2026

AP® Computer Science A

Scoring Guidelines

Applying the Scoring Criteria

Algorithm Points

The scoring guidelines for each question include one or two scoring criteria labeled "(*algorithm*)," each of which is worth one point. These algorithm criteria require that a response applies all necessary steps and connects them in an appropriate order. An algorithm point may be earned even if steps assessed in other scoring criteria are present but incorrect. However, there are some situations in which algorithm points are not earned.

Algorithm points are NOT earned when:

- Required pieces are omitted.
- Steps are assembled incorrectly.
- Additional code not assessed in other scoring criteria makes the solution incorrect (e.g., printing output, incorrect precondition check).
- Array/collection access is reversed (e.g., `[]` instead of `.get`) when not otherwise assessed.
- Persistent data is modified (e.g., changing value referenced by parameter).
- A value is returned from a `void` method or constructor.
- A provided method header is rewritten with a different number or type of parameters.

Non-Algorithm Points

All scoring criteria not labeled "(*algorithm*)," which are also worth one point each, are assessed with a narrower focus. Points for these scoring criteria can be earned even when other errors are present, as long as the specific requirement for that point is met.

Errors to Ignore

For both algorithm and non-algorithm points, some errors are considered minor and do not affect earning the point unless a scoring guideline explicitly says otherwise. The point may still be earned even when the errors are present. Examples include:

- Extraneous code with no side effect (e.g., valid precondition check)
- Spelling/case discrepancies where there is no ambiguity
- Local variable not declared (unless scoring guidelines specifically require it)
- Local variable declared in narrower scope than necessary (inner `{ }`)
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on a class or constructor header
- Provided method header rewritten with different parameter names
- Keyword used as an identifier
- `length/size` confusion for array, `String`, or `ArrayList`, with or without `( )`
- Extraneous size in array declaration (e.g., `int[size] nums = new int[size];`)
- `:` instead of `;` and vice versa
- `,` instead of `;` and vice versa
- Missing `( )` on parameter-less method or constructor invocations
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where structure clearly conveys intent
- Missing `( )` around control headers (`if`, `while`, `for`) where structure clearly conveys intent

General Interpretation Rules

Interpreting Incorrect Spelling/Typing

The point may still be earned for minor discrepancies when they do not introduce ambiguity. The following rules clarify how to interpret such discrepancies:

- The point CAN be earned for spelling and case discrepancies for identifiers only if the correction can be **unambiguously** inferred from context (e.g., `"ArayList"` instead of `"ArrayList"`). Missing words from identifiers (e.g., `"dogShift"` for `"dogWalkShift"`) are only unambiguous if there is absolutely no similar identifier it can be mistaken for.
- The point CANNOT be earned for case discrepancies that introduce ambiguity. For example, if the code declares `"int G=99, g=0;"`, then uses `"while (G < 10)"` instead of `"while (g < 10)"`, the context does **not** allow for the reader to assume the use of the lowercase variable.
- The point CANNOT be earned for spelling discrepancies that introduce ambiguity. For example, calling a `"signature"` method when both `"getSignature"` and `"addSignature"` are seen in the question does **not** allow for the reader to assume the use of the correct method.
- The point CAN be earned when a response **clearly** indicates and explains a necessary keyboard character substitution (e.g., for a broken key on a keyboard). The reader can incorporate that communication when evaluating the response. The point CANNOT be earned for arbitrary character substitutions without explanation.

Interpreting Incorrect Statements When Structure Does Not Convey Intent

As indicated, the point can be earned when omitting semicolons, one or both parentheses around control statement headers, and one or both curly brackets in methods and blocks, *where structure clearly conveys intent*, typically through use of newlines and indentation. Indentation is **not** required for a correct solution, but it is one of the widest and best-understood ways to convey intended structure of a program, and it can be taken into account when scoring if it is clear. Otherwise, the following rules should be used to clarify how to interpret the response **when indentation is inconsistent or absent**:

- If the bracketing `( ) { }` is complete and balanced, score the response as written.
- Assume an open/left curly bracket `{` immediately after any method header or class header that does not already have one.
- Assume an appropriate number of closing/right brackets `}` before any method header to close all open brackets from the previous method or constructor.
- If there is a missing closing/right parenthesis `)`, assume it is at the end of the line where it opened, immediately before the semicolon or curly bracket (if any).
- If there is a missing closing double-quote `"`, for an argument that uses a string literal, assume the closing double quote is at the end of the argument; otherwise assume the closing double quote is at the end of the line where it opened, immediately before the semicolon or curly bracket (if any).
- On a control statement (`if`, `else`, `while`, `for`) without bracketing or indentation, only the next line is controlled by the statement.
- On a control statement (`if`, `else`, `while`, `for`) with open/left curly bracket `{` and no indentation cues, the entire remainder of the method is “inside” the statement.

Question 1: Methods and Control Structures — 7 points

Model Solution

A — **4 points**

```
public Account(String requestedName)
{
   if (isAvailable(requestedName))
   {
      username = requestedName;
   }
   else
   {
      int count = 1;

      while (!isAvailable(requestedName + count))
      {
         count++;
      }
      username = requestedName + count;
   }
}
```

B — **3 points**

```
public String getShortenedName()
{
   String result = username;

   while (result.indexOf("-") >= 0)
   {
      int j = result.indexOf("-");
      result = result.substring(0, j - 1) +
                  result.substring(j + 1);
   }

   return result;
}
```

Part A `Account`

	Scoring Criteria	Decision Rules	
A1	Calls `isAvailable` with `String` argument	Responses **can** still earn the point even if they • fail to save or use the returned value • call `isAvailable` with a `String` other than the parameter Responses **will not** earn the point if they • make any incorrect call to `isAvailable` • call `isAvailable` on anything other than the class `Account` or `this` (use of `this` is optional)	**1 point**
A2	Updates potential usernames until an available username is found	Responses **can** still earn the point even if they • call `isAvailable` incorrectly • fail to use `requestedName` • update potential usernames incorrectly Responses **will not** earn the point if they • fail to call the check availability method • check availability without a potential username • fail to update a potential username in the body of the loop	**1 point**
A3	Increments a count (*in the context of a loop*)	Responses **can** still earn the point even if they • fail to use the count as part of a possible username • fail to initialize the count	**1 point**
A4	Assigns a correct value to `username` based on `requestedName` and identified availability (*algorithm*)	Responses **can** still earn the point even if they • fail to pass the updated username to `isAvailable` • use an incorrect availability check • increment the count incorrectly Responses **will not** earn the point if they • fail to initialize or incorrectly initialize the count • fail to check availability • fail to increment a count • fail to append correct integer to `username`, when necessary • fail to handle the case when `requestedName` itself is available • fail to initialize the instance variable `username` in any case • print or return a value in addition to or instead of initializing `username`	**1 point**

Part B `getShortenedName`

	Scoring Criteria	Decision Rules	
B1	Considers multiple parts or indices of `username` in the context of a loop (*body of control flow statement must not be empty*)	Responses **can** still earn the point even if they • call any `String` method incorrectly • use a loop variable with incorrect bound (e.g., constant), as long as it is used as an index of `username` • fail to use the loop variable as an index of `username`, as long as the length of `username` is used to bound the loop • return early, as long as bounds and indices would otherwise support accessing at multiple indices • use array access notation `[]` on the `String` to obtain single characters Responses **will not** earn the point if they • call `split` but fail to access at multiple indices • use a loop condition that never evaluates to `true` • use a loop condition that always evaluates to `true` and no other conditions stop the loop appropriately	**1 point**
B2	Identifies a location of `"-"` within `username` instance variable (and all `String` method calls are syntactically valid)	Responses **can** still earn the point even if they • access characters out-of-bounds • call the `split` method correctly, but fail to access the result • use underscore instead of hyphen Responses **will not** earn the point if they • call any `String` method incorrectly • fail to distinguish separate words based on `"_"` • fail to store or use the return value of the `String` method that identifies the location of the `"-"` • use array access notation `[]` on the `String` to obtain single characters	**1 point**

B3	Returns correct shortened string (*algorithm*)	Responses **can** still earn the point even if they • call any `String` method incorrectly • use array access notation `[]` on the `String` to obtain single characters Responses **will not** earn the point if they • fail to return the string • incorrectly build the string (e.g., early return, fail to initialize correctly) • have bounds errors • modify `username` • print the string instead of or in addition to returning it	**1 point**

Alternate Model Solution

A

```
public Account(String requestedName)
{
   String temp = requestedName;
   int count = 1;

   while (!isAvailable(temp))
   {
      temp = requestedName + count;
      count++;
   }

   username = temp;
}
```

B

```
public String getShortenedName()
{
   String[] temp = username.split("-");
   String result = "";

   for (int j = 0; j < temp.length - 1; j++)
   {
      result += temp[j].substring(0, temp[j].length() - 1);
   }

   result += temp[temp.length - 1];

   return result;
}
```

or

```
public String getShortenedName()
{
   String result = "";

   for (int j = 1; j < username.length(); j++)
   {
      String curr = username.substring(j, j + 1);

      if (curr.equals("-"))
      {
         j++;
      }
      else
      {
         result += username.substring(j – 1, j);
      }
   }
   result += username.substring(username.length() - 1);

   return result;
}
```

Question 2: Class Design — 7 points

Model Solution

7 points

```
public class Bottle
{
   private double capacity;
   private double currentAmount;

   public Bottle(double start)
   {
      capacity = start;
      currentAmount = start;
   }

   public double updateAmount(double amountUsed)
   {
      currentAmount -= amountUsed;

      if (currentAmount < capacity * 0.25)
      {
          currentAmount = capacity;
      }

      return currentAmount;
   }
}
```

`Bottle`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class Bottle`	Responses **will not** earn the point if they • declare the class as `private` or anything other than `public` • include extraneous code outside the class • include `()` with or without parameters in the class header	**1 point**
2	Declares all necessary `private` instance variables of appropriate types to store the remaining amount and capacity of liquid in a bottle	Responses **can** still earn the point even if they • incorrectly initialize the instance variables • declare capacity amount instance variable as an `int` instead of a `double` Responses **will not** earn the point if they • declare any instance variable `static` • omit `private` in any instance variable declaration • declare an instance variable inside the constructor • declare a variable outside the class	**1 point**
3	Declares constructor header: `Bottle(double ____)`	Responses **will not** earn the point if they • declare the constructor as `private` or `static` • include a return type and/or return a value	**1 point**
4	Constructor assigns correct initial value to capacity instance variable	Responses **can** still earn the point even if they • write a class header with parameters and assign the parameters to the instance variable • fail to declare or use the default initialization for the remaining amount instance variable Responses **will not** earn the point if they • fail to declare or initialize an instance variable for capacity • fail to use the parameter • assign instance variables from an unknown source • assign initial value to local variables instead of the instance variable, even if the local variables are declared as `private` • assign a value to an instance variable with an incompatible data type	**1 point**

5	Declares method header: `public double updateAmount(double ___)`	Responses **can** still earn the point even if they • declare additional methods correctly or incorrectly Responses **will not** earn the point if they • omit `public` in method header or declare the method as something other than `public` • use an incorrect method name • use an incorrect return type or parameter type	**1 point**
6	Calculates and updates the correct remaining amount in the bottle, including a refill when appropriate (*algorithm*)	Responses **can** still earn the point even if they • fail to return the correct remaining amount or print it instead • use variables declared outside the method as instance variables when instance variables do not exist Responses **will not** earn the point if they • fail to handle the case when the remaining amount is less than 25% of the capacity • fail to update the remaining amount instance variable • calculate the correct remaining amount but return an incorrect value • calculate an incorrect remaining amount due to an early return	**1 point**
7	Returns calculated remaining amount from within the `updateAmount` method	Responses **can** still earn the point even if they • return an incorrect remaining amount • return early Responses **will not** earn the point if they • fail to return a value in a case that the response handles • return a value without computation (e.g., conditional) • return a value of a nonnumeric type • print the value in addition to or instead of returning it	**1 point**

Question 3: Data Analysis with `ArrayList` — 5 points

Model Solution

5 points

```
public int moreHistoryThanMathAbsences()
{
   int count = 0;

   for (CourseRecord hst : historyList)
   {
      for (CourseRecord mth : mathList)
      {
         if (hst.getStudentID().equals(mth.getStudentID())
             && hst.getAbsences() > mth.getAbsences())
         {
            count++;
         }
      }
   }
   return count;
}
```

`moreHistoryThanMathAbsences`

	Scoring Criteria	Decision Rules	
1	Accesses all elements in `historyList` and `mathList` (*no bounds errors*)	Responses **can** still earn the point even if they • fail to use the loop variable in the body of an enhanced `for` loop, which inherently accesses each element • fail to use the accessed value • return early, as long as bounds and indices would otherwise support accessing all elements Responses **will not** earn the point if they • use an indexed loop with correct bounds but fail to access elements inside the loop • only access one of the lists • access the elements of `historyList` or `mathList` incorrectly • use array syntax to access `ArrayList` elements (`[ ]`)	**1 point**
2	Calls `getStudentID` and `getAbsences` on an element from one of the lists	Responses **can** still earn the point even if they • use array syntax to access `ArrayList` elements (`[ ]`) • have a bounds error Responses **will not** earn the point if they • call either method on anything other than a `CourseRecord` object • ever call `getStudentID` or `getAbsences` incorrectly	**1 point**
3	Determines whether an element from one list has more absences than an element from the other list	Responses **can** still earn the point even if they • access a `CourseRecord` object or `ArrayList` incorrectly • call `getAbsences` incorrectly • compare elements with possibly different IDs Responses **will not** earn the point if they • use incorrect logic for the condition (e.g., `>=` instead of `>`) • compare absences using `==` or `!=` • fail to call the absences method	**1 point**

	Scoring Criteria	Decision Rules	
4	Initializes count and increments it within a conditional statement (*in the context of a loop*)	Responses **can** still earn the point even if they • initialize the count incorrectly • fail to identify all necessary students • return early • incorrectly call or fail to call `getStudentID` or `getAbsences` Responses **will not** earn the point if they • fail to initialize the count or reset it within a loop • increment count without a conditional statement	**1 point**
5	Returns count of all and only necessary students (*algorithm*)	Responses **can** still earn the point even if they • access a `CourseRecord` object or `ArrayList` incorrectly • stop loop early or late due to incorrect bounds • use array syntax to access `ArrayList` elements (`[ ]`) Responses **will not** earn the point if they • fail to return the computed value • return an incorrect value due to an early return • fail to compare all necessary pairs of elements • use `==` to compare `String` values • modify either list • fail to compare both IDs and absences • print the count in addition to or instead of returning it	**1 point**

Question 4: 2D Arrays — 6 points

Model Solution

6 points

```
public int getPointsForRow(int targetRow)
{
   boolean sameColor = true;

   int total = 0;
   String firstColor = board[targetRow][0].getColor();

   for (Space spc : board[targetRow])
   {
      String currentColor = spc.getColor();

      if (!currentColor.equals(firstColor))
      {
         sameColor = false;
      }

      total += spc.getPoints();
   }

   if (sameColor)
   {
      return total * 2;
   }
   return total;
}
```

`getPointsForRow`

	Scoring Criteria	Decision Rules	
1	Accesses all elements of `board[targetRow]` (*no bounds error*)	Responses **can** still earn the point even if they • fail to use the loop variable in the body of an enhanced `for` loop over a row that inherently accesses each element • fail to use the accessed element • return early, as long as bounds and indices would otherwise support accessing all necessary elements Responses **will not** earn the point if they • incorrectly access or fail to access elements of `board`	**1 point**
2	Calls `getColor` or `getPoints` on an element of the 2D array	Responses **can** still earn the point even if they • access an element of the 2D array incorrectly • have a bounds error Responses **will not** earn the point if they • fail to call `getColor` or `getPoints` at least one time • call `getColor` or `getPoints` incorrectly • call `getColor` or `getPoints` on anything other than a `Space` object • fail to access the 2D array	**1 point**
3	Determines whether two different `Space` objects have the same color	Responses **can** still earn the point even if they • call the `getColor` method incorrectly • access an element of the 2D array incorrectly • confuse rows and columns Responses **will not** earn the point if they • use `==` or `!=` to compare colors • fail to call the color method	**1 point**
4	Initializes point variable and accumulates points (*in the context of a loop*)	Responses **can** still earn the point even if they • initialize the point variable incorrectly • compute an incorrect sum, as long as the sum is calculated using the obtained point values • call the point method incorrectly • confuse rows and columns • traverse a row other than target row • access an element of the 2D array incorrectly	**1 point**

		Responses **will not** earn the point if they • fail to initialize the point variable • fail to call the point method	
5	Determines whether all considered spaces are the same color (*algorithm*)	Responses **can** still earn the point even if they • consider spaces in more than one row • fail to access some elements of the 2D array due to a bounds error • access elements beyond bounds • access an element of the 2D array incorrectly • call `getColor` incorrectly • confuse rows and columns Responses **will not** earn the point if they • fail to access spaces in a loop • fail to consider all spaces due to an early return • compare a determined count of same-color pairs with an inconsistent expected count, based on number of pairs considered by the loop	**1 point**
6	Computes and returns correct point value of target row based on determination of whether all spaces in the row are the same color (*algorithm*)	Responses **can** still earn the point even if they • incorrectly determine whether all spaces in the row are the same color • access an element of the 2D array incorrectly Responses **will not** earn the point if they • initialize the point variable incorrectly • consider spaces other than those in the target row • compute the sum incorrectly • compute the sum based on a column instead of a row • fail to return the computed value • fail to return in any case • return an incorrect point value due to an early return • print the result instead of or in addition to returning it	**1 point**

Alternate Model Solution

```
public int getPointsForRow(int targetRow)
{
   int sum = 0;

   for (Space s : board[targetRow])
   {
      sum += s.getPoints();
   }

   for (int i = 1; i < board[targetRow].length; i++)
   {
      if (!board[targetRow][i - 1].getColor()
            .equals(board[targetRow][i].getColor()))
      {
         return sum;
      }
   }
   return sum * 2;
}
```

or

```
public int getPointsForRow(int targetRow)
{
   int sum = 0;
   int matches = 0;
   int numCols = board[targetRow].length;

   for (int i = 0; i < numCols - 1; i++)
   {
      sum += board[targetRow][i].getPoints();
      if (board[targetRow][i].getColor()
            .equals(board[targetRow][i + 1].getColor()))
      {
         matches++;
      }
   }

   sum += board[targetRow][numCols - 1].getPoints();

   if (matches == numCols - 1)
   {
      return sum * 2;
   }
   else
   {
      return sum;
   }
}
```