AP® Computer Science A
2014 Scoring Guidelines

CollegeBoard

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

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

> **+1**   Accesses all letters in `word,` left to right (*no bounds errors*)

> **+1**   Identifies at least one letter pair consisting of "A" followed by non-"A"

> **+1**   Reverses identified pair in constructing result string

> **+1**   Constructs correct result string (*Point lost if any letters swapped more than once, minor loop bounds errors ok*)

> **+1**   Returns constructed string

Part (b)	scrambleOrRemove	**4 points**

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

> **+1**   Accesses all words in `wordList` (*no bounds errors*)

> **+1**   Calls `scrambleWord` with a word from the list as parameter

> **+1**   Identifies words unchanged by scrambling

> **+1**   On exit: List includes all and only words that have been changed by scrambling once, in their original relative order (*minor loop bounds errors ok*)

Question 1: Word Scramble

Part (a):

```
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):

```
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++;
        }
    }
}
```

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

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);
        }
    }

}
```

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`

Question 3: SeatingChart

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;
}
```

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 4: Trio

Class:	Trio	9 points

Intent: *Define implementation of* `MenuItem` *interface that consists of sandwich, salad, and drink*

+1   `public class Trio implements MenuItem`

+1   Declares appropriate `private` instance variables

+2   Implements constructor

     +1   `public Trio(Sandwich sandwich, Salad salad, Drink drink)`

     +1   Initializes appropriate instance variables using parameters

+1   Implements interface methods
(`public String getName(){…}, public double getPrice(){…}`)

+1   Constructs correct name string and makes available for return in `getName`

+1   Returns constructed name string in `getName`

+1   Computes correct price and makes available for return in `getPrice`

+1   Returns computed price in `getPrice`

Question-Specific Penalties

-0   Missing or extra spaces in name string, "trio"

-1   (w) Extraneous default constructor that causes side effect

Question 4: Trio

```java
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 saladPrice + drinkPrice;
        else if (saladPrice <= sandwichPrice && saladPrice <= drinkPrice)
            return sandwichPrice + drinkPrice;
        else
            return sandwichPrice + saladPrice;
    }
}
```

Alternate

```java
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 = saladPrice + 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.

Question 4: Trio continued

```java
    public String getName(){
        return name;
    }

    public double getPrice(){
        return price;
    }
}
```