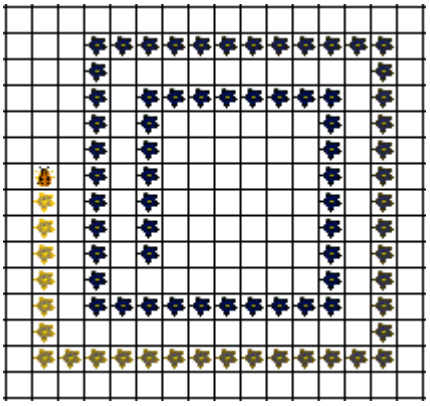

Fig 2-1. When testing the *SpiralBug* class, the graphics should produce something like this for each *SpiralBug* object on the grid

# 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)
```