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 (x • ÷ ≤ ≥ <> ≠)
○ [ ] 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.
### Question 1: Word Scramble

**Part (a) scrambleWord**

**5 points**

**Intent:** Scramble a word by swapping all letter pairs that begin with A

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>Accesses all letters in word, left to right <em>(no bounds errors)</em></td>
</tr>
<tr>
<td>+1</td>
<td>Identifies at least one letter pair consisting of “A” followed by non-“A”</td>
</tr>
<tr>
<td>+1</td>
<td>Reverses identified pair in constructing result string</td>
</tr>
<tr>
<td>+1</td>
<td>Constructs correct result string <em>(Point lost if any letters swapped more than once, minor loop bounds errors ok)</em></td>
</tr>
<tr>
<td>+1</td>
<td>Returns constructed string</td>
</tr>
</tbody>
</table>

**Part (b) scrambleOrRemove**

**4 points**

**Intent:** Modify list by replacing each word with scrambled version and removing any word unchanged by scrambling

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>Accesses all words in wordList <em>(no bounds errors)</em></td>
</tr>
<tr>
<td>+1</td>
<td>Calls scrambleWord with a word from the list as parameter</td>
</tr>
<tr>
<td>+1</td>
<td>Identifies words unchanged by scrambling</td>
</tr>
<tr>
<td>+1</td>
<td>On exit: List includes all and only words that have been changed by scrambling once, in their original relative order <em>(minor loop bounds errors ok)</em></td>
</tr>
</tbody>
</table>
Part (a):

```java
public static String scrambleWord(String word){
    int current = 0;
    String result = "";
    while (current < word.length() - 1){
        if (word.substring(current, current+1).equals("A") &&
            !word.substring(current+1, current+2).equals("A")){
            result += word.substring(current+1, current+2);
            result += "A";
            current += 2;
        } else {
            result += word.substring(current, current+1);
            current++;}
    }
    if (current < word.length()){result += word.substring(current);}
    return result;
}
```

Part (b):

```java
public static void scrambleOrRemove(List<String> wordList){
    int index = 0;
    while (index < wordList.size()){  
        String word=wordList.get(index);
        String scrambled=scrambleWord(word);
        if (word.equals(scrambled)){
            wordList.remove(index);
        } else {
            wordList.set(index, scrambled);
            index++;
        }
    }
}
```

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.
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
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);
        }
    }
}
Question 3: Seating Chart

Part (a)  SeatingChart constructor  5 points

Intent: Create SeatingChart object from list of students

+1 seats = new Student[rows][cols]; (or equivalent code)
+1 Accesses all elements of studentList (no bounds errors on studentList)
+1 Accesses all necessary elements of seats array (no bounds errors on seats array, point lost if access not column-major order)
+1 Assigns value from studentList to at least one element in seats array
+1 On exit: All elements of seats have correct values (minor loop bounds errors ok)

Part (b)  removeAbsentStudents  4 points

Intent: Remove students with more than given number of absences from seating chart and return count of students removed

+1 Accesses all elements of seats (no bounds errors)
+1 Calls getAbsenceCount() on Student object (point lost if null case not handled correctly)
+1 Assigns null to all elements in seats array when absence count for occupying student > allowedAbsences (point lost if seats array element changed in other cases)
+1 Computes and returns correct number of students removed

Question-Specific Penalties

-2 (v) Consistently uses incorrect array name instead of seats or studentList
Part (a):

public SeatingChart(List<Student> studentList, int rows, int cols)
{
    seats = new Student[rows][cols];
    int studentIndex = 0;
    for (int col = 0; col < cols; col++) {
        for (int row = 0; row < rows; row++) {
            if (studentIndex < studentList.size()) {
                seats[row][col] = studentList.get(studentIndex);
                studentIndex++;
            }
        }
    }
}

Part (a) alternate:

public SeatingChart(List<Student> studentList, int rows, int cols)
{
    seats = new Student[rows][cols];
    int row = 0;
    int col = 0;
    for (Student student : studentList) {
        seats[row][col] = student;
        row++;
        if (row == rows) {
            row = 0;
            col++;
        }
    }
}

Part (b):

public int removeAbsentStudents(int allowedAbsences)
{
    int count = 0;
    for (int row = 0; row < seats.length; row++) {
        for (int col = 0; col < seats[0].length; col++) {
            if (seats[row][col] != null &&
                seats[row][col].getAbsenceCount() > allowedAbsences) {
                seats[row][col] = null;
                count++;
            }
        }
    }
    return count;
}
Question 4: Trio

Class: Trio

<table>
<thead>
<tr>
<th>Points</th>
<th>Code Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>public class Trio implements MenuItem</td>
</tr>
<tr>
<td>+1</td>
<td>Declares appropriate private instance variables</td>
</tr>
<tr>
<td>+2</td>
<td>Implements constructor</td>
</tr>
<tr>
<td>+1</td>
<td>public Trio(Sandwich sandwich, Salad salad, Drink drink)</td>
</tr>
<tr>
<td>+1</td>
<td>Initializes appropriate instance variables using parameters</td>
</tr>
<tr>
<td>+1</td>
<td>Implements interface methods</td>
</tr>
<tr>
<td></td>
<td>(public String getName() {...}, public double getPrice() {...})</td>
</tr>
<tr>
<td>+1</td>
<td>Constructs correct name string and makes available for return in getName</td>
</tr>
<tr>
<td>+1</td>
<td>Returns constructed name string in getName</td>
</tr>
<tr>
<td>+1</td>
<td>Computes correct price and makes available for return in getPrice</td>
</tr>
<tr>
<td>+1</td>
<td>Returns computed price in getPrice</td>
</tr>
</tbody>
</table>

Question-Specific Penalties

-0  Missing or extra spaces in name string, “trio”

-1  (w) Extraneous default constructor that causes side effect
public class Trio implements MenuItem {
    private Sandwich sandwich;
    private Salad salad;
    private Drink drink;

    public Trio(Sandwich s, Salad sal, Drink d) {
        sandwich = s;
        salad = sal;
        drink = d;
    }

    public String getName() {
        return sandwich.getName() + "/" + salad.getName() + "/
        drink.getName() + " Trio";
    }

    public double getPrice() {
        double sandwichPrice = sandwich.getPrice();
        double saladPrice = salad.getPrice();
        double drinkPrice = drink.getPrice();
        if (sandwichPrice <= saladPrice && sandwichPrice <= drinkPrice)
            return sandwichPrice + drinkPrice;
        else if (saladPrice <= sandwichPrice && saladPrice <= drinkPrice)
            return sandwichPrice + drinkPrice;
        else
            return sandwichPrice + saladPrice;
    }
}

Alternate

public class Trio implements MenuItem {
    private String name;
    private double price;

    public Trio(Sandwich s, Salad sal, Drink d) {
        double sandwichPrice = s.getPrice();
        double saladPrice = sal.getPrice();
        double drinkPrice = d.getPrice();
        if (sandwichPrice <= saladPrice && sandwichPrice <= drinkPrice)
            price = sandwichPrice + drinkPrice;
        else if (saladPrice <= sandwichPrice && saladPrice <= drinkPrice)
            price = sandwichPrice + drinkPrice;
        else
            price = sandwichPrice + saladPrice;
        name = s.getName() + "/" + sal.getName() + "/" + d.getName() + " Trio";
    }
}

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.
public String getName(){
    return name;
}

public double getPrice(){
    return price;
}
}