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

- 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 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):

```java
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:

```java
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):

```java
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.

Complete the `SeatingChart` constructor below.

```
/** 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){
    seats = new Student [rows][cols];
    int sl=0;
    for (int j = 0 ; j < cols ; j++){
        for (int i = 0; i < rows ; i++){
            if (sl <= studentlist.size(){
                seats [i][j]= studentlist.get(sl);
                sl++ ;
            }
            else {
                seats [i][j]= null ;
            }
        }
    }
}
```

Complete method `removeAbsentStudents` below.

```
/** 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){
    int count=0;
    for (int i=0; i < seats.length; i++){
        for (int j=0; j < seats[0].length; j++){
            if (seats[i][j] != null){
                if (seats[i][j].getAbsenceCount()>allowedAbsences){
                    seats[i][j] = null;
                    count ++;
                }
            }
        }
    }
    return count;
}

// if this is the first time removeAbsentStudents is invoked
// one could iterate through the array as we did in question
// a and break through the loop the instance we found a null;
// would be more run time efficient but wouldn't work for
// multiple invoking
```

GO ON TO THE NEXT PAGE.

Complete the `SeatingChart` constructor below.

```
/** 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) {
    int pos = 0;
    seats = new int[rows][cols];

    for(int r = 0; r < rows; r++){
        for(int c = 0; c < cols; c++){
            seats[c][r] = studentList.get(pos);
            pos++;
            if(pos > studentList.size())
                break;
        }
        if(pos > studentList.size())
            break;
    }
}
```

Complete method `removeAbsentStudents` below.

```
/** 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) {
    int total = 0;
    for(int r = 0; r < seats.length; r++)
        for(int c = 0; c < seats[0].length; c++)
            if( seats[r][c].getAbsenceCount() > allowedAbsences){
                seats[r][c] = null;
                total++;
            }

    return total;
}
```

-15-

Complete the SeatingChart constructor below.

```
/**  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)
```

```
int index = 0;
For( int x=0;  x < rows ; x++ ){
    for( int y=0;  y < cols ;  y++){          NOT
        IF (!(studentList.get(index) == null){  //if out of bands
            seats[x][y] = studentList.get(index);
        else{      ,
            seats[x][y] = null;
        }
    }
}
```

```
int total = studentList.size()
int srow = total/4
int scol = total/3
```

GO ON TO THE NEXT PAGE.

3Cb

Complete method `removeAbsentStudents` below.

```
/** 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)
```

```
int ct = 0;
for (int x = 0; x < rows; x++){
for (int y = 0; y < cols; y++) {

    if (seats[x][y].getAbsenceCount() > allowedAbsences){
          seats[x][y] = null;
    }     ct++;

}
}

return ct;
```

Question 3

Overview

This question involved the construction, initialization, and manipulation of a two-dimensional array. It also tests the student's ability to traverse a `List`, manage a counter, and return a value from a method. Students were asked to implement a constructor and a method of the `SeatingChart` class.

In part (a) students were asked to implement a constructor, which required the instantiation of the instance variable `seats`, a 2D array of `Student` objects, whose dimensions were determined by the parameters `rows` and `cols`. The elements of `studentList` were to be mapped to the 2D array `seats` in column-major order until all list elements had been assigned to the 2D array. Any remaining elements of the 2D array held their default `null` values.

In part (b) students were required to examine the instance 2D array `seats`, removing all `Student` elements whose absence count exceeded the parameter `allowedAbsences` by replacing the `Student` object with `null`. The method calculated and returned the number of `Student` objects that were removed.

Sample: 3A
Score: 8

In part (a) the student correctly creates the `seats` array as a two-dimensional array of `Student` objects. The student uses nested `for` loops to access all of the elements of `seats` in column-major order and attempts to fill the `seats` array with elements from the `studentList`, earning the "accesses all necessary elements of seats array" and "assigns value from `studentList` to at least one element in seats array" points. The student did not earn the second point because the condition `s1 <= studentList.size()` incorrectly checks the index of the `studentList`. This causes an exception to occur. At the time the exception occurs, all elements of `studentList` have been correctly mapped to the `seats` array in column-major order, so the student earned the "all elements of `seats` have correct values" point. The student earned 4 points in part (a).

In part (b), the student accesses all of the elements of the `seats` array. Each element is checked to ensure it is not `null` before calling its `getAbsenceCount()` method. Whenever a student's absence count exceeds `allowedAbsence`, the `Student` object is removed from the `seats` array by replacing it with `null`. A counter is correctly declared, initialized, updated, and returned to report the number of students removed from the `seats` array. The student earned 4 points in part (b).

Sample: 3B
Score: 4

In part (a) the student creates the `seats` array as a two-dimensional array of integers instead of `Student` objects, so the students does not earn the "`seats = new Student[rows][cols]`" point. The student correctly accesses all elements from the `studentList`, breaking the nested loops when the end of the list is reached. However, checking the index after using it fails whenever `studentList.size() == 0`, so the student does not earn the "accesses all elements of `studentList`" point. The student attempts to use nested `for` loops to access all of the elements of `seats` in column-major order, but reverses the row and column indexes when accessing elements from

the array. As a consequence, the response does not earn the "accesses all necessary elements of seats array" point and an exception occurs. Since at least one element of `studentList` is assigned to the `seats` array, the student earned the "assigns value from `studentList` to at least one element in seats array" point. At the time the exception occurs, not all elements of `studentList` have been correctly mapped to the `seats` array in column-major order, so the student did not earn the "all elements of `seats` have correct values" point. The student earned 1 point in part (a).

In part (b), the student accesses all of the elements of the `seats` array, earning the "`seats = new Student[rows][cols]`" point. The student did not earn the "accesses all elements of `studentList`" point because each element is not checked to ensure it is not `null` before calling its `getAbsenceCount()` method. In all other cases, whenever a student's absence count exceeds `allowedAbsence`, the `Student` object is removed from the `seats` array by assigning `null` to its row and column position. A counter is correctly declared, initialized, updated, and returned to report the number of students removed from the `seats` array. The student earned 3 points in part (b).

Sample: 3C
Score: 3

In part (a) the student does not create the `seats` array, and did not earn the "`seats = new Student[rows][cols]`" point. The student attempts to access all elements from the `studentList`, but has a bad out-of-bounds check and fails to increment the index. As a result, the student did not earn the "accesses all elements of `studentList`" point. The student attempts to use nested `for` loops to fill the `seats` array with elements from the `studentList`. The `seats` array is filled in row-major order instead of column-major order, thus the response did not earn the "accesses all necessary elements of seats array" point. Since at least one element of `studentList` is assigned to the `seats` array, the student earned the fourth point. The student did not earn the "all elements of `seats` have correct values" point because the `seats` array was not filled in column-major order and the value `studentList.get(0)` has been assigned to each element of the `seats` array due to the `studentList` index not being incremented. The student earned 1 point in part (a).

In part (b), the student attempts to access all of the elements of the `seats` array. Since the `rows` and `cols` variables are not defined for this method, the response does not earn the "accesses all elements of `seats`" point. The student did not earn the "Calls `getAbsenceCount()` on `Student` object" point because each element is not checked to ensure it is not `null` before calling its `getAbsenceCount()` method. In all other cases, whenever a student's absence count exceeds `allowedAbsence`, the `Student` object is removed from the `seats` array by assigning `null` to its row and column position. A counter is correctly declared, initialized, updated, and returned to report the number of students removed from the `seats` array. The student earned 2 points in part (b).