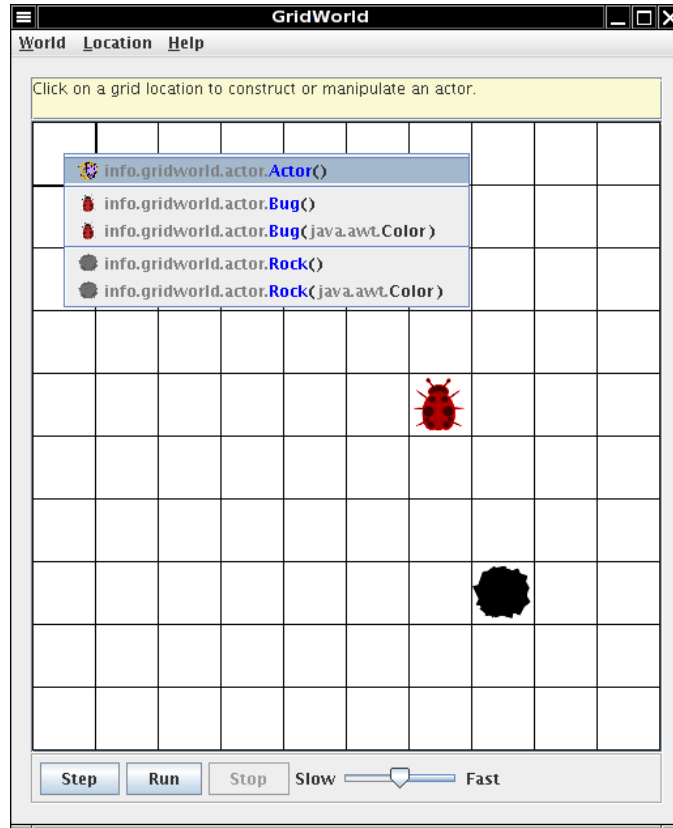
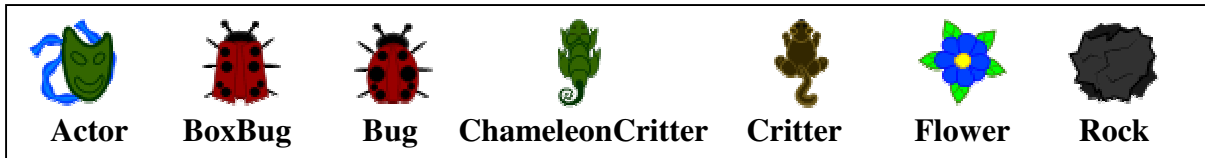


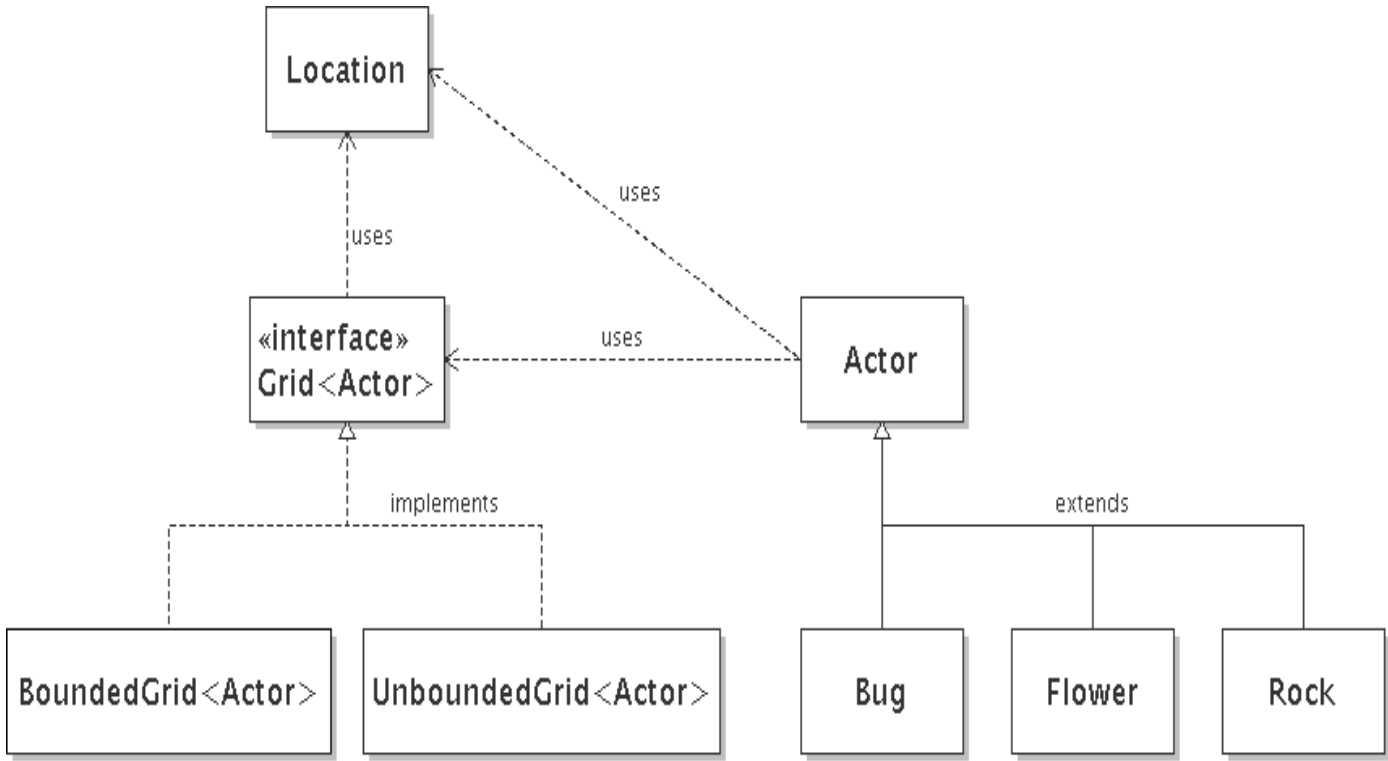
AP Programming – GridWorld Lecture

Below are the actors and their icons that you will need to know for the AP Test:



AP Programming – GridWorld Lecture

Some GridWorld Classes and Interfaces:



Testable GridWorld Classes and Interfaces:

- info.gridworld.grid
 - Location Class
 - info.gridworld.grid package
 - implements Comparable
 - represents the row and column of a location in a two-dimensional grid.
 - Location Class Methods
 - public Location(int r, int c)
 - public int getRow()
 - public int getCol()
 - public Location getAdjacentLocation(int direction)
 - public int getDirectionToward(Location target)
 - public boolean equals(Object other)
 - public int hashCode()
 - public int compareTo(Object other)
 - public String toString()
 - Location Class Constants
 - NORTH, EAST, SOUTH, WEST,
 - NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST,
 - LEFT, RIGHT,
 - HALF_LEFT, HALF_RIGHT,
 - FULL_CIRCLE, HALF_CIRCLE,
 - AHEAD

Testable GridWorld Classes and Interfaces continued:

- info.gridworld.grid
 - Grid<E> (interface)
 - info.gridworld.grid package
 - Implementing Classes:
 - AbstractGrid
 - BoundedGrid
 - UnboundedGrid
 - Grid provides an interface for a two-dimensional, grid-like environment containing arbitrary objects.
 - Grid<E> (interface) Methods
 - int getNumRows()
 - int getNumCols()
 - boolean isValid(Location loc)
 - E put(Location loc, E obj) // returns prev obj or null
 - E remove(Location loc) // returns prev obj or null
 - E get(Location loc) // returns curr obj or null
 - ArrayList<Location> getOccupiedLocations()
 - ArrayList<Location> getValidAdjacentLocations(Location loc)
 - ArrayList<Location> getEmptyAdjacentLocations(Location loc)
 - ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
 - ArrayList<E> getNeighbors(Location loc)

Testable GridWorld Classes and Interfaces continued:

- info.gridworld.actor
 - Actor
 - info.gridworld.actor package
 - Direct Subclasses:
 - Bug, Critter, Flower, Rock
 - An Actor is an entity with a color and direction that can act.
 - Actor Class Constructor and Methods
 - public Actor()
 - public Color getColor()
 - public void setColor(Color newColor)
 - public int getDirection()
 - public void setDirection(int newDirection)
 - public Grid<Actor> getGrid()
 - public Location getLocation()
 - public void putSelfInGrid(Grid<Actor> gr, Location loc)
 - public void removeSelfFromGrid()
 - public void moveTo(Location newLocation)
 - public void act()
 - public String toString()
 - Rock
 - info.gridworld.actor package
 - Extends Actor
 - Rock Class Constructors and Methods
 - public Rock()
 - public Rock(Color rockColor)
 - public void act()

A Rock is an actor that does nothing.
It is commonly used to block other actors from moving.
 - Flower
 - info.gridworld.actor package
 - Extends Actor
 - Flower Class Constructors and Methods
 - public Flower()
 - public Flower(Color initialColor)
 - public void act()

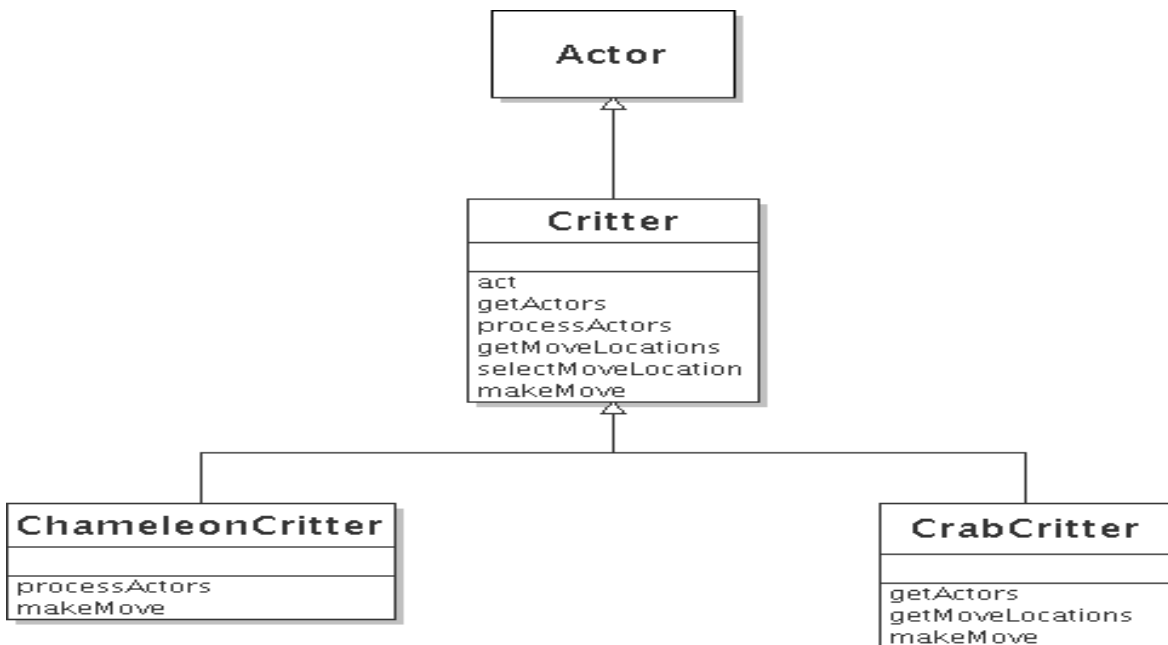
A Flower is an actor that darkens over time.

AP Programming – GridWorld Lecture

Testable GridWorld Classes and Interfaces continued:

- info.gridworld.actor
 - Bug
 - info.gridworld.actor package
 - Extends Actor
 - Bug Class Constructors and Methods
 - public Bug()
 - public Bug(Color bugColor)
 - public void turn()
 - public void move()
 - public boolean canMove()
 - public void act()

A Bug is an actor that can move and turn.
It drops flowers as it moves.
 - BoxBug (extends Bug)
 - Not in a gridworld package
 - Extends Bug
 - BoxBug Class Constructors and Methods
 - public BoxBug(int length)
 - public void act()
 - Critters:



AP Programming – GridWorld Lecture

- info.gridworld.actor package
- Extends Actor
- Critters are included in GridWorld to emphasize design
- Critters Class Methods
 - public void act()

A Critter is an actor that moves through its world, processing other actors in some way and then picking a new location.

The act() method calls five methods, and subclasses override some or all of them to achieve some desired behavior.

Ideally, act() should have been declared final, but that is not in the AP CS subset.

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

Each of the five methods called from act() has postconditions that restrict what it can do. The following table shows the postconditions as they are stated in the GridWorld documentation.

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.

Here, the *state* of an actor includes

- the location
- the direction
- the contents of any other instance variables

Removing an actor changes its state: its location becomes null. (Do not try to weasel out of that by calling `grid.remove(anActor)` - see Grid Section page 4.)

If a critter wants to change its own state, it can only do so in `processActors` and `makeMove`. If it wants to change its location (or remove itself), it can only do so in `makeMove`.

If a critter wants to change the state of another critter, it can only do so in `processActors`. (There is a teensy exception: `makeMove` removes the critter at the target location, and it may add a new critter to the old location.)

Note that `getActors`, `getMoveLocations`, and `selectMoveLocation` can't change the state of *any* actor.

Also note that `processActors` can only mutate actors that were handed to it, and `selectMoveLocations` can only select from the locations that were handed to it.

The `makeMove` method has even less choice - it *must* move to the location given in the parameter.

These conditions significantly restrict what a critter can do, and how it can do it. For example, consider a `BluesCritter` that chooses a blue rock in the grid, eats the chosen rock's neighbors, and jumps to the rock (thereby removing it from the grid). The first method called by `act` is the `getActors` method. We must implement that method to return a set that includes the blue rock's neighbors since `processActors` can only remove actors that are given to it. But `getActors` cannot update *any* instance variable of the `BluesCritter`. In particular, it cannot store the blue rock's location. Instead, the `processActors` method must leave some trace of that location so that the `getMoveLocations` method can recall it. To finesse this, we can make the `getActors` method return *all* actors in the grid, so that `processActors` can pick a blue rock, remember its location, and remove its neighbors.

When you design a particular critter, you need to distribute responsibilities among the five methods that are called by `act`. For example, in the `BluesCriticter` class, if `processActors` sets an instance variable to the location of the chosen blue rock, the `getMoveLocations` relies on that setting. Some people are concerned about the apparent fragility of such an arrangement. What would happen if someone else called one of the methods, or if a subclass of `BluesCriticter` changed one but not the other? In the context of the AP CS Exam, students should *not* worry about such issues unless they are specifically directed otherwise. These five methods were intended to be used in the `act` method, and not for any other purpose.

To summarize, here are simple rules for your students when working with critters.

- Don't override `act`
- A critter can change its state only in `processActors` or `makeMove`
- A critter can change the state of other actors in `processActors`

Remember, these rules are *only for critters*, not for bugs or other actors.

Also keep in mind that not every actor can or should be represented as a critter. In the context of `GridWorld`, a critter is not a warm and fuzzy creature, but an actor that first processes actors and then makes a move. The `BluesCriticter` doesn't fit that description very well, and it would have been better to design it as a `BluesActor`. Then you can override `act` without any tortured logic, simply moving to a blue rock and removing its neighbors.

- Default package
 - `ChameleonCriticter`
 - Not in a `gridworld` package
 - Extends `Criticter`
 - `ChameleonCriticter` Class Methods
 - `public void processActors(ArrayList<Actor> actors)`
 - `public void makeMove(Location loc)`

More about the Grid

In a grid of actors, you must *always* use the `putSelfInGrid/removeSelfFromGrid` in the Actor class, and never the `put/remove` method of the Grid class. These methods ensure that the Grid reference in each Actor object properly refers to the grid containing the actor. In particular, don't try calling `getGrid().remove(this)` in an attempt to bypass the postcondition of a Critter method. The mere fact that the resulting code may happen to “work” when you run it in the GridWorld environment does not make it right.

Of course, if you have a grid of other objects that are not actors, then use `put` and `remove`.

Finally, it is easy to be confused about valid and empty locations. There are two simple rules.

- `null` is never a location. Do not pass it to any of the Grid methods, not even `isValid`.
- *All* methods other than `isValid` require a valid location.

What happens if you pass `null` or an invalid location to a grid method? If you *know* that the grid is a `BoundedGrid` or `UnboundedGrid`, then you can consult the implementation and see that a `NullPointerException` or `IllegalArgumentException` is thrown. But for a general `Grid`, you have no way of knowing what happens - someone could have implemented the `Grid` interface in a different way.

“Runner” Sample Application:

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