2026

AP® Computer Science A

Free-Response Questions

COMPUTER SCIENCE A
SECTION II
TIME – 1 HOUR AND 30 MINUTES

Directions:

Section II has 4 free-response questions and lasts 1 hour and 30 minutes.

All program segments must be written in Java.

Java Quick Reference information for programming questions can be used throughout the exam. A digital version is available in this application.

Assume that the classes listed in the Java Quick Reference have been imported where appropriate.

Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.

In writing solutions for each question, you may use any of the accessible methods listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

You may use the available paper for scratch work, but credit will only be given for responses entered in this application. Text you enter as an annotation will **not** be included as part of your answer.

You can go back and forth between questions in this section until time expires. The clock will turn red when 5 minutes remain—**the proctor will not give you any time updates or warnings.**

Note: This exam was originally administered digitally. It is presented here in a format optimized for teacher and student use in the classroom.

During the AP Exam administration, students have access to reference information. To see the reference information for this course, please visit AP Central: https://apcentral.collegeboard.org/exam-administration-ordering-scores/administering-exams/subject-specific/reference-information

1. This question involves the `Account` class, which is used to represent user accounts for a website. The `Account` class contains a helper method, `isAvailable`, which determines whether a username is available. You will write a constructor and a method in the `Account` class.

```
public class Account
{
   private String username; // To be initialized in part (a)

   /**
    *  Creates a username based on the parameter requestedName. If the
    *  username is unavailable, appends successive integers, beginning
    *  with 1, to requestedName until an available username is found,
    *  as described in part (a).
    */
   public Account(String requestedName)
   { /* to be implemented in part (a) */ }

   /**
    *  Returns true if the parameter str is an available username;
    *  returns false otherwise.
    */
   public static boolean isAvailable(String str)
   { /* implementation not shown */ }

   /**
    *  Returns a shortened version of username with each hyphen ("-")
    *  and the character before it removed, as described in part (b)
    *  Preconditions: username does not start or end with a hyphen.
    *                 username does not contain consecutive hyphens.
    *                 username.length() >= 2
    *  Postcondition: username is unchanged.
    */
   public String getShortenedName()
   { /* to be implemented in part (b) */ }

   /* There may be instance variables, constructors, and methods
      that are not shown. */
}
```

The following information applies to part A.

In part A, you will write the `Account` constructor, which creates a username based on the parameter `requestedName`.

A helper method, `isAvailable`, has been provided to determine whether a username is available. The `isAvailable` method returns `true` if its parameter is an available username and returns `false` otherwise.

If `requestedName` is an available username, the `Account` constructor will assign `requestedName` to the instance variable `username`. If not, the constructor will try different variations of `requestedName` until an available username is found and assigned to the instance variable `username`. The variations consist of `requestedName` followed by an integer, as in the following example.

- If `requestedName` is `"Luis-Cruz"` and this username is not available, then the constructor will check the availability of `"Luis-Cruz1"`, `"Luis-Cruz2"`, `"Luis-Cruz3"`, and so on, until an available username is found. The first available username found is assigned to `username`.
- If `requestedName` is `"PSmith"` and this username is available, then `"PSmith"` is assigned to `username`.

The following information applies to part B.

In part B, you will write the `getShortenedName` method, which returns a shortened version of the instance variable `username` with each hyphen (`"-"`) and the character immediately before it removed. If no hyphens appear in `username`, the value of `username` is returned.

For example, if `username` is `"Amy-Marie-Lin"`, `getShortenedName` should return `"AmMariLin"`.

As another example, if `username` is `"SammyB3"`, `getShortenedName` should return `"SammyB3"`.

Part A

Complete the `Account` constructor. You must use `isAvailable` appropriately in order to receive full credit.

```
/**
 *  Creates a username based on the parameter requestedName. If the
 *  username is unavailable, appends successive integers, beginning
 *  with 1, to requestedName until an available username is found,
 *  as described in part (a).
 */
public Account(String requestedName)
```

Part B

Complete method `getShortenedName`.

```
/**
 *  Returns a shortened version of username with each hyphen ("-")
 *  and the character before it removed, as described in part (b)
 *  Preconditions: username does not start or end with a hyphen.
 *                 username does not contain consecutive hyphens.
 *                 username.length() >= 2
 *  Postcondition: username is unchanged.
 */
public String getShortenedName()
```

2. The `Bottle` class, which you will write, represents a bottle that contains liquid.

`Bottle` objects are created by calls to a constructor with a `double` parameter that represents the bottle's capacity, in milliliters (mL), which is the maximum amount of liquid in the bottle. Assume that this value will be greater than or equal to 0. When `Bottle` objects are constructed, each is filled to its capacity.

The `Bottle` class contains an `updateAmount` method, which updates the amount of liquid in the bottle. This method has a `double` parameter that represents the amount of liquid, in mL, to be removed from the bottle, with a precondition that the parameter is always greater than zero and less than or equal to the amount of liquid remaining in the bottle. If updating the amount of liquid in the bottle would cause it to be less than 25% of the capacity, the bottle is filled and reset to the capacity. The `updateAmount` method returns a `double` that represents the amount remaining in the bottle, in mL, after the amount has been updated.

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `Bottle`.

Statement	Return Value (blank if no value)	Explanation
`double amt;`		
`Bottle water = new Bottle(1000.0);`		A `Bottle` object named `water` is constructed with a capacity of 1,000 mL of liquid.
`amt = water.updateAmount(400.0);`	`600.0`	400 mL are used, so 600 mL remain in the bottle.
`amt = water.updateAmount(100.0);`	`500.0`	100 mL are used, so 500 mL remain in the bottle.
`amt = water.updateAmount(300.0);`	`1000.0`	300 mL are used, so 200 mL remain in the bottle. Since this amount is less than 250 mL (25% of the capacity of 1,000 mL), the bottle is immediately filled and reset to the initial amount of 1,000 mL.
`Bottle shampoo = new Bottle(40.0);`		A `Bottle` object named `shampoo` is constructed with a capacity of 40 mL of liquid.
`amt = shampoo.updateAmount(30.0);`	`10.0`	30 mL are used, so 10 mL remain in the bottle.
`amt = shampoo.updateAmount(1.0);`	`40.0`	1 mL is used, so 9 mL remain in the bottle. Since this amount is less than 10 mL (25% of the capacity of 40 mL), the bottle is immediately filled and reset to the initial amount of 40 mL.

Write the complete `Bottle` class. Your implementation must meet all specifications and conform to the examples shown in the table.

3. The `CourseRecord` class is used to store information about a student enrolled in a course. A partial definition of the `CourseRecord` class is shown.

```
public class CourseRecord
{
   /**
    *  Returns a unique ID for the student associated with this
    *  CourseRecord
    */
   public String getStudentID()
   { /* implementation not shown */ }

   /**
    *  Returns a nonnegative integer representing the number of times the
    *  student associated with this CourseRecord has been absent in the
    *  course
    */
   public int getAbsences()
   { /* implementation not shown */ }

   /* There may be instance variables, constructors, and methods
      that are not shown. */
}
```

The `Attendance` class maintains two `ArrayLists` of `CourseRecord` objects, named `historyList` and `mathList`. A partial definition of the `Attendance` class is shown.

```
public class Attendance
{
   /* The students enrolled in a history course */
   private ArrayList<CourseRecord> historyList;

   /* The students enrolled in a math course */
   private ArrayList<CourseRecord> mathList;

   /**
    *  Returns the number of students who are enrolled in both
    *  the history course and the math course but have more
    *  absences in the history course than the math course
    *  Preconditions:
    *        No student ID appears multiple times in historyList.
    *        No student ID appears multiple times in mathList.
    *        historyList and mathList do not contain any null elements.
    *  Postcondition:
    *        historyList and mathList are unchanged.
    */
   public int moreHistoryThanMathAbsences()
   { /* to be implemented */ }

   /* There may be instance variables, constructors, and methods
      that are not shown. */
}
```

Write the `Attendance` method `moreHistoryThanMathAbsences`. The method should return the number of students who are enrolled in both the history course and the math course but have more absences in the history course than the math course.

For example, suppose `historyList` and `mathList` have the following contents.

`historyList`:

Student ID	"rc29"	"br98"	"dr03"	"ot32"	"sq98"	"ry00"
Number of Absences	1	1	2	2	3	1

`mathList`:

Student ID	"fr27"	"sq98"	"dr03"	"dk12"	"ot32"	"js33"	"ry00"
Number of Absences	2	1	2	1	1	0	3

In this example, there are four student IDs (`"dr03"`, `"ot32"`, `"sq98"`, and `"ry00"`) that appear in both `historyList` and `mathList`. Of these, only `"ot32"` and `"sq98"` have more absences in the history course than the math course, so the `moreHistoryThanMathAbsences` method should return `2`.

Complete method `moreHistoryThanMathAbsences`.

```
/**
 *  Returns the number of students who are enrolled in both
 *  the history course and the math course but have more
 *  absences in the history course than the math course
 *  Preconditions:
 *        No student ID appears multiple times in historyList.
 *        No student ID appears multiple times in mathList.
 *        historyList and mathList do not contain any null elements.
 *  Postcondition:
 *        historyList and mathList are unchanged.
 */
public int moreHistoryThanMathAbsences()
```

4. The `Space` class is used to represent the spaces on a game board. A partial definition of the `Space` class is shown.

```
public class Space
{
   /**
    *  Returns the color of the space
    */
   public String getColor()
   { /* implementation not shown */ }

   /**
    *  Returns the point value of the space
    */
   public int getPoints()
   { /* implementation not shown */ }

   /* There may be instance variables, constructors, and methods
      that are not shown. */
}
```

The `GameBoard` class maintains a two-dimensional array of `Space` objects that represents a game board. A partial definition of the `GameBoard` class is shown.

```
public class GameBoard
{
   private Space[][] board;

   /**
    *  Returns the point value of the row in board specified by the
    *  parameter. The point value is the sum of the points in the row,
    *  or two times the sum of the points in the row if all spaces in
    *  the row are the same color.
    *  Preconditions: No elements of board are null.
    *                 board has at least two rows and
    *                 at least two columns.
    *                 targetRow is a valid row index.
    */
   public int getPointsForRow(int targetRow)
   { /* to be implemented */ }

   /* There may be instance variables, constructors, and methods
      that are not shown. */
}
```

When an element of the two-dimensional array `board` is accessed, the first index is used to specify the row and the second index is used to specify the column.

Write the `GameBoard` method `getPointsForRow`. The method should return the point value of the spaces in the row specified by the parameter `targetRow`. The point value of a row is determined as follows.

- If not all spaces in the row are the same color, the point value for the row is the sum of the points in the row.
- If all spaces in the row are the same color, the point value for the row is two times the sum of the points in the row.

For example, suppose `board` has the following contents. For each element, the first value is the color and the second value is the point value.

	0	1	2	3	4
0	"orange" 100	"red" 100	"blue" 500	"green" 500	"red" 100
1	"red" 300	"blue" 200	"blue" 400	"red" 100	"green" 100
2	"red" 200	"red" 300	"red" 100	"red" 200	"red" 200
3	"green" 100	"blue" 100	"blue" 200	"blue" 100	"blue" 200

For these contents of `board`, the expected behavior of `getPointsForRow` is as follows.

- The call `getPointsForRow(0)` should return `1300`. Not all spaces in this row are the same color, so the sum of the points in the row is returned.
- The call `getPointsForRow(2)` should return `2000`. The sum of the points in row 2 is `1000`, and all spaces in this row are the same color, so two times this sum is returned.

Complete method `getPointsForRow`.

```
/**
 *  Returns the point value of the row in board specified by the
 *  parameter. The point value is the sum of the points in the row,
 *  or two times the sum of the points in the row if all spaces in
 *  the row are the same color.
 *  Preconditions: No elements of board are null.
 *                 board has at least two rows and
 *                 at least two columns.
 *                 targetRow is a valid row index.
 */
public int getPointsForRow(int targetRow)
```

STOP

END OF EXAM