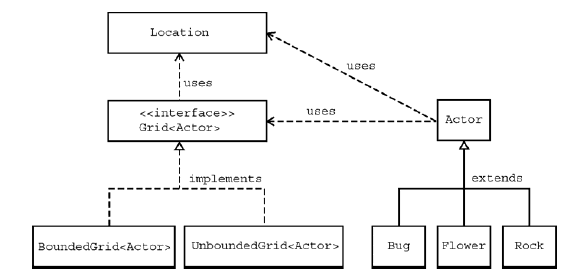
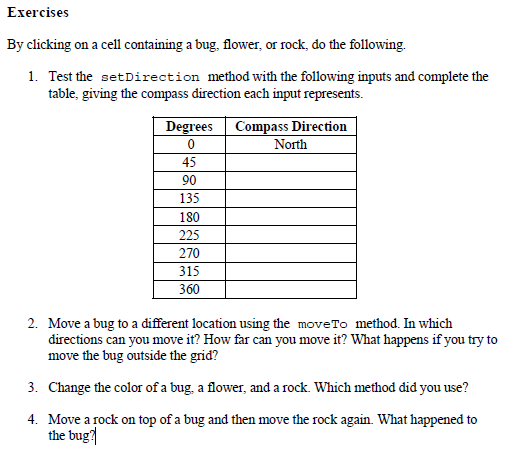
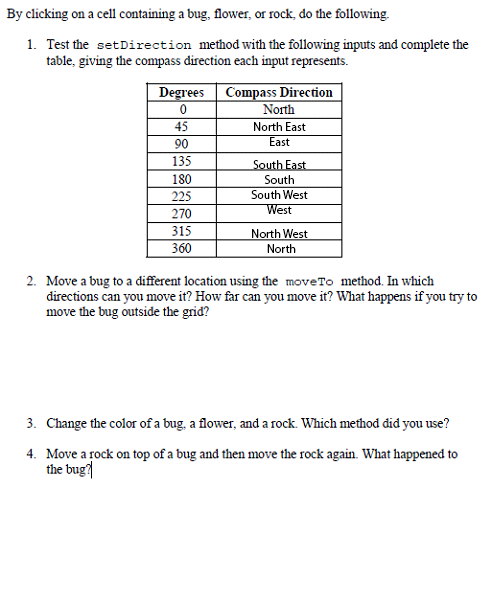
**Grid World**



**Assignment Part 1 – The GridWorld GUI**

**Questions and Exercises**

Use the default Starter Runner class, start up a world and perform the following actions:



1. 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?

Within an enclosed grid, it can only move within the grid, trying to go outside the grid throws a java.lang.IllegalArgumentException: Location (x, y) is not valid error.

It can only move to an adjacent location.

1. The Location class provides some final variables to help you setDirection. Write the code to set a Direction to SouthWest.

xman.setDirection(225);

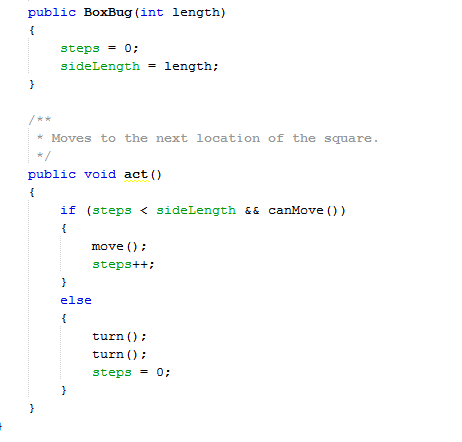
1. Move a rock on top of a bug and then move the rock again. What happened to the bug?

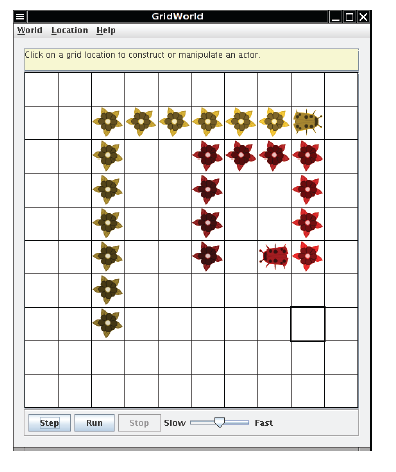
The bug got squashed and is no longer on the grid.

**Part 2 – Bugs**

Take a moment to familiarize yourself with the ***Bug*** API. There are 3 key methods -> canMove, move, turn. Plus there are all the methods inherited from ***Actor***.

While the Bug class has default behaviours, you can always extend the Bug class to create your own behaviour. The only method that must be overridden is that ***act()*** method which gets called each step of the simulation. Consider the new Bug Below:



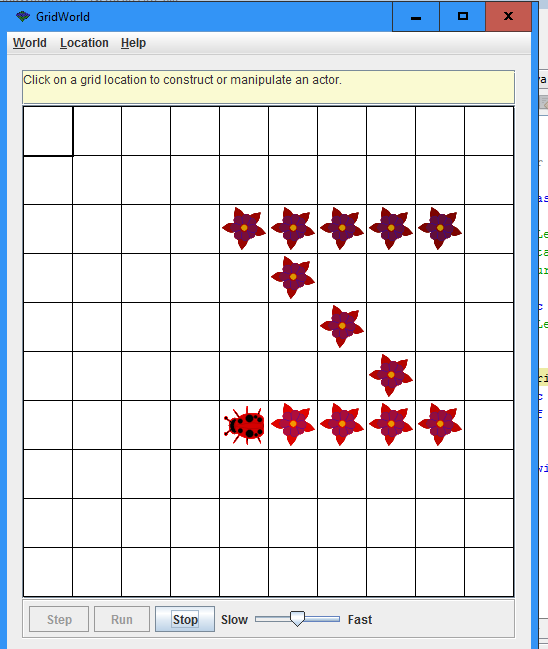


Questions (Add comments to your code to answer the following)

1. What is the role of the instance variable sideLength?
2. What is the role of the instance variable steps?
3. Why is the turn method called twice when the steps become equal to the sideLength?
4. Why can the move method be called in the BoxBug class when there is no move method in the BoxBug code?
5. Is it possible to change the size of a BoxBug’s square pattern after it has been constructed?
6. When will the value of steps be zero? Be specific and exhaustive.

Exercises

1. Write a class SBug to implement bugs that move in a “S” pattern. Starting in the top right corner. After completing one “S” pattern, a SBug should stop moving. In any step, if a SBug can’t move and is still attempting to complete its “S” pattern, the SBug does not move and should NOT start a new side. Supply the length of the “S” as a parameter in the constructor. The following image shows a “S” pattern of length 5. Notice that a SBug needs to be facing west before beginning its “S” pattern.



**Part 3 – Critters**

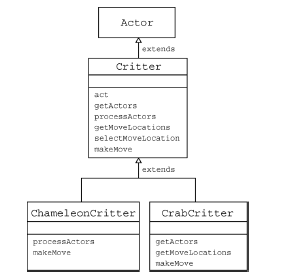
Prerequisite: If Mr. R-D hasn’t taught you ArrayLists yet, shame on him. Make sure you tell him off.

Critters are different Actors than bugs. Their purpose is to have a common ***pattern*** of behaviour, but the details of that pattern can be changed. We therefore do NOT override the ***act()*** method, but rather the individual behaviours within the pattern.

The ***act()*** behaviour of a Critter is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Action** | **Default Behaviour** | **Java Method** |
| 1 | get a list of all of the “relevant actors around it | returns a list of adjacent actors | *getActors()* |
| 2 | interact with those relevant actors in some way | eats (removes) all actors | *processActors(ArrayList)* |
| 3 | get a list of possible locations to move to | all ***adjacent*** empty locations | *getMoveLocations()* |
| 4 | choose a location to move to | randomly | *selectMoveLocation(ArrayList)* |
| 5 | move to the chosen location | called moveTo | *makeMove(Location)* |

Since you are not supposed to change the pattern of behaviour, we do not usually override the act() method, but rather the individual methods representing the actions we want to change. For example: Choosing different possible locations, choosing the location to move to differently, etc. The image below shows two different Critters that extend Critter in different ways.



Have a look at the Chameleon Critter provided to you by your teacher in order to get a good understanding of how these methods can be overridden.

**Questions**

1. Why does act() cause the ChameleonCritter to behave differently even though it does not override act()?

The methods processActors() and makeMove() that act() uses have been overrided instead.

1. What does it mean that makeMove calls super.makeMove() in ChameleonCritter? And why does it do that? Add comments to the code to explain.
2. How would you make the ChameleonCritter drop flowers in its old location when it moves?

**Exercises**

1. Modify the *processActors* method in *ChameleonCritter* so that if the list of actors to process is empty, the colour of the ChameleonCritter will darken (like a flower). All other behaviour remains the same.
2. Create a class called *Obliterator* that extends *Critter*. An Obliterator removes any non-Critter from the list of actors. Rem: instanceof can be used to determine if an object is an instance of a particular class. For example: x instanceOf BoxBug