Apply the question-specific rubric first; the question-specific rubric always takes precedence.

**Penalties:** The penalty categorization below is for cases not covered by the question-specific rubric. Points can only be deducted in a part of the question that has earned credit via the question-specific rubric, and no section may have a negative point total. A given penalty can be assessed only once in a question, even if it occurs on different parts of that question. A maximum of 3 penalty points may be assessed over the entire question.

### Nonpenalized Errors
- spelling/case discrepancies if no ambiguity*
- local variable not declared if other variables are declared in some part
- use of keyword as identifier
- [] vs. () vs. <>
- = instead of == (and vice versa)
- length/size confusion for array, String, and ArrayList, with or without ()
- private qualifier on local variable
- extraneous code with no side effect; e.g., precondition check
- common mathematical symbols for operators (x • ÷ < > ≤ ≥)
- missing {} where indentation clearly conveys intent and {} used elsewhere
- default constructor called without parens; e.g., new Critter;
- missing ( ) on parameter-less method call
- missing ( ) around if/while conditions
- missing ; when majority are present
- missing public on class or constructor header
- extraneous [] when referencing entire array
- [i,j] instead of [i][j]
- extraneous size in array declaration, e.g., int[size] nums = new int[size];

* Spelling and case discrepancies for identifiers fall under the “nonpenalized” category only if the correction can be unambiguously inferred from context; for example, “ArayList” instead of “ArrayList”. As a counterexample, note that if a student declares “Bug bug;” then uses “Bug.move()” instead of “bug.move()”, the context does not allow for the reader to assume the object instead of the class.

### Minor Errors (½ point)
- confused identifier (e.g., len for length or left() for getLeft())
- local variables used but none declared
- missing new in constructor call
- modifying a constant (final)
- use of equals or compareTo method on primitives, e.g., int x; ...x.equals(val)
- array/collection access confusion ([] get)
- assignment dyslexia, e.g., x + 3 = y; for y = x + 3;
- super(method()) instead of super.method()
- formal parameter syntax (with type) in method call, e.g., a = method(int x)
- missing public from method header when required
- “false”/”true” or 0/1 for boolean values
- “null” for null

### Major Errors (1 point)
- extraneous code that causes side effect; e.g., information written to output
- interface or class name instead of variable identifier; e.g., Bug.move() instead of aBug.move()
- aMethod(obj) instead of obj.aMethod()
- attempt to use private data or method when not accessible
- destruction of persistent data (e.g., changing value referenced by parameter)
- use of class name in place of super in constructor or method call
- void method (or constructor) returns a value

#### Applying Minor Penalties (½ point):
A minor infraction that occurs exactly once when the same concept is correct two or more times is regarded as an oversight and not penalized. A minor penalty must be assessed if the item is the only instance, one of two, or occurs two or more times.
Question 2: Attractive Critter (GridWorld)

| Class: AttractiveCritter | 9 points |

**Intent:** Define extension to Critter class that relocates all other actors closer to itself

**+1** Properly formed class header for AttractiveCritter that extends Critter class

**+2½** Override Critter methods and maintain all postconditions
  **+1** Overrides at least one method of Critter and satisfies all postconditions *(point not awarded if also overrides act method)*
  **+½** Overrides getActors
  **+1** Overrides processActors

**+5½** Move other actors in grid to be closer to self
  **+1** Considers all other actors in grid
  **+½** Checks for an empty movement destination
  **+1½** Moves an actor
    **+½** Moves at least one other actor to different location in grid
    **+1** Moves another actor and guards against inappropriate self-movement
  **+1½** Determines correct direction and location
    **+½** Determines correct direction toward self for at least one other actor
    **+1** Determines adjacent location to at least one other actor *(point awarded only if calculated direction is used as parameter)*
  **+1** Moves all other actors to calculated destinations

**Question-Specific Penalties**

**-1** Inappropriate state change in world (Grid, Actor,...)
Question 2: Attractive Critter (GridWorld)

Solution that checks for self in `getActors`

```java
public class AttractiveCritter extends Critter {
    public ArrayList<Actor> getActors() {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        for (Location loc : getGrid().getOccupiedLocations()) {
            if (!loc.equals(this.getLocation())) {
                actors.add(getGrid().get(loc));
            }
        }
        return actors;
    }
}
```

```java
public void processActors(ArrayList<Actor> actors) {
    for (Actor a : actors) {
        int direction = (a.getLocation()).getDirectionToward(this.getLocation());
        Location newLoc = (a.getLocation()).getAdjacentLocation(direction);
        if (getGrid().get(newLoc) == null) {
            a.moveTo(newLoc);
        }
    }
}
```

Solution that checks for self in `processActors`

```java
public class AttractiveCritter extends Critter {
    public ArrayList<Actor> getActors() {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        for (Location loc : getGrid().getOccupiedLocations()) {
            actors.add(getGrid().get(loc));
        }
        return actors;
    }
}
```

```java
public void processActors(ArrayList<Actor> actors) {
    for (Actor a : actors) {
        if (a != this) {
            int direction = (a.getLocation()).getDirectionToward(this.getLocation());
            Location newLoc = (a.getLocation()).getAdjacentLocation(direction);
            if (getGrid().get(newLoc) == null) {
                a.moveTo(newLoc);
            }
        }
    }
}
```
The order in which the actors in the grid are processed is not specified, making it possible to get different results from the same grid of actors.

Write the complete `AttractiveCritter` class, including all instance variables and required methods. Do NOT override the `act` method. Remember that your design must not violate the postconditions of the methods of the `Critter` class and that updating an object's instance variable changes the state of that object.

```java
public class AttractiveCritter extends Critter {
    public ArrayList<Actor> getActors() {
        ArrayList<Location> occLocs = getGrid().getOccupiedLocations();
        ArrayList<Actor> a = new ArrayList<Actor>;
        for (Location loc : occLocs) {
            a.add(getGrid().get(loc));
        }
        return a;
    }

    public void processActors(ArrayList<Actor> actors) {
        int direct;
        Location loc;
        for (Actor a : actors) {
            direct = a.getLocation().getDirectionToward(getLocation());
            loc = a.getLocation().getAdjacentLocation(direct);
            if (getGrid().get(loc) == null)
                a.moveTo(loc);
        }
    }
}
```

GO ON TO THE NEXT PAGE.
The order in which the actors in the grid are processed is not specified, making it possible to get different results from the same grid of actors.

Write the complete `AttractiveCritter` class, including all instance variables and required methods. Do NOT override the `act` method. Remember that your design must not violate the postconditions of the methods of the `Critter` class and that updating an object's instance variable changes the state of that object.

```java
public class AttractiveCritter extends Critter {

    public void processActors(ArrayList<Actor> actors)
    {
        for (Actor a : actors)
        {
            Location move = a.getLocation().getAdjacentLocation(a.getDirectionTowards(this).getLocation());
            if (a.getGrid().get(move) == null)
                a.moveTo(move);
        }
    }
```

GO ON TO THE NEXT PAGE.
The order in which the actors in the grid are processed is not specified, making it possible to get different results from the same grid of actors.

Write the complete `AttractiveCritter` class, including all instance variables and required methods. Do **NOT** override the `act()` method. Remember that your design must not violate the postconditions of the methods of the `Critter` class and that updating an object's instance variable changes the state of that object.

```java
public class AttractiveCritter extends Critter {
  public ArrayList<Actor> getActors() {
    ArrayList<Actor> x = new ArrayList<Actor>();
    x.add(get(getOccupiedLocations()));
    return x;
  }

  public void processActors(ArrayList<Actor> actors) {
    for (Actor a : actors)
      a.setDirection(getDirectionTowards(getLocation()));
    if (getAdjacentLocation(a.getDirection()) == null)
      a.moveTo(getAdjacentLocation(a.getDirection()));
  }
}
```
Question 2

Overview

This question involved the design of a complete class within the setting of the GridWorld case study. Students were asked to design and code the AttractiveCritter class. An attractive critter was described as a critter that processed other actors by attempting to relocate all of the other actors in the grid, including other attractive critters, one grid cell closer to itself in the direction specified by getDirectionToward. The question tested class definition and construction, method implementation, and knowledge of the GridWorld case study. Students were instructed to write the complete class, including all instance variables and required methods. They were cautioned NOT to override the act method nor violate any postconditions of methods in the Critter class.

Sample: 2A
Score: 8

The student earned the first 3½ points for writing a correct class heading, overriding at least one Critter method (not act and not violating any postconditions), overriding getActors, and overriding processActors.

This response demonstrates the advantages of using local variables and writing clear, readable code to simplify a solution. In getActors the student identifies occupied locations and then creates and returns an array list of the actors in those locations.

In processActors the student correctly determines the direction and location of the new destination for each actor and then checks to be sure the location is empty before making the move. The student did not earn the point for guarding against moving the attractive critter itself. That guard could have been included in getActors when the array list of actors is created, or it could have been included in processActors before the move is made. Overlooking this guard was the most frequent reason an otherwise perfect solution dropped from a score of 9 to a score of 8.

Sample: 2B
Score: 6

The student earned 3 of the first 3½ points. There is a correct class header, an override of at least one Critter method, and an override of processActors.

The student does not override getActors and so did not earn the “considers all actors” point. Unless overridden, getActors returns an array list of only the neighbors of an actor rather than an array list of all actors in the grid.

In processActors the student correctly determines the location but did not earn the ½ point for direction because getDirectionToward is not called on a location. The response earned ½ point for an empty destination check. It also earned 1½ points for moving the actor to a new location in the grid, even though direction is not perfect. There is no guard against moving the attractive critter itself, so the student did not earn that point. This response earned a total of 6 points.
Question 2 (continued)

Sample: 2C
Score: 2.5 (rounded to 3)

The student earned the first 3½ points for writing a correct class heading, overriding at least one Critter method (not act and not violating any postconditions), overriding getActors, and overriding processActors. These points address the design issues of what is necessary for this class and can be earned even if no implementation is included within the methods.

The student includes implementation code but earned no additional points. In getActors the student attempts to create an array list of all actors in the grid but uses incorrect method calls and does not get the actor at each location. There is an attempt in processActors to identify the direction and location of the destination, but these are Location methods and need to be called on the actor’s location. Owing to these incorrect calls, the student did not earn the points for checking for an empty destination, guarding against moving self, and moving one or all actors to a location.

There was a 1-point penalty for inappropriate state change based on the call to setDirection. Moving an attractive critter in a specified direction does not mean the critter turns to face that direction. The score was calculated as 3½ points minus 1 point for a score of 2½ points, which was rounded up to 3 points.