GRIDWORLD OVERVIEW

CHRIS THIEL, OFMCAP

SFHS APCS 2013

TEST

About one fourth of the AP exam will be on Gridworld (5 to 10 multiple-choice questions, one free response question)

You must be familiar with the Bug, BoxBug, Critter, and ChameleonCritter classes (including their implementation)

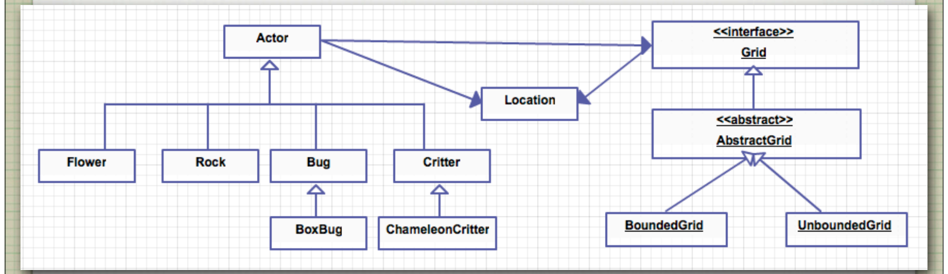
Know the documentation for location, Actor, Rock and Flower classes, as well as the Grid interface

You'll have the Quick Reference: containimg a list of methods for these classes and the source code for Bug,BoxBug, Critter and ChameleonCritter classes

WHAT IS TESTABLE?

- ☐ THE IMPLEMENTATION OF THE CLASS IS TESTABLE.
- ☐ YOU NEED TO KNOW ALL THE MEMBERS OF THE CLASS AND ITS FUNCTIONALITY.
- ☐ YOU NEED TO KNOW HOW TO CALL ANY METHOD OF THIS CLASS FROM A CLIENT PROGRAM SEGMENT.
- ☐ YOU UNDERSTAND THE IMPLEMENTATION CODE OF ANY METHODS OF THE CLASS.
- ☐ YOU ARE EXPECTED TO ALTER THE PROGRAM CODE OF THE CLASS TO ALTER ITS BEHAVIOR
- ☐ YOU NEED TO KNOW ALL THE MEMBERS OF THE CLASS AND ITS FUNCTIONALITY (KNOW THE API!)

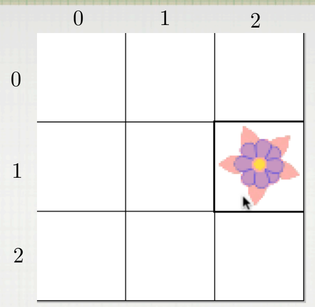
THE CLASSES



ROCKS

- ☐ DO NOTHING
- ☐ SEE APPENDIX E
- ☐ APPENDIX B4

```
public class Rock extends Actor
{

    private static final Color DEFAULT_COLOR = Color.BLACK;

    /**
     * Constructs a black rock.
     */
    public Rock()
    {
        setColor(DEFAULT_COLOR);
    }

    /**
     * Constructs a rock of a given color.
     * @param rockColor the color of this rock
     */
    public Rock(Color rockColor)
    {
        setColor(rockColor);
    }

    /**
     * Overrides the <code>act</code> method in the <code>Actor</code> class
     * to do nothing.
     */
    public void act()
    {
    }
}
```

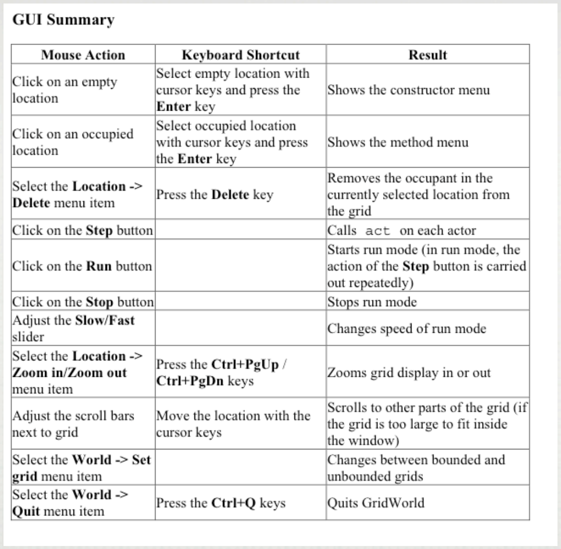


FLOWERS

- ☐ DARKEN IN COLOR- SEE APPENDIX B4

CODE NOT IN APPENDIX (BLACK BOX)

```
public void act()
    {
        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));
    }
```



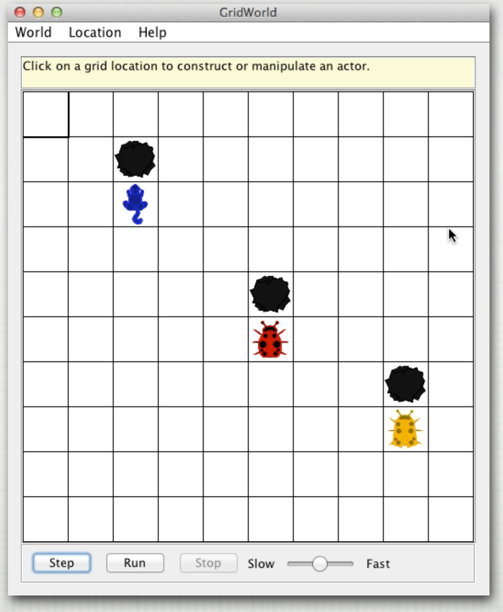
GUI Summary

Mouse Action	Keyboard Shortcut	Result
Click on an empty location	Select empty location with cursor keys and press the **Enter** key	Shows the constructor menu
Click on an occupied location	Select occupied location with cursor keys and press the **Enter** key	Shows the method menu
Select the **Location** -> **Delete** menu item	Press the **Delete** key	Removes the occupant in the currently selected location from the grid
Click on the **Step** button		Calls `act` on each actor
Click on the **Run** button		Starts run mode (in run mode, the action of the **Step** button is carried out repeatedly)
Click on the **Stop** button		Stops run mode
Adjust the **Slow/Fast** slider		Changes speed of run mode
Select the **Location** -> **Zoom in/Zoom out** menu item	Press the **Ctrl+PgUp** / **Ctrl+PgDn** keys	Zooms grid display in or out
Adjust the scroll bars next to grid	Move the location with the cursor keys	Scrolls to other parts of the grid (if the grid is too large to fit inside the window)
Select the **World** -> **Set grid** menu item		Changes between bounded and unbounded grids
Select the **World** -> **Quit** menu item	Press the **Ctrl+Q** keys	Quits GridWorld

GUI

BUG

- ☐ TRIES TO GO FORWARD, LEAVES A FLOWER IN OLD LOCATION, EATS (REMOVES) FLOWER IN NEW LOCATION
- ☐ IF BLOCKED TURN RIGHTS 45° (NON-FLOWERS BLOCK)
- ☐ CODE IN APPENDIX ON PAGES C1-C2



Click on a grid location to construct or manipulate an actor.

BOX BUG

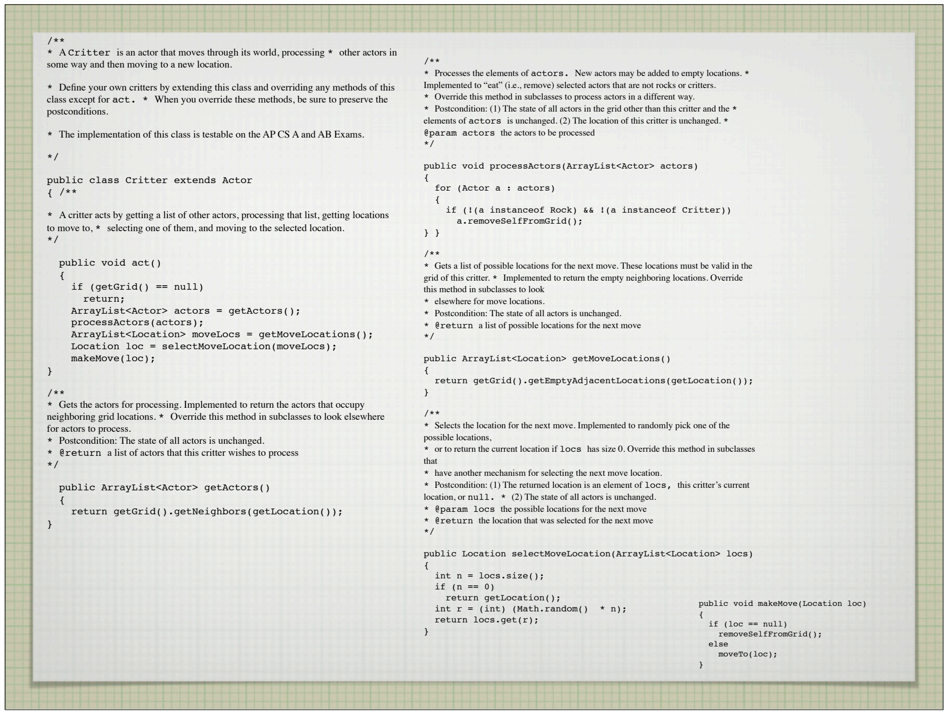
- ☐ MOVES LIKE BUG, BUT TURNS 90°,
- ☐ MAKES A BOX AFTER A GIVEN NUMBER OF TURNS TO LEAVE BEHIND A SQUARE IF IT CAN
- ☐ IF BLOCKED, TURNS TWICE TO RIGHT AND STARTS AGAIN
- ☐ CODE IN APPENDIX ON PAGE C3

PAGE C3: BOX BUG

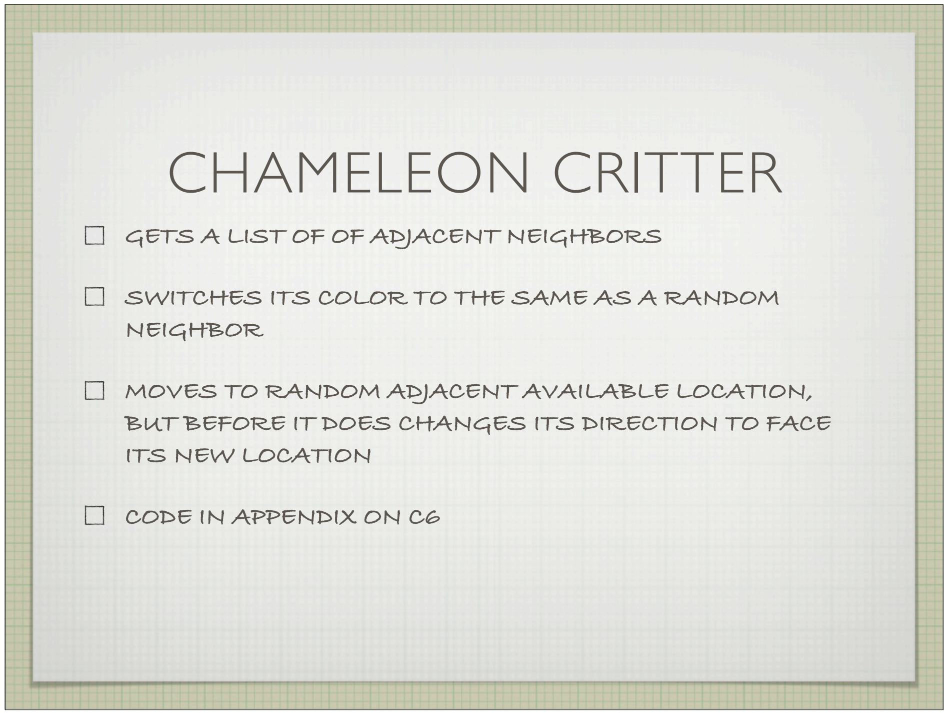
```java
import info.gridworld.actor.Bug;

/**
 * A <code>BoxBug</code> traces out a square "box" of a given size. <br />
 * 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;
        }
    }
}
```

CRITTER

- ☐ GETS A LIST OF OF ADJACENT LOCATIONS
- ☐ EATS EACH FLOWER OR BUG
- ☐ MOVES TO RANDOM ADJACENT
- ☐ IF NONE EMPTY IT DOESNT MOVE (?OR TURN? CHECK!)
- ☐ CODE IN APPENDIX ON PAGES C4-C6

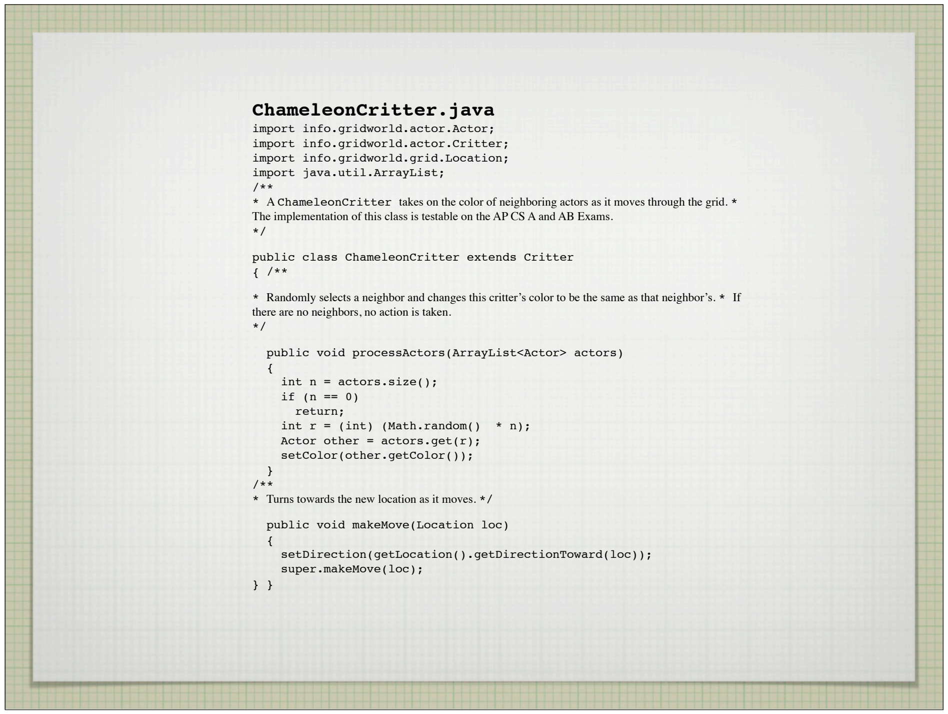


```
/**
* A Critter is an actor that moves through its world, processing * other actors in
some way and then moving to a new location.

* 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);
}

public void makeMove(Location loc)
{
  if (loc == null)
    removeSelfFromGrid();
  else
    moveTo(loc);
}
```



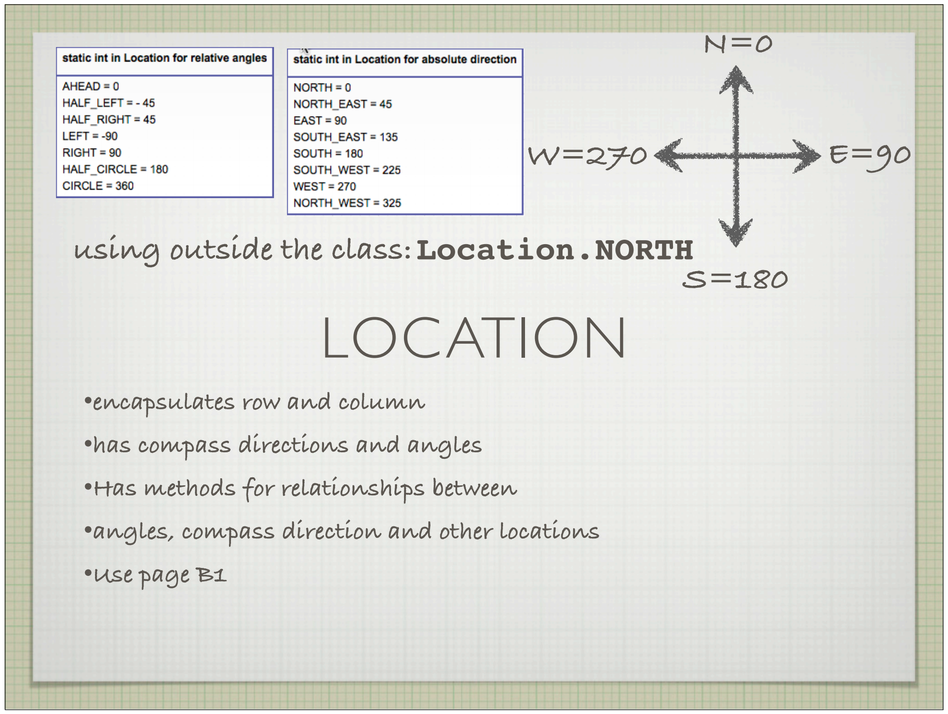
CHAMELEON CRITTER

- ☐ GETS A LIST OF OF ADJACENT NEIGHBORS
- ☐ SWITCHES ITS COLOR TO THE SAME AS A RANDOM NEIGHBOR
- ☐ MOVES TO RANDOM ADJACENT AVAILABLE LOCATION, BUT BEFORE IT DOES CHANGES ITS DIRECTION TO FACE ITS NEW LOCATION
- ☐ CODE IN APPENDIX ON C6



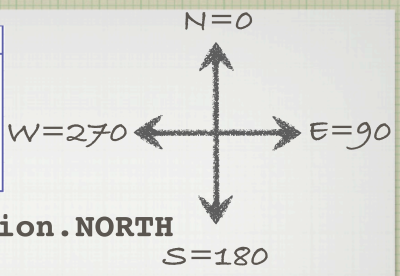
ChameleonCritter.java

```
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);
} }
```



static int in Location for relative angles
AHEAD = 0
HALF_LEFT = - 45
HALF_RIGHT = 45
LEFT = -90
RIGHT = 90
HALF_CIRCLE = 180
CIRCLE = 360

static int in Location for absolute direction
NORTH = 0
NORTH_EAST = 45
EAST = 90
SOUTH_EAST = 135
SOUTH = 180
SOUTH_WEST = 225
WEST = 270
NORTH_WEST = 325



using outside the class: **Location.NORTH**

LOCATION

- •encapsulates row and column
- •has compass directions and angles
- •Has methods for relationships between
- •angles, compass direction and other locations
- •Use page B1

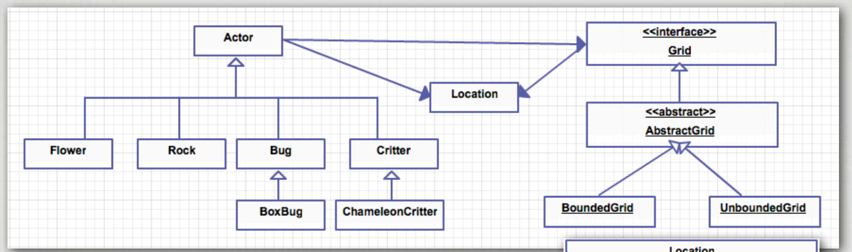
change direction to

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

the current direction

+ 90

EXAMPLE



static int in Location for absolute direction
NORTH = 0
NORTH_EAST = 45
EAST = 90
SOUTH_EAST = 135
SOUTH = 180
SOUTH_WEST = 225
WEST = 270
NORTH_WEST = 325

static int in Location for relative angles
AHEAD = 0
HALF_LEFT = - 45
HALF_RIGHT = 45
LEFT = -90
RIGHT = 90
HALF_CIRCLE = 180
CIRCLE = 360

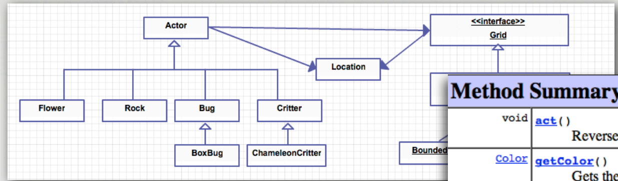
Location

Properties
- row : int
- col : int

Constrctors
Location()
Location(row, col)
Location(Location)

Methods
+ getRow (): int
+ getCol () : int
+ getAdjacentLocation (Direction) : Location
+ getDirectionToward (Location) : Direction
+ equals(Location) : boolean
+ compareTo(Location) : int
+ toString () : String

LOCATION



PRECONDITIONS

1. ACTOR IN A GRID
2. NEWLOC IS VALID IN THAT GRID

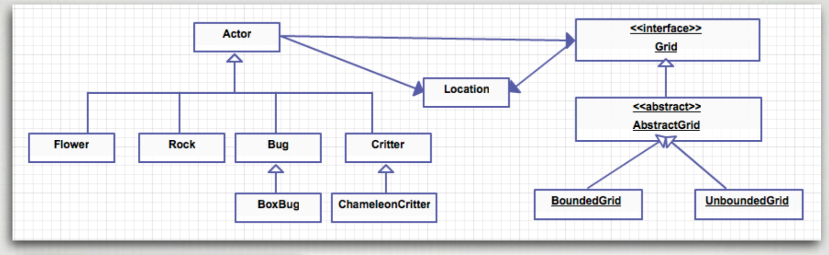
1. NOT IN A GRID,
2. LOC IS VALID

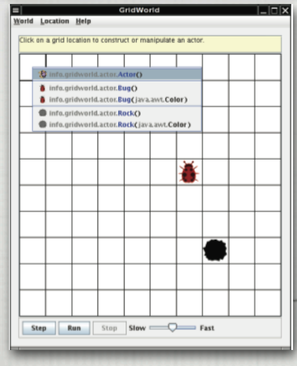
1. ACTOR IN A GRID

Method Summary

void	`act()` Reverses the direction of this actor.
Color	`getColor()` Gets the color of this actor.
int	`getDirection()` Gets the current direction of this actor.
Grid<Actor>	`getGrid()` Gets the grid in which this actor is located.
Location	`getLocation()` Gets the location of this actor.
void	`moveTo(Location newLocation)` Moves this actor to a new location.
void	`putSelfInGrid(Grid<Actor> gr, Location loc)` Puts this actor into a grid.
void	`removeSelfFromGrid()` Removes this actor from its grid.
void	`setColor(Color newColor)` Sets the color of this actor.
void	`setDirection(int newDirection)` Sets the current direction of this actor.
String	`toString()` Creates a string that describes this actor.

ACTOR - B3





GridWorld
World
Location
Help
Click on a grid location to construct or manipulate an actor.
info.gridworld.actor.Actor()
info.gridworld.actor.Bug()
info.gridworld.actor.Bug(java.awt.Color)
info.gridworld.actor.Rock()
info.gridworld.actor.Rock(java.awt.Color)
Step
Run
Stop
Slow
Fast

Method Summary

void	`act()` Moves if it can move, tu
boolean	`canMove()` Tests whether this bug can move forward into a location that is empty or contains a flower.
void	`move()` Moves the bug forward, putting a flower into the location it previously occupied.
void	`turn()` Turns the bug 45 degrees to the right without changing its location.

1. override **act()** ok for bug
2. **super.act()** for default

BUG- C1-2

When adding or removing actors, do *not* use the `put` and `remove` methods of the `Grid` interface. Those methods do not update the `location` and `grid` instance variables of the actor. That is a problem since most actors behave incorrectly if they do not know their location. To ensure correct actor behavior, always use the `putSelfInGrid` and `removeSelfFromGrid` methods of the `Actor` class.

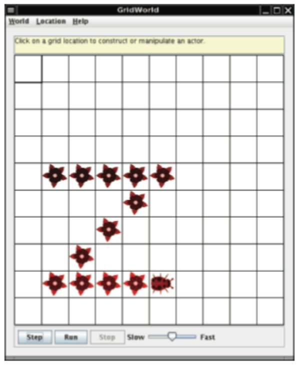
To Make Different BUGS:

Override the **act()** method

- **moveTo(),**
- **setColor()**
- **setDirection()**
- **putSelfInGrid()**
- **removeSelfFromGrid()**

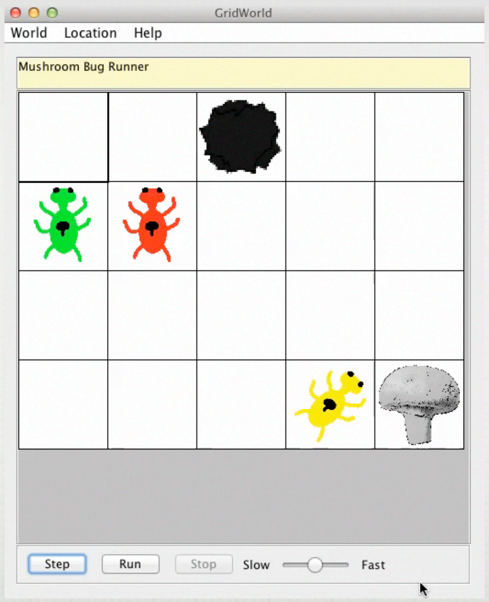
BUG SUBCLASSES

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.

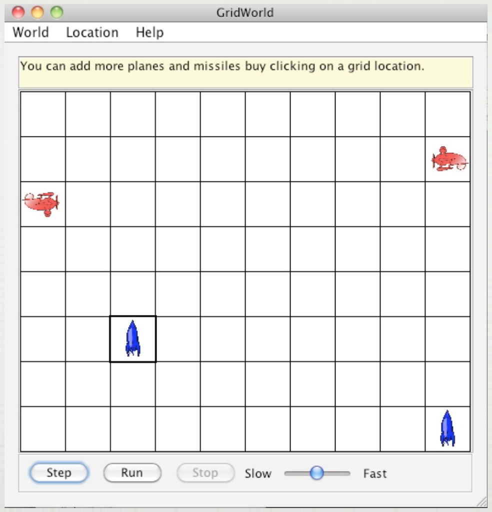


Click on a grid location to construct or manipulate an actor.

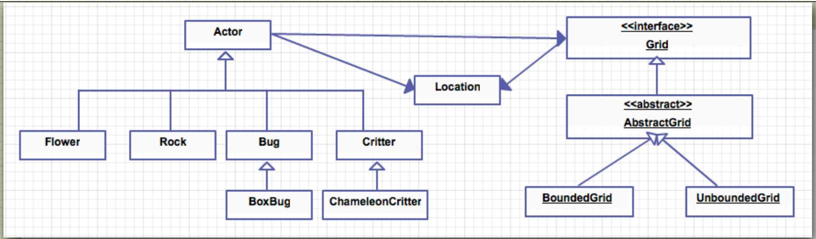
Learn the methods so you can make new sub classes



Mushroom Bug Runner



You can add more planes and missiles buy clicking on a grid location.



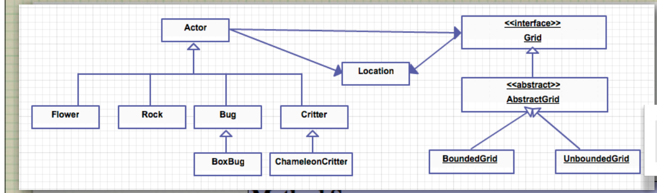
Adds two attributes

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

overwrites one method

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

BOXBUG - C3



1. DO NOT CHANGE STATE OF other ACTORS!

Method Summary

void	`act()` 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.
`ArrayList<Actor>`	`getActors()` Gets the actors for processing.
ion>	`getMoveLocations()` Gets a list of possible locations for the next move.
void	`makeMove(Location loc)` Moves this critter to the given location `loc` or removes this critter from its grid if `loc` is `null`.
void	`processActors(ArrayList<Actor> actors)` Processes the elements of `actors`.
`Location`	`selectMoveLocation(ArrayList<Location> locs)` Selects the location for the next

1. DO NOT TOUCH CRITTER'S **act()** METHOD!!!

1. CAN ONLY CHANGE THE STATE OF ACTORS IN OLD/NEW LOCATION
2. USE **moveTo(loc)** TO UPDATE STATE
3. **if (loc==null) removeSelfFromGrid()**

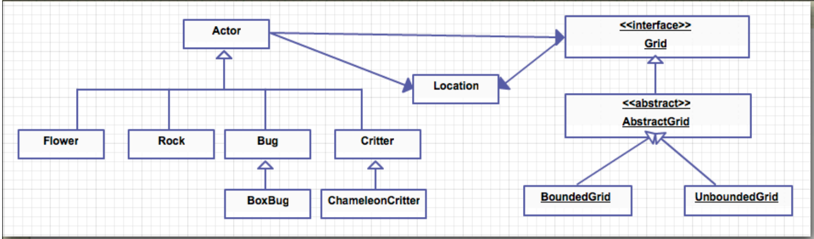
1. OK CHANGE STATE OF THIS CRITTER ONLY!
2. ACTORS PARAM UNCHANGED
3. LOC OF THIS CRITTER UNCHANGED

1. RETURN: ELEMENT OF (LOCS)
2. OR CURRENT LOCATION
3. OR NULL

CRITTER: C4-5

`getActors`	The state of all actors is unchanged.
`processActors`	(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.
`getMoveLocations`	The state of all actors is unchanged.
`selectMoveLocation`	(1) The returned location is an element of `locs`, this critter's current location, or `null`. (2) The state of all actors is unchanged.
`makeMove`	(1) `getLocation() == loc`. (2) The state of all actors other than those at the old and new locations is unchanged.

CRITTER POSTCONDITIONS



Overrides two methods

```
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);
}
```

CHAMELEONCRITTER

Method Summary

E	**get**(Location loc) Returns the object at a given location in this gri
ArrayList<Location>	**getEmptyAdjacentLocations**(Location loc) Gets the valid empty locations adjacent to a giv compass directions (north, northeast, east, southeast, s northwest).
ArrayList<E>	**getNeighbors**(Location loc) Gets the neighboring occupants in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
int	**getNumCols**() Returns the number of columns in this grid.
int	**getNumRows**() Returns the number of rows in this grid.
ArrayList<Location>	**getOccupiedAdjacentLocations**(Location loc) Gets the valid occupied locations adjacent to a given location in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
ArrayList<Location>	**getOccupiedLocations**() Gets the locations in this grid that contain objects.
ArrayList<Location>	**getValidAdjacentLocations**(Location loc) Gets the valid locations adjacent to a given location in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
boolean	**isValid**(Location loc) Checks whether a location is valid in this grid.
E	**put**(Location loc, E obj) Puts an object at a given location in this grid.
E	**remove**(Location loc) Removes the object at a given location from this grid.

Method	BoundedGrid	UnboundedGrid
getNeighbors	$O(1)$	$O(1)$
getValidAdjacentLocations	$O(1)$	$O(1)$
getEmptyAdjacentLocations	$O(1)$	$O(1)$
getOccupiedAdjacentLocations	$O(1)$	$O(1)$
toString	$O(rc)$	$O(n)$
getOccupiedLocations	$O(rc)$	$O(n)$
get	$O(1)$	$O(1)$
put	$O(1)$	$O(1)$
remove	$O(1)$	$O(1)$

GRID<E>- PAGE B2

put

```
E put(Location loc,
      E obj)
```

Puts an object at a given location in this grid.
Precondition: (1) `loc` is valid in this grid (2) `obj` is not `null`

Parameters:
`loc` - the location at which to put the object
`obj` - the new object to be added

Returns:
the object previously at `loc` (or `null` if the location was previously unoccupied)

But usually from actors don't call **put** ...instead:

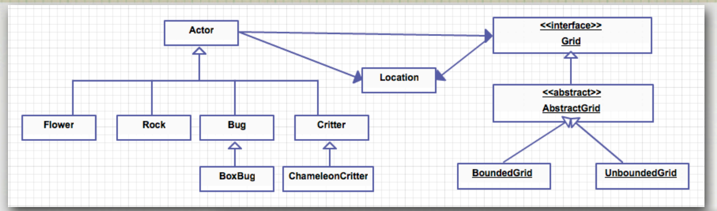
```
if (gr.isValid(next))
  moveTo(next);
else
  removeSelfFromGrid();
Flower flower = new Flower(getColor());
flower.putSelfInGrid(gr, loc);
```

n if the

```
public void processActors(ArrayList<Actor> actors)
{
  for (Actor a : actors)
  {
    if (!(a instanceof Rock) && !(a instanceof Critter))
      a.removeSelfFromGrid();
  }
}
```

is null

RID: PUT



Constructor Summary

AbstractGrid()

Method Summary

ArrayList<Location>	**getEmptyAdjacentLocations**(Location loc) Gets the valid empty locations adjacent to a given location in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
ArrayList<E>	**getNeighbors**(Location loc) Gets the neighboring occupants in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
ArrayList<Location>	**getOccupiedAdjacentLocations**(Location loc) Gets the valid occupied locations adjacent to a given location in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
ArrayList<Location>	**getValidAdjacentLocations**(Location loc) Gets the valid locations adjacent to a given location in all eight compass directions (north, northeast, east, southeast, south, southwest, west, and northwest).
String	**toString**() Creates a string that describes this grid.

INTERFACE AND ABSTRACT CLASSES

TO MAKE DIFFERENT CRITTERS:

NEVER OVERRIDE THE **act()** METHOD!

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

CRITTERS

Usually you need info from the grid:

```
        Grid gr = anActor.getGrid();
        Grid gr = this.getGrid();
```

getting occupied locations returns **Locations** not **Actors**!

```
ArrayList<Location> locs = gr.getOccupiedLocations();
ArrayList<Actor> actors = new ArrayList<Actor>();
for (Location loc:locs)
{
      actors.add( gr.get(loc) );
}
```

getActors/processActors NOTES

```
public ArrayList<Location> getMoveLocations()
```

Usually you need info from the grid:

```
        Grid gr = anActor.getGrid();
        Grid gr = this.getGrid();
```

check if its valid first!

```
if( gr.isValid(loc) && gr.get(loc)==null   )...

   if( ! gr.isValid(loc)    )
         return;

if( gr.isValid(loc)    )
        Location next = loc.getAdjacentLocation(Direction.NORTH);
if (gr.isValid(next) )
        locs.add(next);
```

getMoveLocations NOTES

```
public Location selectMoveLocation(ArrayList<Location> locs)
```

if a condtion requires "default" behavior:

```
    if (something==true)
          return super.selectMoveLocation(locs);
```

random from the `locs ArrayList<Location>`

```
    int rand = (int)(locs.size()*Math.random() );
    Location loc=locs.get(rand);
```

To die: don't **removeSelfFromGrid**-it changes state

```
    return null;
```

if you cant move, and want to live:

```
    return this.getLocation();
```

Critter's **SelectMoveLocation** NOTES

Know how each of the actors move and act

Know the inheritance relationships between the actors

Know how to write subclasses of bug or critter and how to modify their default methods

Know how to use the quick reference

SUMMARY