GridWorld
AP® Computer Science
Case Study

Solutions Manual

The AP® Program wishes to acknowledge and to thank
Judith Hromcik of Arlington High School in Arlington, Texas.

Part 1

Part 1 Answers:

Do You Know?

Set 1

1. Does the bug always move to a new location? Explain.

   No. A bug will only move to the location in front of it if the cell exists and is empty or if there is a flower in the cell.

2. In which direction does the bug move?

   A bug attempts to move forward.

3. What does the bug do if it does not move?

   When a bug cannot move, it turns 45 degrees to the right.

4. What does a bug leave behind when it moves?

   A bug leaves a flower in its old cell when it moves to a new cell. The flower is the same color as the bug.

5. What happens when the bug is at an edge of the grid? (Consider whether the bug is facing the edge as well as whether the bug is facing some other direction when answering this question.)

   If a bug is facing the grid edge and it is told to act, it will turn 45 degrees to the right. When told to act again, it will turn another 45 degrees to the right.

   If a bug is facing the grid edge and it is told to move, it will remove itself from the grid and a flower will replace the bug in that location.

6. What happens when a bug has a rock in the location immediately in front of it?

   The bug turns 45 degrees to the right.

7. Does a flower move?

   No.

8. What behavior does a flower have?

   A flower “wilts” over time. The color of a flower gets darker until it turns a dark gray.

9. Does a rock move or have any other behavior?

   No. A rock stays in its location and does not appear to have any other behaviors when the step or run button is used.

10. Can more than one actor (bug, flower, rock) be in the same location in the grid at the same time?

    No. A location in the grid can contain only one actor at a time.

Exercises (Page 8)

1. Test the `setDirection` method with the following inputs and complete the table, giving the compass direction each input represents.

Degrees	Compass Direction
0	North
45	NorthEast
90	East
135	SouthEast
180	South
225	SouthWest
270	West
315	NorthWest
360	North

2. Move a bug to a different location using the `moveTo` method. In which directions can you move it? How far can you move it? What happens if you try to move the bug outside the grid?

   A bug can be moved to any valid location in the grid using the `moveTo` method. When moving the bug using the `moveTo` method, the bug will not change its original direction. The `setDirection` method or `turn` method must be used to change a bug's direction.

   Attempting to move a bug to a location outside of the grid (an invalid location for the grid) will cause an `IllegalArgumentException` to be thrown.

3. Change the color of a bug, a flower, and a rock. Which method did you use?

   The `setColor` method.

4. Move a rock on top of a bug and then move the rock again. What happened to the bug?

   When a rock is moved "on top" of a bug, the bug disappears. When the rock is moved to another location, the bug is no longer there, or anywhere else in the grid. When a new actor moves into a grid location occupied by another actor, the old actor is removed from the grid.

Do You Know?

Set 2

1. What is the role of the instance variable `sideLength`?

   The `sideLength` instance variable defines the number of steps a `BoxBug` moves on each side of its box.

2. What is the role of the instance variable `steps`?

   The `steps` instance variable keeps track of how many steps a `BoxBug` has moved on the current side of its box.

3. Why is the `turn` method called *twice* when `steps` becomes equal to `sideLength`?

   When a `BoxBug` travels `sideLength` steps, it has to turn 90 degrees to travel along the next side of its box. The `turn` method only executes a 45 degree turn; therefore it takes two `turn` method calls to turn 90 degrees.

4. Why can the `move` method be called in the `BoxBug` class when there is no `move` method in the `BoxBug` code?

   The `BoxBug` class extends the `Bug` class, and the `Bug` class has a public `move` method. Since the `BoxBug` class is a subclass of the `Bug` class, it inherits the `move` method from the `Bug` class.

5. After a `BoxBug` is constructed, will the size of its square pattern always be the same? Why or why not?

   Yes. When a `BoxBug` is constructed, the side length is determined and cannot be changed by client code.

6. Can the path a `BoxBug` travels ever change? Why or why not?

   Yes. If another `Actor`, like a `Rock` or `Bug`, is in front of a `BoxBug` when it tries to move, the `BoxBug` will turn and start a new box path.

7. When will the value of `steps` be zero?

   Initially, the value of `steps` is set to zero when a `BoxBug` is constructed. After that, the value of `steps` will be set to zero when `steps` is equal to `sideLength`—meaning the `BoxBug` has completed one side of its box path, or when the `BoxBug` cannot move and turns instead to start a new box path.

Part 2 Exercises

1. Write a class `CircleBug` that is identical to `BoxBug`, except that in the `act` method the `turn` method is called once instead of twice. How is its behavior different from a `BoxBug`?

```
import info.gridworld.actor.Bug;
public class CircleBug extends Bug
{
  private int steps;
  private int sideLength;

  public CircleBug(int n)
  {
    sideLength = n;
  }

  public void act()
  {
    if (steps < sideLength && canMove())
    {
      move();
      steps++;
    }
    else
    {
      turn();
      steps = 0;
    }
  }
}
```

The path of a `CircleBug` is an octagon, instead of a square.

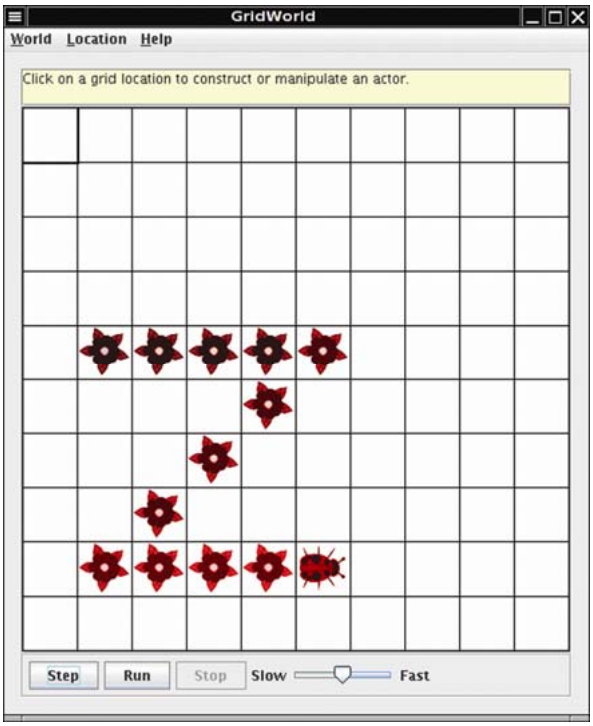
2. Write a class `SpiralBug` that drops flowers in a spiral pattern. Hint: Imitate `BoxBug`, but adjust the side length when the bug turns. You may want to change the world to an `UnboundedGrid` to see the spiral pattern more clearly.

```
import info.gridworld.actor.Bug;
public class SpiralBug extends Bug
{
  private int sideLength;
  private int steps;

  public SpiralBug(int n)
  {
    sideLength = n;
    steps = 0;
  }

  public void act()
  {
    if (steps < sideLength && canMove())
    {
      move();
      steps++;
    }
    else
    {
      turn();
      turn();
      steps = 0;
      //Each time a SpiralBug turns, increase the sideLength by one
      sideLength++;
    }
  }
}
```

The following `SpiralBugRunner` class can be used with the suggested `UnboundedGrid`.

```
import info.gridworld.actor.Actor;
import info.gridworld.grid.UnboundedGrid;
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;

public class SpiralBugRunner
{
  public static void main(String[] args)
  {
    UnboundedGrid grid = new UnboundedGrid<Actor>();
    ActorWorld world = new ActorWorld(grid);
    SpiralBug sp = new SpiralBug(3);
    world.add(new Location(3,5),sp);
    world.show();
  }
}
```

3. Write a class `ZBug` to implement bugs that move in a "Z" pattern, starting in the top left corner. After completing one "Z" pattern, a `ZBug` should stop moving. In any step, if a `ZBug` can't move and is still attempting to complete its "Z" pattern, the `ZBug` does not move and should not turn to start a new side. Supply the length of the "Z" as a parameter in the constructor. The following image shows a "Z" pattern of length 4. Hint: Notice that a `ZBug` needs to be facing east before beginning its "Z" pattern.




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

/**
 * A ZBug traces out a Z pattern of a given size.
 */

public class ZBug extends Bug
{
  private int segmentLength; // the number of flowers in each segment
  private int steps; // the number of steps in the current side
  private int segment; // which segment of the Z the ZBug is on
```

```
  /**
   * Constructs a Zbug that traces a Z pattern in which each segment
   * of the Z has the given length
   * When the Z is drawn, the ZBug stops.
   * @param length the segment length
   */
  public ZBug(int length)
  {
    setDirection(Location.EAST);
    steps = 0;
    segment = 1;
    segmentLength = length;
  }

  public void act()
  {
    if (segment <= 3 && steps < segmentLength)
    {
      if (canMove())
      {
        move();
        steps++;
      }
    }
    else if (segment == 1)
    {
      setDirection(Location.SOUTHWEST);
      steps = 0;
      segment++;
    }
    else if (segment == 2)
    {
      setDirection(Location.EAST);
      steps = 0;
      segment++;
    }
  }
}
```

4. Write a class `DancingBug` that "dances" by making different turns before each move. The `DancingBug` constructor has an integer array as parameter. The integer entries in the array represent how many times the bug turns before it moves. For example, an array entry of 5 represents a turn of 225 degrees (recall one turn is 45 degrees). When a dancing bug acts, it should turn the number of times given by the current array entry, then act like a `Bug`. In the next move, it should use the next entry in the array. After carrying out the last turn in the array, it should start again with the initial array value so that the dancing bug continually repeats the same turning pattern.

   The `DancingBugRunner` class should create an array and pass it as a parameter to the `DancingBug` constructor.

```
import info.gridworld.actor.Bug;
public class DancingBug extends Bug
{
  private int[] turnList;
  private int currentStep;

  public DancingBug(int[] turns)
  {
    turnList = turns;
    currentStep = 0;
  }

  public void turn(int times)
  {
    for(int j = 1; j <= times; j++)
    {
      turn();
    }
  }

  public void act()
  {
    if(currentStep == turnList.length)
    currentStep = 0;
    turn (turnList[currentStep]);
    currentStep++;
    super.act();
  }
}
```

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;
public class DancingBugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    int[] turns = {2,2,1,3};
    DancingBug ballerina = new DancingBug(turns);
    ballerina.setColor(Color.ORANGE);
    world.add(new Location(9, 9), ballerina);
    world.show();
  }
}
```

5. Study the code for the `BoxBugRunner` class. Summarize the steps you would use to add another `BoxBug` actor to the grid.

   1. Create a `BoxBug` object with the desired side length
      `BoxBug anotherOne = new BoxBug(2);`
   2. Add the new `BoxBug` to the word at a random or specific location
      `world.add(anotherOne);`
      or
      `world.add(new Location(1,1), anotherOne);`

Do You Know?
Set 3

Assume the following statements to answer the following questions.

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

1. How would you access the row value for `loc1`?
   `loc1.getRow()`

2. What is the value of `b` after the following statement is executed?

```
boolean b = loc1.equals(loc2);
```
   `false`

3. What is the value of `loc3` after the following statement is exectued?

```
Location loc3 = loc2.getAdjacentLocation(Location.SOUTH);
```
   `(4,4)`

4. What is the value of `dir` after the following statement is executed?

```
int dir = loc1.getDirectionToward(new Location(6,5));
```
   135 (degrees)—Southeast

5. How does the `getAdjacentLocation` method know which adjacent location to return?
   The parameter in the `getAdjacentLocation` method indicates the direction of the adjacent neighbor to find. It returns the adjacent location in the compass direction that is closest to the direction given in the parameter list.

Do You Know?
Set 4

1. How can you obtain a count of the objects in a grid? How can you obtain a count of the empty locations in a bounded grid?

   Assume that `gr` is a reference to a `Grid` object.

   `gr.getOccupiedLocations().size()` will find the number of occupied locations in any type of grid.

   `gr.getNumRows()*gr.getNumCols() - gr.getOccupiedLocations().size()` will find the number of empty locations in a `BoundedGrid` object.

2. How can you check if location (10,10) is in the grid?

   `gr.isValid(new Location(10,10))`

   The `Grid` method `isValid` returns true if and only if the given location is a valid location in the grid.

3. `Grid` contains method declarations, but no code is supplied in the methods. Why? Where can you find the implementations of these methods?

   `Grid` is an interface. In Java, an interface specifies which methods another class must implement. You can find the implementations of the methods in the `AbstractGrid` and the `BoundedGrid` and `UnboundedGrid` classes. Since the `AbstractGrid` only implements some of the required methods of the `Grid` interface, it is declared an `abstract` class. The `BoundedGrid` and `UnboundedGrid` extend the `AbstractGrid` class and implement the rest of the methods required by the `Grid` interface.

4. All methods that return multiple objects return them in an `ArrayList`. Do you think it would be a better design to return the objects in an array? Explain your answer.

   As a user of the `Grid` classes, perhaps. Accessing elements with the [] notation is somewhat easier than using different method calls. For example:

   `locs[j]` versus `locs.get(j)`

   In terms of implementing the methods, an `ArrayList` does not require the user to size the list before filling it. An array does. Since the `BoundedGrid` does not keep track of the number of objects in the grid, you would have to first count the number of occupied locations to size the array, and then go back through the grid to find and store each of the locations. If the `Grid` kept track of the number of occupied locations, filling an array would be just as easy as filling an `ArrayList`.

Do You Know?

Set 5

1. Name three properties of every actor.

   An actor has a color, a direction, and a location. It also has a reference to its grid.

2. When an actor is constructed, what is its direction and color?

   An actor's initial color is blue and its initial direction is `North`.

3. Why do you think that the `Actor` class was created as a class instead of an interface?

   An actor has both state and behavior. An interface does not allow the programmer to declare instance variables or implement methods.

4. (a) Can an actor put itself into a grid twice without removing itself? (b) Can an actor remove itself from a grid twice? (c) Can an actor be placed into a grid, remove itself, and then put itself back? Try it out. What happens?

(a) No—if the actor is already in the grid, it may not put itself in the grid again. This version of `BugRunner.java` will compile, but when it runs it will throw an `IllegalStateException.`

```
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    Bug b = new Bug();
    world.add(b);
    b.putSelfInGrid(b.getGrid(),b.getLocation());
    world.add(new Rock());
    world.show();
  }
}
```

(b) No—if the actor has already been removed from the grid (and is currently out of the grid), it cannot be removed again. This version of `BugRunner.java` will compile, but when it runs it will throw an `IllegalStateException`.

```
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    Bug b = new Bug();
    world.add(b);
    world.add(new Rock());
    world.show();
    b.removeSelfFromGrid();
    b.removeSelfFromGrid();
  }
}
```

(c) Yes—an actor can be placed in the grid, remove itself, and then put itself back in the grid. Try this `ActorRunner.java`. It will compile and run without error.

```
public class ActorRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    Actor a = new Actor();
    world.add(a);
    Grid g = a.getGrid();  //must remember the grid for placement back in
                           //the grid
    world.add(new Rock());
    world.show();
    a.removeSelfFromGrid();
    a.putSelfInGrid(g,new Location(5,5)); //must specify a location here
  }
}
```

5. How can an actor turn 90 degrees to the right?

To turn an actor 90 degrees to the right, use the `setDirection` method as follows in an `Actor` class:

```
setDirection(getDirection() + Location.RIGHT)
or
setDirection(getDirection() + 90);
```

To turn our bug in `BugRunner.java`, try this code:

```
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    Bug b = new Bug();
    world.add(b);
    world.add(new Rock());
    world.show();
    b.setDirection(getDirection() + Location.RIGHT);
  }
}
```

Do You Know?

Set 6

1. Which statement(s) in the `canMove` method ensures that a bug does not try to move out of its grid?

```
if(!gr.isValid(next))
  return false;
```

This statement makes sure that the next location is a valid location in the grid.

2. Which statement(s) in the `canMove` method determines that a bug will not walk into a rock?

```
Actor neighbor = gr.get(next);
return (neighbor == null) || (neighbor instanceof Flower);
```

These two statements work together to make sure that a bug will only travel to the next location if it is unoccupied or occupied by a flower.

3. Which methods of the `Grid` interface are invoked by the `canMove` method and why?

`isValid` and `get`. These methods are called to ensure that the next location is a valid location in the grid and to look at the object in that location to ensure that it is empty or contains an actor that can be replaced by the bug.

4. Which method of the `Location` class is invoked by the `canMove` method and why?

`getAdjacentLocation`. This method is called by the bug with the bug's current direction to find its next possible location.

5. Which methods inherited from the `Actor` class are invoked by the `canMove` method?

`getLocation`, `getDirection`, `getGrid`

6. What happens in the `move` method when the location immediately in front of the bug is out of the grid?

The bug will remove itself from the grid.

7. Is the variable `loc` needed in the `move` method, or could it be avoided by calling `getLocation()` muliple times?

Yes, the variable `loc` is needed. The variable `loc` stores the bug's location before it moves. It is used to insert a flower in the bug's old location after the bug has moved to its new location.

8. Why do you think the flowers that are dropped by a bug have the same color as the bug?

(Answers may vary.) Initially (before the flower wilts), it is easier to see which bug dropped what flowers because the color of the bug and its flowers are the same.

9. When a bug removes itself from the grid, will it place a flower into its previous location?

   If you just call the `removeSelfFromGrid` method, no. This method is inherited from the `Actor` class. Actors do not put a flower in their old location.

   When the `removeSelfFromGrid` is called in the `Bug move` method, yes. A flower will be placed in a bug's vacated location. The following lines from the `move` method show these actions.

```
if (gr.isValid(next))
  moveTo(next);
else
  removeSelfFromGrid();
Flower flower = new Flower(getColor());
flower.putSelfInGrid(gr,loc);
```

10. Which statement(s) in the `move` method places the flower into the grid at the bug's previous location?

```
Flower flower = new Flower(getColor());
flower.putSelfInGrid(gr, loc);  //loc is the old location of the bug
```

11. If a bug needs to turn 180 degrees, how many times should it call the `turn` method?

    Four times—each turn is a 45 degree turn.

Part 3 **Group Activity**

1. Specify—When putting students into groups, one student could be assigned to the role of the client. The other students could ask this client the questions suggested in this activity (and other pertinent questions) and let the client decide what a Jumper will do. (Answers may vary.)
   (a) One choice could be to turn.
   (b) One choice could be to turn.
   (c) One choice could be to turn.
   (d) Turn, remove the actor from the grid are a few examples
   (e) Turn, remove the jumper from the grid are a few examples
   (f) What if the location in front of the `Jumper` is occupied? Can a `Jumper` jump over only rocks and flowers? Students may think of other questions as well.

2. Design (Answers may vary.)
   (a) There is a natural tendency to extend `Bug`. The description of `Jumper` states it is an actor, so it should extend `Actor`.
   (b) The `Bug` class is a good template for the `Jumper`. The methods `canMove` and `move` can be used to help code `canJump` and `Jump`.
   (c) It depends—if the students want to default the color of the `Jumper` to a color that is different from the default actor and/or provide a way for users to construct a `Jumper` with a color, yes.
   (d) `act`— a `Jumper` acts differently than an `Actor`.
   (e) `canJump`, `jump`— similar to the `Bug`'s `canMove` and `move` methods. Adding a `turn` method would also be helpful.
   (f) Place the `Jumper`
      - On the edge of the grid, facing the edge
      - One from the edge of the grid, facing the edge
      - Surrounded by rocks
      - Surrounded by flowers
      - Surrounded by `Actors`
      - `Place` a rock, flower, or an actor in the location two in front of the `Jumper`

3.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Flower;
import info.gridworld.actor.Rock;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A <code>Jumper</code> is an actor that will jump over Rocks and Flowers
 * and turn.
 */
public class Jumper extends Actor
{
  /**
   * Constructs a pink Jumper.
   */
  public Jumper()
  {
    setColor(Color.PINK);
  }

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

  public void act()
  {
    if (canJump())
      jump();
    else
      turn();
  }

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

```
  /**
   * Moves the Jumper forward two locations.
   * The location two in front must be valid or the Jumper will remove
   * itself from the grid.
   */
  public void jump()
  {
    Grid<Actor> gr = getGrid();
    if (gr == null)
      return;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    Location twoAway = next.getAdjacentLocation(getDirection());
    if (gr.isValid(twoAway))
      moveTo(twoAway);
    else
      removeSelfFromGrid();
  }

  /**
   * Tests whether this Jumper can move forward into a location two in
   * front that is empty or contains a flower.
   * The location one in front must be empty or contain a Rock or a
   * Flower.
   * @return true if this Jumper can move.
   */
  public boolean canJump()
  {
    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);
    if (!((neighbor == null) || (neighbor instanceof Flower)
                             || (neighbor instanceof Rock)))
      return false;

    Location twoAway = next.getAdjacentLocation(getDirection());
    if (!gr.isValid(twoAway))
      return false;

    neighbor = gr.get(twoAway);
    return (neighbor == null) || (neighbor instanceof Flower);
  }
}
```

Part 3

The `JumperRunner` class is provided below. When you run this class, click on the grid to add other actors.

```
import info.gridworld.actor.ActorWorld;

import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Flower;
/**
 * This class runs a world that contains a jumper, a bug, a flower, and a
 * rock added at random locations.
 */
public class JumperRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Jumper());
    world.add(new Rock());
    world.add(new Bug());
    world.add(new Flower());
    world.show();
  }
}
```

Do You Know?

Set 7

1. What methods are implemented in `Critter`?

   `act`, `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation`, `makeMove`

2. What are the five basic actions common to all critters when they act?

   `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation`, `makeMove`

3. Should subclasses of `Critter` override the `getActors` method? Explain.

   Yes—if the new critter subclass selects its actors from different locations than `Critter` class does, it will need to override this method.

4. Describe three ways that a critter could process actors.

   Answers may vary. It could eat all of the actors in its list, it could make them all change colors, or it could ask them all to move.

5. What three methods must be invoked to make a critter move? Explain each of these methods.

   `getMoveLocations`, `selectMoveLocation`, `makeMove`

   Moving a critter is a three-step process. First, the `act` method calls the `getMoveLocations` method. For a basic critter, this method returns a list of all the empty adjacent locations around the critter. After receiving the list of possible empty locations, the `selectMoveLocation` randomly chooses one of the locations and returns that location. If there are no empty locations to choose from, `selectMoveLocation` returns the current location of the critter. The returned location is then passed to the `makeMove` method, and the critter is moved to the new location.

6. Why is there no `Critter` constructor?

   `Critter` extends `Actor`. The `Actor` class has a default constructor. If you do not create a constructor in a class, Java will write a default constructor for you. The `Critter` default constructor that Java provides will call `super()`, which calls the `Actor` default constructor. The `Actor` default constructor will make a blue critter that faces north.

Do You Know?

Set 8

1. Why does `act` cause a `ChameleonCritter` to act differently from a `Critter` even though `ChameleonCritter` does not override `act`?

   The `act` method calls `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation`, and `makeMove`. The `ChameleonCritter` class overrides the `processActors` and `makeMove` methods. Therefore, calling `act` for a `ChameleonCritter` will produce different behavior than calling `act` for a `Critter`.

   A critter processes its actors by removing any neighbor that is not a `Rock` or a `Critter`. A `ChameleonCritter` processes its actors by randomly choosing one of its neighbors, getting the neighbor's color, and then changing its own color to that of its neighbor. When a `ChameleonCritter` calls `makeMove`, it first faces the direction of its next location and then moves. A `Critter` does not change its direction when it moves.

2. Why does the `makeMove` method of `ChameleonCritter` call `super.makeMove`?

   The `makeMove` method of the `ChameleonCritter` first changes the direction of the critter to face its new location. Then it calls `super.makeMove` of the `Critter` class to actually move to the new location. After it changes its direction, it behaves like (`makeMove` like) a `Critter`.

3. How would you make the `ChameleonCritter` drop flowers in its old location when it moves?

   Modify the `makeMove` method to drop flowers in the old location. A variable is needed to keep track of the `ChameleonCritter`'s current location. After the critter moves, put a flower in its old location only if the critter actually moved to a new location. See the modified `makeMove` method below.

```
public void makeMove(Location loc)
{
  Location oldLoc = getLocation();
  setDirection(getLocation().getDirectionToward(loc));
  super.makeMove(loc);
  if(!oldLoc.equals(loc)) //don't replace yourself if you did not move
  {
    Flower flo = new Flower(getColor());
    flo.putSelfInGrid(getGrid(), oldLoc);
  }
}
```

4. Why doesn't `ChameleonCritter` override the `getActors` method?

   Because it processes the same list of actors that its base class `Critter` does. Since `ChameleonCritter` does not define a new behavior for `getActors`, it does not need to override this method.

5. Which class contains the `getLocation` method?

   The `Actor` class contains the `getLocation` method. All `Actor` subclasses inherit this method.

6. How can a `Critter` access its own grid?

   A critter can access its grid by calling the `getGrid` method that it inherits from the `Actor` class.

Do You Know?

Set 9

1. Why doesn't `CrabCritter` override the `processActors` method?

   A `CrabCritter` processes its actors by eating all of the neighbors returned when `getActors` is called. This is the same behavior that it inherits from its base class `Critter`. There is no need to override this method.

2. Describe the process a `CrabCritter` uses to find and eat other actors. Does it always eat all neighboring actors? Explain.

   The `CrabCritter`'s `getActors` method only looks for neighbors that are immediately in front of the crab critter and to its right-front and left-front locations. Any neighbors found in these locations will be "eaten" when the `processActors` method is called. Actors in the other neighboring locations will not be disturbed.

3. Why is the `getLocationsInDirections` method used in `CrabCritter`?

   The parameter for this method brings in an array of directions. For the crab critter, this array contains the directions of the possible neighbors that this crab can eat. The method `getLocationsInDirections` uses this array to determine and return valid adjacent locations of this critter in the directions given by the array parameter.

4. If a `CrabCritter` has location (3,4) and faces south, what are the possible locations for actors that are returned by a call to the `getActors` method?

   (4,3), (4,4), and (4,5)

5. What are the similarities and differences between the movements of a `CrabCritter` and a `Critter`?

   Similarities: When critters and crab critters move, they do not turn in the direction that they are moving. They both randomly choose their next location from their list of possible move locations.

   Differences: A crab critter will only move to its left or its right. A critter's possible move locations are any of its eight adjacent neighboring locations. When a crab critter cannot move, it will randomly turn right or left. When a critter cannot move, it does not turn.

6. How does a `CrabCritter` determine when it turns instead of moving?

   If the parameter `loc` in `makeMove` is equal to the crab critter's current location, it turns instead of moving.

7. Why don't the `CrabCritter` objects eat each other?

   A crab critter inherits the `processActors` method from the `Critter` class. This method only removes actors that are not rocks and not critters. Since a `CrabCritter` is a `Critter`, a crab critter will not eat any other critter.

Part 4 Exercises

1. Modify the `processActors` method in `ChameleonCritter` so that if the list of actors to process is empty, the color of the `ChameleonCritter` will darken (like a flower).

Add the following static constant to the `ChameleonCritter` class (like the `Flower` class).

```
private static final double DARKENING_FACTOR = 0.05;

//Use the code found in the Flower class to darken a ChameleonCritter when it has no neighbors.
//This code has been put in a method, darken.

/**
 * 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)
  {
    darken();
    return;
  }

  int r = (int) (Math.random() * n);
  Actor other = actors.get(r);
  setColor(other.getColor());
}

/**
 * Darkens this critter's color by DARKENING_FACTOR.
 */
private void darken()
{
  Color c = getColor();
  int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
  int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
  int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));

  setColor(new Color(red, green, blue));
}
```

In the following exercises, your first step should be to decide which of the five methods—`getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation`, and `makeMove`—should be changed to get the desired result.

2. Create a class called `ChameleonKid` that extends `ChameleonCritter` as modified in exercise 1. A `ChameleonKid` changes its color to the color of one of the actors immediately in front or behind. If there is no actor in either of these locations, then the `ChameleonKid` darkens like the modified `ChameleonCritter`.

   Override the `getActors` method to only return actors that are in front and behind the `ChameleonCritterKid`. This solution uses the `getLocationsInDirections` method found in `CrabCritter` to find the actors in the required directions and uses the `CrabCritter`'s version of `getActors`.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.actor.Flower;
import info.gridworld.grid.Location;
import info.gridworld.grid.Grid;

import java.util.ArrayList;

/**
 * A <code>ChameleonKid</code> takes on the color of neighboring actors
 * that are in front or behind as it moves through the grid. <br />
 */
public class ChameleonKid extends ChameleonCritter
{
  /**
   * Gets the actors for processing. The actors must be contained in the
   * same grid as this critter. Implemented to return the actors that
   * occupy neighboring grid locations in front or behind this critter.
   * @return a list of actors that are neighbors of this critter
   */
  public ArrayList<Actor> getActors()
  {
    ArrayList<Actor> actors = new ArrayList<Actor>();
    int[] dirs =
    { Location.AHEAD, Location.HALF_CIRCLE };
    for (Location loc : getLocationsInDirections(dirs))
    {
      Actor a = getGrid().get(loc);
      if (a != null)
        actors.add(a);
    }

    return actors;
  }
```

```
  /**
   * Finds the valid adjacent locations of this critter in different
   * directions.
   * @param directions - an array of directions (which are relative to the
   * current direction)
   * @return a set of valid locations that are neighbors of the current
   * location in the given directions
   */
  public ArrayList<Location> getLocationsInDirections(int[] directions)
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    Grid gr = getGrid();
    Location loc = getLocation();

    for (int d : directions)
    {
      Location neighborLoc = loc.getAdjacentLocation(getDirection() + d);
      if (gr.isValid(neighborLoc))
        locs.add(neighborLoc);
    }
    return locs;
  }
}
```

3. Create a class called `RockHound` that extends `Critter`. A `RockHound` gets the actors to be processed in the same way as a `Critter`. It removes any rocks in that list from the grid. A `RockHound` moves like a `Critter`.

Override `processActors` to remove all rocks from the list of neighboring locations.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;

import java.util.ArrayList;

public class RockHound extends Critter
{
  /**
   * Processes the actors. Implemented to "eat" (i.e. remove) all rocks
   * <br />
   * Precondition: All objects in <code>actors</code> are contained in the
   * same grid as this critter.
   * @param actors the actors to be processed
   */
  public void processActors(ArrayList<Actor> actors)
  {
    for (Actor a : actors)
    {
      if (a instanceof Rock)
        a.removeSelfFromGrid();
    }
  }
}
```

4. Create a class `BlusterCritter` that extends `Critter`. A `BlusterCritter` looks at all of the neighbors within two steps of its current location. (For a `BlusterCritter` not near an edge, this includes 24 locations). It counts the number of critters in those locations. If there are fewer than *c* critters, the `BlusterCritter`'s color gets brighter (color values increase). If there are *c* or more critters, the `BlusterCritter`'s color darkens (color values decrease). Here, *c* is a value that indicates the courage of the critter. It should be set in the constructor.

Override the `getActors` and `processActors` methods to create the new behavior required of the `BlusterCritter`. Create two new methods to lighten and darken the color of the `BlusterCritter`. To darken a `BlusterCritter`, subtract one from the red, green, and blue components as long as they are greater than 0 (or use the same process as the `Flower` class to darken the critter). To lighten a `BlusterCritter`, add one to the red, green, and blue components as long as they are less than 255.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Critter;

import info.gridworld.grid.Location;

import java.util.ArrayList;
import java.awt.Color;

public class BlusterCritter extends Critter
{

  private int courageFactor;

  public BlusterCritter(int c)
  {
    super();
    courageFactor = c;
  }
```

```
/**
 * Gets the actors for processing. The actors must be contained in the
 * same grid as this critter. Implemented to return the actors that
 * occupy neighboring grid locations within two steps of this critter
 * @return a list of actors that are neighbors of this critter
 */
public ArrayList<Actor> getActors()
{
  ArrayList<Actor> actors = new ArrayList<Actor>();

  Location loc = getLocation();
  for(int r = loc.getRow() - 2; r <= loc.getRow() + 2; r++ )
    for(int c = loc.getCol() - 2; c <= loc.getCol() + 2; c++)
    {
      Location tempLoc = new Location(r,c);
      if(getGrid().isValid(tempLoc))
      {
        Actor a = getGrid().get(tempLoc);
        if(a != null && a != this)
          actors.add(a);
      }
    }
  return actors;
}

/**
 * Processes the actors. Implemented to count all the actors within
 * 2 locations of this critter.  If there are fewer than courageFactor
 * critters in these locations, this BlusterCritter lightens, otherwise
 * it darkens.
 * Precondition: All objects in <code>actors</code> are contained in the
 * same grid as this critter.
 * @param actors the actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
  int count = 0;
  for(Actor a: actors)
  if(a instanceof Critter)
    count++;
  if(count < courageFactor)
    lighten();
  else
    darken();
}
```

```
/**
 * Darkens this critter's color by subtracting 1 from red, green, and
 * blue components if they are greater than 0. To darken the color
 * faster, subtract a slightly larger value.
 */
private void darken()
{
  Color c = getColor();
  int red = c.getRed();
  int green = c.getGreen();
  int blue = c.getBlue();

  if(red > 0) red--;
  if(green > 0) green--;
  if(blue > 0) blue--;

  setColor(new Color(red, green, blue));


  // this segment of code uses same logic as the flower class to darken
  // an object's color
  // to use this technique add DARKENING_FACTOR as a class instance
  // variable; then replace the active code for darken with the
  // following five lines of code

  // private static final double DARKENING_FACTOR = 0.05;

  // Color c = getColor();
  // int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
  // int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
  // int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));

  // setColor(new Color(red, green, blue));
}
```

```
  /**
   * Lightens this critter's color by adding 1 to the red, green, and blue
   * components if they are less than 255. To lighten the color faster,
   * add a slightly larger value.
   */
  private void lighten()
  {
    Color c = getColor();
    int red = c.getRed();
    int green = c.getGreen();
    int blue = c.getBlue();

    if(red < 255) red++;
    if(green < 255) green++;
    if(blue < 255) blue++;

    setColor(new Color(red, green, blue));
  }
}
```

5. Create a class `QuickCrab` that extends `CrabCritter`. A `QuickCrab` processes actors the same way a `CrabCritter` does. A `QuickCrab` moves to one of the two locations, randomly selected, that are two spaces to its right or left, if that location and the intervening location are both empty. Otherwise, a `QuickCrab` moves like a `CrabCritter`.

This solution overrides the `getMoveLocations` method. In this solution, an additional method was created to find good locations two spaces away and add them to the `ArrayList` that `getMoveLocations` returns.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;
import java.util.ArrayList;

/**
 * A <code>QuickCrab</code> looks at a limited set of neighbors when it
 * eats and moves.<br />
 */
public class QuickCrab extends CrabCritter
{

  public QuickCrab()
  {
    setColor(Color.CYAN);
  }
```

```
  /**
   * @return list of empty locations
   * two locations to the right and two locations to the left
   */
  public ArrayList<Location> getMoveLocations()
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    Grid g = getGrid();

    addIfGoodTwoAwayMove(locs,getDirection() + Location.LEFT);
    addIfGoodTwoAwayMove(locs,getDirection() + Location.RIGHT);

    if (locs.size() == 0)
    return super.getMoveLocations();

    return locs;
  }

  /**
   * Adds a valid and empty two away location in direction dir to the
   * ArrayList locs.
   * To be a valid two away location, the location that is one away in
   * direction dir must also be valid and empty.
   */

  private void addIfGoodTwoAwayMove(ArrayList<Location> locs,int dir)
  {
    Grid g = getGrid();
    Location loc = getLocation();

    Location temp = loc.getAdjacentLocation(dir);

    if(g.isValid(temp) && g.get(temp) == null)
    {
      Location loc2 = temp.getAdjacentLocation(dir);
      if(g.isValid(loc2) && g.get(loc2)== null)
        locs.add(loc2);
    }
  }

}
```

6. Create a class `KingCrab` that extends `CrabCritter`. A `KingCrab` gets the actors to be processed in the same way a `CrabCritter` does. A `KingCrab` causes each actor that it processes to move one location further away from the `KingCrab`. If the actor cannot move away, the `KingCrab` removes it from the grid. When the `KingCrab` has completed processing the actors, it moves like a `CrabCritter`.

This solution overrides the `processActors` method. This solution also includes two new methods, `distanceFrom` and `moveOneMoreAway`, to move an actor away from the `KingCrab`.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;
import java.util.ArrayList;

/**
 * A <code>KingCrab</code> looks at a limited set of neighbors when it
 * eats and moves.<br />
 */
public class KingCrab extends CrabCritter
{
  public KingCrab()
  {
    setColor(Color.PINK);
  }

  /**
   * Computes the rounded integer distance between two given locations.
   */
  public int distanceFrom(Location loc1, Location loc2)
  {
    int x1 = loc1.getRow();
    int y1 = loc1.getCol();
    int x2 = loc2.getRow();
    int y2 = loc2.getCol();
    double dist = Math.sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)) + .5;
    return (int)Math.floor(dist);
  }
```

```
  /*
   * This method moves the Actor to a location that is one location
   * further away from this KingCrab and returns true.  If there is no
   * location that is one location further away, the method returns false.
   */
  private boolean moveOneMoreAway(Actor a)
  {
    ArrayList<Location> locs =
      getGrid().getEmptyAdjacentLocations(a.getLocation());

    for(Location loc:locs)
    {
      if(distanceFrom(getLocation(), loc) > 1)
      {
        a.moveTo(loc);
        return true;
      }
    }

    return false;
  }

  /*
   * Each actor in the list actors is told to move one location further
   * away from this KingCrab.  If that is not possible, the actor is
   * removed from the grid.
   */
  public void processActors(ArrayList<Actor> actors)
  {
    for (Actor a : actors)
    {
      if (!moveOneMoreAway(a))
      {
        a.removeSelfFromGrid();
      }
    }
  }
}
```

Do You Know?
Set 10

1. Where is the `isValid` method specified? Which classes provide an implementation of this method?

   The `isValid` method is specified in the `Grid` interface. The `BoundedGrid` and `UnboundedGrid` classes implement this method.

2. Which `AbstractGrid` methods call the `isValid` method? Why don't the other methods need to call it?

   Method `getValidAdjacentLocations` calls the `isValid` method. The methods `getEmptyAdjacentLocations` and `getOccupiedAdjacentLocations` do not directly call `isValid`. Instead, they both call `getValidAdjacentLocations`, which calls `isValid`. Method `getNeighbors` does not directly call `isValid`. It calls `getOccupiedAdjacentLocations`, which calls `getValidAdjacentLocations` which calls `isValid`.

3. Which methods of the `Grid` interface are called in the `getNeighbors` method? Which classes provide implementations of these methods?

   Method `getNeighbors` calls the `Grid` methods `get` and `getOccupiedAdjacentLocations`. The `AbstractGrid` class implements the `getOccupiedAdjacentLocations` method. The `get` method is implemented in the `BoundedGrid` and `UnboundedGrid` classes.

4. Why must the `get` method, which returns an object of type `E`, be used in the `getEmptyAdjacentLocations` method when this method returns locations, not objects of type `E`?

   The `get` method returns a reference to the object stored in the grid at the given location or `null` if no object exists. The `getEmptyAdjacentLocations` calls the `get` method and tests the result to see if it is `null`. If `null` is returned, that location is "empty" and is added to the list. Calling the `get` method is the only way to test whether or not a given location is empty or occupied.

5. What would be the effect of replacing the constant `Location.HALF_RIGHT` with `Location.RIGHT` in the two places where it occurs in the `getValidAdjacentLocations` method?

   The number of possible valid adjacent locations would decrease from eight to four. The valid adjacent locations would be those that are north, south, east, and west of the given location.

Do You Know?

Set 11

1. What ensures that a grid has at least one valid location?

   The constructor for the `BoundedGrid` will throw an `IllegalArgumentException` if the number of rows <= 0 or the number of cols <= 0.

2. How is the number of columns in the grid determined by the `getNumCols` method? What assumption about the grid makes this possible?

   `occupantArray[0].length`
   `getNumCols` returns the number of columns in row 0 of the `occupantArray`. The constructor ensures that each `BoundedGrid` object has at least one row and one column.

3. What are the requirements for a `Location` to be valid in a `BoundedGrid`?

   A location's row value has to be greater than or equal to 0 and less than the number of rows in the `BoundedGrid`. A location's column value has to be greater than or equal to 0 and less than the number of columns in the `BoundedGrid`.

   In the next four questions, let *r* = number of rows, *c* = number of columns, and *n* = number of occupied locations.

4. What type is returned by the `getOccupiedLocations` method? What is the time complexity (Big-Oh) for this method?

   An `ArrayList<Location>` is returned by the `getOccupiedLocations` method.

   O(r * c)—each location in the `BoundedGrid` must be visited to see if it is occupied or not. Occupied locations are added to the end of the `ArrayList` (O(1)).

5. What type is returned by the `get` method? What parameter is needed? What is the time complexity (Big-Oh) for this method?

   The type returned for the `get` method is `E`, which is whatever type is stored in the `occupantArray`.

   The `get` method requires a `Location` object.

   Accessing a two-dimensional array given and row and column value is O(1).

6. What conditions may cause an exception to be thrown by the `put` method? What is the time complexity (Big-Oh) for this method?

   If the location where an object is to be added is not in the grid (invalid location), an `IllegalArgumentException` will be thrown.

   If the object sent to the `put` method is `null`, a `NullPointerException` will be thrown.

   The time complexity for the `put` method is O(1).

7. What type is returned by the `remove` method? What happens when an attempt is made to remove an item from an empty location? What is the time complexity (Big-Oh) for this method?

   The generic type `E`: whatever type is actually stored in the `BoundedGrid` object.

   If an attempt is made to remove an item from an empty location, `null` is stored in the location and `null` is returned. It is not an error to call the `Grid` class's `remove` method on a location that is empty.
   The time complexity for the `remove` method is O(1).

8. Based on the answers to questions 4, 5, 6, and 7, would you consider this an efficient implementation? Justify your answer.

   Overall, yes. The only method that is inefficient is the `getOccupiedLocations` method, O(r * c). The other methods (`put`, `get`, and `remove`) are O(1).

   A `BoundedGrid` stores more than just its occupants. It also stores `null` values in the unoccupied locations. A bounded grid that only stored the occupants, while still offering O(1) access to the occupants, would be a better implementation. A `HashMap` could implement the `BoundedGrid` this way.

Do You Know?

Set 12

1. Which method must the `Location` class implement so that an instance of `HashMap` can be used for the map? What would be required of the `Location` class if a `TreeMap` were used instead? Does `Location` satisfy these requirements?

   The `Location` class must implement the `hashCode` and the `equals` methods. The `hashCode` method must return the same value for two locations that test true when the `equals` method is called.

   The `Location` class implements the `Comparable` interface. Therefore, the `compareTo` method must be implemented for the `Location` class to be a nonabstract class. The `compareTo` method should return 0 for two locations that test true when the `equals` method is called. The `TreeMap` requires keys of the map to be `Comparable`.

   The `Location` class satisfies all of these requirements.

2. Why are the checks for `null` included in the `get`, `put`, and `remove` methods? Why are no such checks included in the corresponding methods for the `BoundedGrid`?

   The `UnboundedGrid` uses a `HashMap` as its data structure to hold the items in the grid. All non-null locations are valid in the `UnBoundedGrid`. The `isValid` method for the `UnBoundedGrid` always returns `true`; it does not check for `null` locations. In a `Map` object, `null` is a legal value for a key. In an `UnboundedGrid` object, `null` is not a valid location. Therefore, the `UnboundedGrid` methods `get`, `put`, and `remove` must check the location parameter and throw a `NullPointerException` when the parameter is `null`.

   The `isValid` method is called before using a location to access the `occupantArray` in the `BoundedGrid`. If the location parameter is `null` in the `isValid` method, an attempt to access `loc.getRow()` will cause a `NullPointerException` to be thrown.

   If code is written that does not call the `isValid` method before calling the `get`, `put`, and `remove` methods, attempting to access `loc.getRow()` in these methods will also cause a `NullPointerException` to be thrown.

3. What is the average time-complexity (Big-Oh) for the three methods: `get`, `put`, and `remove`? What would it be if a `TreeMap` were used instead of a `HashMap`?

   The average time complexity for the `get`, `put`, and `remove` is O(1).

   If a `TreeMap` were used instead of a `HashMap`, the average time complexity would be O(log n), where n is the number of occupied locations in the grid.

4. How would the behavior of this class differ, aside from time complexity, if a `TreeMap` were used instead of a `HashMap`?

   Most of the time, the `getOccupiedLocations` method will return the occupants in a different order. Keys (locations) in a `HashMap` are placed in a hash table based on an index that is calculated by using the keys' `hashCode` and the size of the table. The order in which a key is visited when the `keySet` is traversed depends on where it is located in the hash table.

   A `TreeMap` stores its keys in a balanced binary search tree and traverses this tree with an inorder traversal. The keys in the `keySet (Locations)` will be visited in ascending order (row major order) as defined by `Location's compareTo` method.

5. Could a map implementation be used for a bounded grid? What advantage, if any, would the two-dimensional array implementation that is used by the `BoundedGrid` class have over a map implementation?

   Yes, a map could be used to implement a bounded grid.

   If a `HashMap` were used to implement the bounded grid, the average time complexity for `getOccupiedLocations` would be O(n), where n is the number of items in the grid.

   If the bounded grid were almost full, the map implementation would use more memory, because the map stores the item and its location. A two-dimensional array only stores the items. The locations are produced by combining the row and column indices for each item.

Part 5 Exercises

1. Suppose that a program requires a very large bounded grid that contains very few objects and that the program frequently calls the `getOccupiedLocations` method (as, for example, `ActorWorld`). Create a class `SparseBoundedGrid` that uses a "sparse array" implementation. Your solution need not be a generic class; you may simply store occupants of type `Object`.

   The sparse array is an array of linked lists. Each linked list entry holds both a grid occupant and a column index. Each entry in the array is a linked list or is `null` if that row is empty.

   You may choose to implement the linked list in one of two ways.You can use raw list nodes.

```
public class SparseGridNode
{
      private Object occupant;
      private int col;
      private SparseGridNode next;
      . . .
}
```

   Or you can use a `LinkedList<OccupantInCol>` with a helper class.

```
public class OccupantInCol
{
      private Object occupant;
      private int col;
      . . .
}
```

   For a grid with *r* rows and *c* columns, the sparse array has length *r*. Each of the linked lists has maximum length *c*.

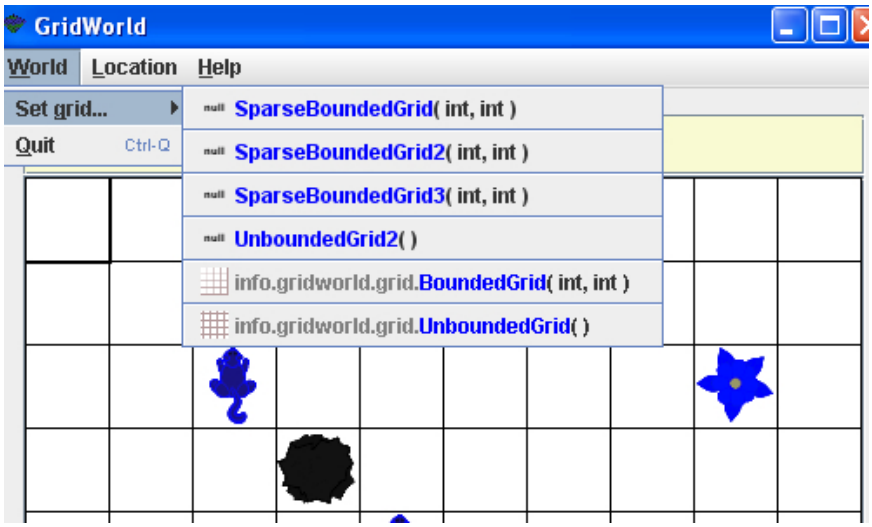
   Implement the methods specified by the `Grid` interface using this data structure. Why is this a more time-efficient implementation than `BoundedGrid`?

   For a program that requires a very large bounded grid and frequently calls the `getOccupiedLocations` method, the time complexity for this implementation is O(r + n), where r is the number of rows and n is the number of items in the grid. The time complexity for the `BoundedGrid` implementation for this method is O(r * c), where r is the number of rows and c is the number of columns.

The `World` has a public `addGridClass` method. Since the `ActorWorld` is a `World`, you can call this method in a runner. Here is the code to add a new grid to the GUI.

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import info.gridworld.actor.Critter;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Flower;

/**
 * This class runs a world with additional grid choices.>
 */
public class SparseGridRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.addGridClass("SparseBoundedGrid");
    world.addGridClass("SparseBoundedGrid2");
    world.addGridClass("SparseBoundedGrid3");
    world.addGridClass("UnboundedGrid2");
    world.add(new Location(2, 2), new Critter());
    world.show();
  }
}
```

When you execute a runner class and choose the World menu->set grid, the new grid type will be available for you to choose.



SparseGridNode.java

```
public class SparseGridNode
{
  private Object occupant;
  private int col;
  private SparseGridNode next;


  public SparseGridNode(Object occ, int colNum,
                        SparseGridNode initNext)
  {
    occupant = occ;
    col = colNum;
    next = initNext;
  }

  public Object getOccupant()
  {
    return occupant;
  }

  public int getColumn()
  {
    return col;
  }

  public SparseGridNode getNext()
  {
    return next;
  }

  public void setOccupant(Object occ)
  {
    occupant = occ;
  }

  public void setNext(SparseGridNode newNext)
  {
    next = newNext;
  }
}
```

```
SparseBoundedGrid.java - SparseGridNode linked list of SparseGridNode version

import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;

import java.util.ArrayList;

/**
 * A <code>BoundedGrid</code> is a rectangular grid with a finite
 * number of rows and columns. <br />
 * The implementation of this class is testable on the AP CS AB exam.
 */
public class SparseBoundedGrid<E> extends AbstractGrid<E>
{
  private SparseGridNode[] occupantArray; // the array storing the
                                          // grid elements
  private int numCols;
  private int numRows;
  /**
   * Constructs an empty bounded grid with the given dimensions.
   * (Precondition: <code>rows > 0</code> and <code>cols > 0</code>.)
   * @param rows number of rows in BoundedGrid
   * @param cols number of columns in BoundedGrid
   */
  public SparseBoundedGrid(int rows, int cols)
  {
    if (rows <= 0)
      throw new IllegalArgumentException("rows <= 0");
    if (cols <= 0)
      throw new IllegalArgumentException("cols <= 0");
    numCols = cols;
    numRows = rows;
    occupantArray = new SparseGridNode[rows];
  }

  public int getNumRows()
  {
    return numRows;
  }

  public int getNumCols()
  {
    return numCols;
  }
```

```
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++)
  {

    SparseGridNode p = occupantArray[r]; //get the row linked list
    while(p != null) //traverse the row
    {
      Location loc = new Location(r, p.getColumn());
      theLocations.add(loc);
      p = p.getNext();
    }
  }
  return theLocations;
}

public E get(Location loc)
{
  if (!isValid(loc))
    throw new IllegalArgumentException("Location " + loc
                                          + " is not valid");

  SparseGridNode p = occupantArray[loc.getRow()]; //get the row
  while(p != null) //traverse to find the item at location loc
  {
    if(loc.getCol() == p.getColumn())
      return (E)p.getOccupant(); //must cast to E
    p = p.getNext();
  }

  return null;
}
```

```
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");

  // Remove the old occupant from the grid.
  E oldOccupant = remove(loc);

  //The following steps add the object to the grid.

  //Get the row for the new occupant
  SparseGridNode p = occupantArray[loc.getRow()];

  //put the new occupant on the front of the list
  occupantArray[loc.getRow()] = new SparseGridNode(obj,
                                                   loc.getCol(), p);

  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 obj = get(loc);

    if (obj == null) return null; //if not found, return null

    SparseGridNode p = occupantArray[loc.getRow()];

    if(p != null)
    {
      if(p.getColumn() == loc.getCol()) //check the first node
        occupantArray[loc.getRow()] = p.getNext();
      else
      {
         //q searches for the occupant in the loc.getCol()
         //p stays behind to remove the occupant, if found
         SparseGridNode q = p.getNext();
         while(q!= null && q.getColumn() != loc.getCol())
         {
           p = p.getNext();
           q = q.getNext();
         }

         //if found, remove the occupant at loc.getCol()
         if(q != null)
           p.setNext(q.getNext());
      }
    }
    return obj;
  }
}
```

```
OccupantInCol.java - For LinkedList version

public class OccupantInCol
{
  private Object occupant;
  private int col;

  public OccupantInCol(Object occ, int colNum)
  {
    occupant = occ;
    col = colNum;
  }

  public Object getOccupant()
  {
    return occupant;
  }

  public int getColNum()
  {
    return col;
  }

  public void setOccupant(Object occ)
  {
    occupant = occ;
  }

}
```

```
SparseBoundedGrid2.java - LinkedList version

import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;

import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * A <code>BoundedGrid</code> is a rectangular grid with a finite
 * number of rows and columns. <br />
 * The implementation of this class is testable on the AP CS AB exam.
 */
public class SparseBoundedGrid2<E> extends AbstractGrid<E>
{
  private ArrayList<LinkedList> occupantArray; // the array storing
                                               // the grid elements
  private int numCols;
  private int numRows;
  /**
   * Constructs an empty bounded grid with the given dimensions.
   * (Precondition: <code>rows > 0</code> and <code>cols > 0</code>.)
   * @param rows number of rows in BoundedGrid
   * @param cols number of columns in BoundedGrid
   */
  public SparseBoundedGrid2(int rows, int cols)
  {
    if (rows <= 0)
      throw new IllegalArgumentException("rows <= 0");
    if (cols <= 0)
       throw new IllegalArgumentException("cols <= 0");
    numCols = cols;
    numRows = rows;
    occupantArray = new ArrayList<LinkedList>();
    for(int j = 0; j < rows; j++)
    {
      occupantArray.add(new LinkedList<OccupantInCol>());
    }
  }

  public int getNumRows()
  {
    return numRows;
  }
```

```
public int getNumCols()
{
  return numCols;
}

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++)
  {
    //get the row linked list
    LinkedList<OccupantInCol> row = occupantArray.get(r);
    if(row != null)
    {
      for(OccupantInCol occ: row) //traverse the row
      {
        Location loc = new Location(r, occ.getColNum());
        theLocations.add(loc);
      }
    }
  }
  return theLocations;
}
```

```
public E get(Location loc)
{
  if (!isValid(loc))
    throw new IllegalArgumentException("Location " + loc
                                       + " is not valid");

  LinkedList<OccupantInCol> row = occupantArray.get(loc.getRow());
                                                    //get the row

  if(row != null) //traverse to find the item at location loc
  {
    for(OccupantInCol occ: row)
    {
      if(loc.getCol() == occ.getColNum())
      {
        return (E)occ.getOccupant(); //must cast to E
      }
    }
  }
  return null;
}


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");

  // Remove the old occupant from the grid, if there is one.
  E oldOccupant = remove(loc);

  //Add the object to the grid.
  occupantArray.get(loc.getRow()).add(
                               new OccupantInCol(obj,loc.getCol()));

  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 obj = get(loc);

    if (obj == null) return null; //if not found, return null

    LinkedList<OccupantInCol> row = occupantArray.get(loc.getRow());

    if(row != null)
    {
      Iterator<OccupantInCol> itr = row.iterator();
      while(itr.hasNext())
      {
        if(itr.next().getColNum() == loc.getCol())
        {
          itr.remove();
          break;
        }
      }
    }
    return obj;
  }

}
```

2. Consider using a `HashMap` or `TreeMap` to implement the `SparseBoundedGrid`. How could you use the `UnboundedGrid` class to accomplish this task? Which methods of `UnboundedGrid` could be used without change?

   Most of the code in the `UnboundedGrid` class could be used without modification to create a `SparseBoundedGrid` class.

   The methods that do not need to be changed are:

   - `getOccupiedLocations`
   - `get`
   - `put`
   - `remove`

   The changes needed are highlighted in the following `SparseBoundedGrid` class:

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

import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

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

  /**
   * Constructs an empty unbounded grid.
   */
  public SparseBoundedGrid(int rows, int cols)
  {
    occupantMap = new HashMap<Location, E>();
    numRows = rows;
    numCols = cols;
  }

  public int getNumRows()
  {
    return numRows;
  }
```

```
  public int getNumCols()
  {
    return numCols;
  }

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

  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);
  }
}
```

Part 5

Fill in the following chart to compare the expected Big-Oh efficiencies for the following methods for each implementation of the `SparseBoundedGrid`.

Let r = number of rows, c = number of columns, and n = number of occupied locations

Methods	**`SparseGridNode` version**	**`LinkedList <OccupantInCol>` version**	**`HashMap` version**	**`TreeMap` version**
`getNeighbors`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**
`getEmptyAdjacentLocations`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**
`getOccupiedAdjacentLocations`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**
`getOccupiedLocations`	**O(r + n)**	**O(r + n)**	**O(n)**	**O(n)**
`get`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**
`put`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**
`remove`	**O(c)**	**O(c)**	**O(1)**	**O(log n)**

3. Consider this implementation of an unbounded grid in which all valid locations have non-negative row and column values. The constructor allocates a 16 x 16 array. When a call is made to the `put` method with a row or column index that is outside the current array bounds, double both array bounds until they are large enough, construct a new square array with those bounds, and place the existing occupants into the new array.

   Implement the methods specified by the `Grid` interface using this data structure. What is the Big-Oh efficiency of the `get` method? What is the efficiency of the `put` method when the row and column index values are within the current array bounds? What is the efficiency when the array needs to be resized?

   Big-Oh of `get`: O(1)

   Big-Oh of `put`: O(1) when row and column index are within current array bounds
   O(dim * dim), where dim is the number of rows and columns in the current array before it is resized. Must traverse through the current array to copy values over to the new resized array.

```
import info.gridworld.grid.*;

import java.util.ArrayList;

import java.util.*;

/**
 * An <code>UnboundedGrid</code> is a rectangular grid with an
 * unbounded number of rows and columns. <br />
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public class UnboundedGrid2<E> extends AbstractGrid<E>
{
  private  Object[][] occupantArray;
  private int dim; //current dimension of the occupantArray

  /**
   * Constructs an empty unbounded grid.
   */
  public UnboundedGrid2()
  {
    dim = 16;
    occupantArray = new Object[dim][dim];;
  }

  public int getNumRows()
  {
    return -1;
  }

  public int getNumCols()
  {
    return -1;
  }

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

```
public ArrayList<Location> getOccupiedLocations()
{
  ArrayList<Location> theLocations = new ArrayList<Location>();

  // Look at all grid locations.
  for (int r = 0; r < dim; r++)
  {
    for (int c = 0; c < dim; 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 null if a location is valid, but not in the array
  if(loc.getRow() >= dim || loc.getCol() >= dim)
    return null;
  return (E) occupantArray[loc.getRow()][loc.getCol()];
                                         // unavoidable warning
}

public E put(Location loc, E obj)
{
  if (loc == null)
     throw new NullPointerException("loc == null");
  if (obj == null)
     throw new NullPointerException("obj == null");

  //if new location is out of the array, resize the array
  if (loc.getRow() >= dim || loc.getCol() >= dim)
    resize(loc);

  // 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");

    // if location is valid and not in array, return null
    if(loc.getRow() >= dim || loc.getCol() >= dim)
      return null;

    // Remove the object from the grid.
    E r = get(loc);
    occupantArray[loc.getRow()][loc.getCol()] = null;
    return r;
  }

  private void resize(Location loc)
  {
    //double the size until it is greater than needed
    int size = dim;
    while (loc.getRow() >= size || loc.getCol() >= size)
      size *= 2;

    //create a new array
    Object[][] temp = new Object[size][size];

    //copy over the old contents
    for(int r = 0; r < dim; r++)
      for(int c = 0; c < dim; c++)
         temp[r][c] = occupantArray[r][c];

    //assign occupantArray the new array and update dim
    occupantArray = temp;
    dim = size;
  }
}
```