AP® COMPUTER SCIENCE A
2012 GENERAL SCORING GUIDELINES

Apply the question-specific 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-specific rubric. No part of a question — (a), (b), or (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 different parts of that question.

1-Point Penalty

(w) Extraneous code that causes a side effect or prevents earning points in the rubric (*e.g., information written to output*)

(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 that causes no side effect
- Extraneous code that is unreachable and would not have earned points in rubric
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared, provided that other variables are declared in some part
- `private` qualifier on local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (x • ÷ ≤ ≥ < > ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` (and vice versa)
- Array/collection element access confusion (`[]` vs. `get` for r-values)
- Array/collection element modification confusion (`[]` vs. `set` for l-values)
- `length`/`size` confusion for array, `String`, and `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 that line breaks and indentation clearly convey intent
- Missing `{ }` where indentation clearly conveys intent and `{ }` are used elsewhere
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if/while` conditions
- Use of local variable outside declared scope (must be within same method body)
- Failure to cast object retrieved from nongeneric collection

* *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;`*" and 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
2012 SCORING GUIDELINES

Question 2: RetroBug (GridWorld)

Class: `RetroBug`	**9 points**

Intent: *Define extension to* `Bug` *class that implements a restore method to revert to previous location and direction*

+1 Provides properly formed class header for `RetroBug` that extends `Bug` class

+1 Overrides at least one `Bug` method, other than constructor, and maintains all `Bug` behaviors

+2 Saves state at beginning of `act`
- **+1** Remembers location or direction in `RetroBug` instance variable at beginning of `act` method and nowhere else
  (*point awarded only if instance variable is explicitly declared*)
- **+1** Remembers both location and direction in `RetroBug` instance variables

+5 Implements `restore`
- **+½** Provides correct method header: `public void restore()`
- **+½** Guards against any effect if called before first invocation of `act`
- **+1** Always restores remembered direction
- **+1** Moves to remembered location
- **+1** Moves if remembered location is empty (*must check for empty location*)
- **+1** Moves if remembered location is occupied only by a flower
  (*must check for flower at location*)

Question-Specific Penalties

-1 (r) Use of `"RetroBug."` instead of `"this."`

-1 (v) Confused use of location and direction
(*e.g., saved location used as direction and vice versa*)

-1 (z) Attempts to return a value from `restore`

-0 Missing `public` qualifier on class header

Question 2: RetroBug (GridWorld)

```
public class RetroBug extends Bug {
   Location savedLocation;
   int savedDirection;


   public void act() {
      savedLocation = getLocation();
      savedDirection = getDirection();
      super.act();
   }


   public void restore() {
      if (savedLocation == null) return;
      setDirection(savedDirection);
      if ( getGrid().get(savedLocation) == null
          || getGrid().get(savedLocation) instanceof Flower ) {
         moveTo(savedLocation);
      }
   }

}
```

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

Write the entire `RetroBug` class, including all necessary instance variables and methods.

```
public class RetroBug extends Bug
{
   private Location cur;
   private int direction;

   public void act()
   {
     cur = getLocation();
     direction = getDirection();
     super.act();
   }
   public void restore()
   {
      setDirection(direction);
      if(getGrid.get(cur) == null || getGrid.get(cur) instanceof Flower)
      {
       moveTo(cur);
      }
   }

}
```

GO ON TO THE NEXT PAGE.

Write the entire RetroBug class, including all necessary instance variables and methods.

```
public class RetroBug extends Bug
{
   private Location previousLocation;
   private int previousDirection;
   public void act()
   {
       previousLocation = super.getLocation();
       previousDirection = super.getDirection();
       super.act();
   }
    public void restore()
    {
        if( previousLocation.get()==null || previousLocation.get() - :flower)
        {
             super.moveTo (previousLocation);
             super.setDirection (previousDirection);
        }

    }

 }
```

GO ON TO THE NEXT PAGE.

Write the entire `RetroBug` class, including all necessary instance variables and methods.

```
public class RetroBug extends Bug
{
  public void restore()
  {
    Location oldLoc = getLocation();
    int oldDir = getDirection();
    moveTo(oldLoc);
    setDirection(oldDir);
  }

  public void act()
  {
    super.act();
  }

}


    if (oldLoc instanceOf(flower) ||.oldLoc == null)
```

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE A
2012 SCORING COMMENTARY

Question 2

Overview

This question involved reasoning in the context of the GridWorld case study and the design of a class using inheritance, method overriding, and instance variables to maintain state. This problem tested students' knowledge of the `Bug` class, and both creating and overriding appropriate methods. Students were required to create the `RetroBug` class, as a subclass of `Bug`, whose behavior included remembering previous direction and location, as well as the means to restore previously remembered values. Students had to override an appropriate method of the `Bug` class and write a new method. To be successful in this problem, students needed to understand a bug's behavior, the intended behavior of a retro bug, declare instance variables to remember previous state, override the `act` method, and write a `restore` method.

Sample: 2A
Score: 8.5 (rounded to 9)

The `RetroBug` class is properly declared and correctly extends the `Bug` class. There are instance variables declared to store both the location and direction at the beginning of the `act` method call. The `restore` method is declared correctly and always restores the direction to the remembered direction stored in the instance variables. The `restore` method correctly checks the grid to see if the bug can move to the remembered location. The missing parentheses in the call to `getGrid` are not penalized.

This student's solution is almost perfect but makes a fairly common error. It does not protect the `restore` method from changing the object if the `act` method has not been called and so lost ½ point.

Sample: 2B
Score: 5.5 (rounded to 6)

This student's solution correctly remembers the state of the object but has difficulty restoring the object to the remembered state.

The `RetroBug` class is properly declared and correctly extends the `Bug` class. There are instance variables declared to store both the location and direction at the beginning of the `act` method call. The `restore` method is declared correctly. However, there is no attempt to ensure that `restore` does nothing if `act` has not been called, so ½ point was lost. The `restore` method only restores the direction if the object also moves, so the student lost the point for always restoring the direction.

The student attempts to check if the remembered location is empty or contains a flower, but there are errors in both of these checks, resulting in the loss of another 2 points. When checking if the location is empty, the method checks the location, not the grid; in checking if there is a flower in the location, the equality operator `(==)` is used instead of the `instanceof` operator.

Question 2 (continued)

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

This student's solution fails to remember the state of the object and so is unable to properly restore it.

The `RetroBug` class is properly declared and extends `Bug` correctly. There are no instance variables, so the solution lost both points for correctly storing the direction and location. The solution did receive credit for overriding the `act` method, even though it does not change the `act` method when it overrides it.

The `restore` method is declared correctly. However, there is no attempt to guard the `restore` method from changing the object if the `act` method has not been called, so ½ point was lost. When the `restore` method tries to restore the location and direction, it uses variables declared locally in the `restore` method, so there is no use of remembered values, and 2 points were lost for failing to restore the direction and location. Checking for an empty location or a flower also relies on using a remembered location, so these 2 points were also lost.