Notes on the GridWorld API

This note discusses two aspects of the design of the GridWorld API. These observations may be helpful for teachers and students as they prepare for the AP CS Exam.

A. Critter Methods

Critters are included in GridWorld to emphasize design. (For those of you familiar with OO design patterns, the `act` method is an example of the "template method" design pattern. But you do not need to know that pattern to use critters. Design patterns are definitely not a part of the AP CS curriculum.)

The `act` method calls five methods, and subclasses override some or all of them to achieve some desired behavior. Ideally, `act` would have been declared `final`, but that is not in the AP CS subset.

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`. (By the way, don't try to weasel out of that by calling `grid.remove(anActor)`—see Part B below.) Similarly, adding an actor changes its state.

Note that `getActors`, `getMoveLocations`, and `selectMoveLocation` can't change the state of *any* actor. The postcondition of `processActors` is subtly worded to allow adding new actors, since it only prohibits state change of actors *in the grid*.

If a critter wants to change *its own state*, it must do so in `processActors` or `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 actor*, it can only do so in `processActors`, and the actor must have been included in the `actors` parameter, or it must be newly added to the grid. (There is a teensy exception: `makeMove` removes the actor at the target location, and it may add a single actor, but only to the old location.)

Finally, `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 `BluesCritter` 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 `BluesCritter` 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 `BluesCritter` 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.

B. 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 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.