2. This question involves reasoning about the GridWorld case study. Reference materials are provided in the appendixes.

   A `Director` is a type of `Rock` that has the following characteristics.
   - A `Director` has an initial color of `Color.RED` and alternates between `Color.RED` and `Color.GREEN` each time it acts.
   - If the color of a `Director` is `Color.GREEN` when it begins to act, it will cause any `Actor` objects in its neighboring cells to turn 90 degrees to their right.

   Write the complete `Director` class, including the zero-parameter constructor and any necessary instance variables and methods. Assume that the `Color` class has been imported.

3. A student in a school is represented by the following class.

```
public class Student
{
  /** Returns the name of this Student. */
  public String getName()
  {  /* implementation not shown */  }

  /** Returns the number of times this Student has missed class. */
  public int getAbsenceCount()
  {  /* implementation not shown */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

The class `SeatingChart`, shown below, uses a two-dimensional array to represent the seating arrangement of students in a classroom. The seats in the classroom are in a rectangular arrangement of rows and columns.

```
public class SeatingChart
{
  /** seats[r][c] represents the Student in row r and column c in the classroom. */
  private Student[][] seats;

  /** Creates a seating chart with the given number of rows and columns from the students in
   *  studentList. Empty seats in the seating chart are represented by null.
   *  @param rows the number of rows of seats in the classroom
   *  @param cols the number of columns of seats in the classroom
   *  Precondition: rows > 0; cols > 0;
   *                rows * cols >= studentList.size()
   *  Postcondition:
   *     - Students appear in the seating chart in the same order as they appear
   *       in studentList, starting at seats[0][0].
   *     - seats is filled column by column from studentList, followed by any
   *       empty seats (represented by null).
   *     - studentList is unchanged.
   */
  public SeatingChart(List<Student> studentList,
                      int rows, int cols)
  {  /* to be implemented in part (a) */  }

  /** Removes students who have more than a given number of absences from the
   *  seating chart, replacing those entries in the seating chart with null
   *  and returns the number of students removed.
   *  @param allowedAbsences an integer >= 0
   *  @return number of students removed from seats
   *  Postcondition:
   *    - All students with allowedAbsences or fewer are in their original positions in seats.
   *    - No student in seats has more than allowedAbsences absences.
   *    - Entries without students contain null.
   */
  public int removeAbsentStudents(int allowedAbsences)
  {  /* to be implemented in part (b) */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

The menu allows customers to create `Trio` menu items, each of which includes a sandwich, a salad, and a drink. The name of the `Trio` consists of the names of the sandwich, salad, and drink, in that order, each separated by `"/"` and followed by a space and then `"Trio"`. The price of the `Trio` is the sum of the two highest-priced items in the `Trio`; one item with the lowest price is free.

A trio consisting of a cheeseburger, spinach salad, and an orange soda would have the name `"Cheeseburger/Spinach Salad/Orange Soda Trio"` and a price of $4.00 (the two highest prices are $2.75 and $1.25). Similarly, a trio consisting of a club sandwich, coleslaw, and a cappuccino would have the name `"Club Sandwich/Coleslaw/Cappuccino Trio"` and a price of $6.25 (the two highest prices are $2.75 and $3.50).

Write the `Trio` class that implements the `MenuItem` interface. Your implementation must include a constructor that takes three parameters representing a sandwich, salad, and drink. The following code segment should have the indicated behavior.

```
Sandwich sandwich;
Salad salad;
Drink drink;
/* Code that initializes sandwich, salad, and drink */

Trio trio = new Trio(sandwich, salad, drink);   // Compiles without error

Trio trio1 = new Trio(salad, sandwich, drink); // Compile-time error
Trio trio2 = new Trio(sandwich, salad, salad); // Compile-time error
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

AP® COMPUTER SCIENCE A
2014 SCORING GUIDELINES

Question 2: Director

Class: `Director`	**9 points**

Intent: *Define extension to* `Rock` *class that alternates between red and green and, if color is green when acting, causes all neighbors to turn right 90 degrees*

+1 `class Director extends Rock`

+2 Implement constructor

- **+1** `Director(){...}`
  (empty body OK, point lost if extraneous code causes side effect)
- **+1** Sets initial color to `Color.RED` with `setColor` or `super(Color.RED)`

+6 Override `act`

- **+1** Alternates color correctly (*point lost for incorrect* `act` *header*)
- **+5** Turn neighbors
  - **+1** Instructs other object to turn if and only if this Director's color is green when it begins to act
  - **+1** Uses `getGrid` in identifying neighbors
  - **+1** Identifies all and only neighbors or neighboring locations
  - **+1** Accesses all identified actors or locations *(no bounds errors)*
  - **+1** Calls `setDirection` with appropriate parameter on all identified actors

AP® COMPUTER SCIENCE A
2014 CANONICAL SOLUTIONS

Question 2: Director

```
public class Director extends Rock
{
   public Director()
   {
      super(Color.RED);
   }

   public void act()
   {
      if (getColor().equals(Color.GREEN))
      {
         ArrayList<Actor> neighbors = getGrid().getNeighbors(getLocation());
         for (Actor actor : neighbors)
         {
            actor.setDirection(actor.getDirection() + Location.RIGHT);
         }
         setColor(Color.RED);
      }
      else
      {
         setColor(Color.GREEN);
      }
   }

}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.