# AP® COMPUTER SCIENCE A
# 2014 GENERAL SCORING GUIDELINES

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question.

**1-Point Penalty**

(w) Extraneous code that causes side effect (*e.g., writing to output, failure to compile*)

(x) Local variables used but none declared

(y) Destruction of persistent data *(e.g., changing value referenced by parameter)*

(z) Void method or constructor that returns a value

**No Penalty**

- Extraneous code with no side effect (*e.g., precondition check, no-op*)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- Array/collection access confusion (`[] get`)
- `length/size` confusion for array, `String`, `List`, or `ArrayList`, with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, *e.g.,* `int[size] nums = new int[size];`
- Missing `;` provided majority are present and indentation clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent and `{ }` are used elsewhere
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

**Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously** inferred from context; for example, "`ArayList`" instead of "`ArrayList`". As a counterexample, note that if the code 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.*

# 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.

2A

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.

```
public class Director extends Rock
{

   public Director()
   {
      super(Color.RED);
   }

   public void act()
   {
      if (getColor().equals(Color.GREEN)
      {
         setColor(Color.RED);
         ArrayList<Actor> actors = getGrid().getNeighbors(getL
                                                  cation());

         if (locs.size() == 0)
            return;
         else
         { for (Actor a : actors)
            { a.setDirection(a.getDirection()+ Location.RIGHT
            }
         }
      }
      else if (getColor().equals(Color.RED)
         setColor(Color.GREEN);
      else
      {}
```



GO ON TO THE NEXT PAGE.

2B

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.

```
import info.gridworld.grid.Location;
import java.util.ArrayList;
public class Director extends info.gridworld.actor.Rock {
    public Director() {
        this.setColor(Color.RED)}
    public void act() {
        if (getGrid() == null)
            return;
        ArrayList<Actor> actors = getActors();
        for (int i=0; i < actors.size(); i++){
            actors.get(i).setDirection(actors.get(i).getDirection()+90);
            }
    public ArrayList<Actor> getActors() {
        return getGrid.getNeighbors(getLocation());
        }
```

2C

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.

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

 public void act()
 {
   if(getColor() == Color.GREEN)
   {
     ArrayList<Actor> actors = new ArrayList<Actor>();
     actors = getActors
```

# AP® COMPUTER SCIENCE A
# 2014 SCORING COMMENTARY

## Question 2

### Overview

This question involved reasoning in the context of the GridWorld case study. The question required the design and declaration of a class including proper keywords, inheritance, constructor, method overriding, and accessing and modifying inherited and nonmember instance variables and using constants. This problem tested students' knowledge of GridWorld classes/interfaces: `Rock`, `Actor`, `Location`, `Grid`, and Java classes/interfaces from the AP subset: `Object`, and `List<E>`. The question necessitated a class header, a no-argument constructor and overriding the `act` method. Students were required to create the `Director` class as a subclass of `Rock`. The question required a no-argument constructor, which set the color state of a director to be `Color.RED`. Students had to override the `act` method of the superclass so the same method signature and return type as the superclass (`Rock`) were required. The director's behavior required a check of the color state of the director in order to determine how to change colors. When the director starts to `act` in the green color state, the director will turn all of its neighbors to the right 90 degrees. The director turns (changes the direction of) all of its neighbors right by accessing the neighbor's current direction state in order to update and set the new direction state of the neighbor. If the director's color state is green at the beginning of `act`, the director changes color to `Color.RED` and vice versa.

**Sample: 2A**
**Score: 9**

The response begins with a proper class heading for a `Director` class that inherits from `Rock`. The heading is correctly followed by the class body, surrounded by braces. The no-argument constructor invokes the super class's constructor with the correct argument of `Color.RED`.

There is an `act` method, which overrides the superclass's `act` method. There is a check for the director's current color state of green to guard both the director's color change and turning neighbors. The director will turn red when the director is green at the start of the `act` method. When the director is not green at the start of `act`, the director will turn red.

This solution completely guards the neighbor's turning with a check for green (many students did not account for the director's color state being colors other than red or green). The neighboring actors are collected into a list by getting the current grid and using it to access the neighbors based on the director's current location. The solution correctly iterates over the list and sets each neighbor's direction by adding `Location.RIGHT` to its current direction.

**Sample: 2B**
**Score: 7**

The response begins with a proper class heading for a `Director` class that inherits from `Rock`. The heading is correctly followed by the class body, surrounded by braces. The `Director` no-argument constructor is properly declared and does not need to explicitly invoke the super class's constructor. The solution correctly uses the `setColor` modifier method to alter the color state of the constructed director. The use of the constant `Color.RED` from the `Color` class was required for full credit when modifying the director's color in the constructor.

There is an `act` method, which overrides the inherited `act` method. The director will not alternate colors and was not awarded the "alternates color" point. The director will always turn the neighbors when

## Question 2 (continued)

it starts `act`, so the solution is not awarded the "only turn when green" point. The `getGrid` accessor method is used to get the `Grid` object of the director. The solution checks to see if the grid is valid by comparing it to `null` (this code was not required but is correct in the context of the case study). The soltuion creates an `ArrayList<Actor>` reference and sets it to the list returned by the `getActors` helper method, which the solution includes as part of the `Director` class. The `getActors` helper method returns the list generated by the director's grid object using the `getNeighbors` method based on the director's location. The solution uses a correct for-loop to access all of the elements in the list of neighbors. Each neighboring actor's direction is correctly modified by setting it to the current direction plus 90.

**Sample: 2C**
**Score: 3**

The response begins with a proper class heading for a `Director` class that inherits from `Rock`. The heading is correctly followed by the class body, surrounded by braces. The `Director` no-argument constructor is properly declared and does not need to explicitly invoke the super class's constructor. The `setColor` modifier method alters the color state of the director. The correct color argument `Color.RED` guarantees an initial color state of red.

There is an `act` method, which overrides the `Rock` class's `act` method. There is a check for the director's current color state of green, but the guard does not protect any useful code because the `getActors` method is not defined for the `Director` class. The solution does not earn any of the 5 points associated with the implementation of the `act` method.