AP® COMPUTER SCIENCE A
2016 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. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)

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 (**× • ÷ ≤ ≥ <> ≠**)
- `[]` vs. `()` vs. `<>`
- = instead of == and vice versa
- `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 ; where structure clearly conveys intent
- Missing { } where indentation clearly conveys intent
- 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 counter example, 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 2: Log Messages

Part (a) `LogMessage` constructor **2 points**

Intent: *Initialize instance variables using passed parameter*

+1 Locates colon

+1 Initializes instance variables with correct parts of the parameter

Part (b) `containsWord` **2 points**

Intent: *Determine whether description properly contains a keyword*

+1 Identifies at least one properly-contained occurrence of `keyword` in `description`

+1 Returns `true` if and only if `description` properly contains `keyword`
Returns `false` otherwise (*no bounds errors*)

Part (c) `removeMessages` **5 points**

Intent: *Remove log messages containing keyword from system log list and return these messages in a new list*

+1 Accesses all items in `messageList` (*no bounds errors; point lost if no removal attempted*)

+1 Identifies keyword-containing entry using `containsWord`

+1 Adds all and only identified entries to new list (*point lost if original order not maintained*)

+1 Removes all identified entries from `messageList` (*point lost if* `messageList` *reordered)*

+1 Constructs and returns `new ArrayList<LogMessage>`

AP® COMPUTER SCIENCE A
2016 CANONICAL SOLUTIONS

Question 2: Log Messages

Part (a):

```
public LogMessage(String message)
{
      int colon = message.indexOf(":");
      machineId = message.substring(0, colon);
      description = message.substring(colon + 1);
}
```

Part (b):

```
public boolean containsWord(String keyword)
{
   if (description.equals(keyword))
   {   return true;   }
   if (description.indexOf(keyword + " ") == 0)
   {   return true;   }
   if (description.indexOf(" " + keyword + " ") != -1)
   {   return true;   }
   if (description.length() > keyword.length())
   {
       if ((description.substring(description.length() -
                                  keyword.length() - 1).equals(
                                  " " + keyword)))
       {
          return true;
       }
   }
   return false;
}
```

Part (c):

```
public List<LogMessage> removeMessages(String keyword)
{
      List<LogMessage> removals = new ArrayList<LogMessage>();

      for (int i = 0; i < messageList.size(); i++)
      {
           if (messageList.get(i).containsWord(keyword))
           {
                removals.add(messageList.remove(i));
                i--;
           }
      }
      return removals;
}
```

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.

2Aa

(a) Write the constructor for the `LogMessage` class. It must initialize the private data of the object so that `getMachineId` returns the *machineId* part of the message and `getDescription` returns the *description* part of the message.

Complete the `LogMessage` constructor below.

```
/** Precondition: message is a valid log message. */
public LogMessage(String message)
{
    machineId = message.trim().substring(0, message.indexOf(":"));
    description = message.trim().substring(message.indexOf(":") +1);
}
```

Part (b) begins on page 10.

GO ON TO THE NEXT PAGE.

2Ab

Assume that the `LogMessage` constructor works as specified, regardless of what you wrote in part (a).

Complete method `containsWord` below.

```
/** Returns true if the description in this log message properly contains keyword;
 *          false otherwise.
 */
public boolean containsWord(String keyword)
{
   string s = description.trim();
   ArrayList<String> words = new Arraylist<String>();
   int i;
   while((i=s.indexOf(" ")) ~~=~~>-1){
        words.add(s.substring(0,i));
        s= s.substring(i+1);
   }
   if (s.length >0)
        words.add(s);
   for (String w:words)
        if (keyword.compareTo(w) = =0)
             return true;

   return false;
}
```

Part (c) begins on page 12.

GO ON TO THE NEXT PAGE.

2Ac

Assume that the `LogMessage` class works as specified, regardless of what you wrote in parts (a) and (b). You must use `containsWord` appropriately to receive full credit.

Complete method `removeMessages` below.

```
/** Removes from the system log all entries whose descriptions properly contain keyword,
 *  and returns a list (possibly empty) containing the removed entries.
 *  Postcondition:
 *    – Entries in the returned list properly contain keyword and
 *      are in the order in which they appeared in the system log.
 *    – The remaining entries in the system log do not properly contain keyword and
 *      are in their original order.
 *    – The returned list is empty if no messages properly contain keyword.
 */
public List<LogMessage> removeMessages(String keyword)
{
    List<Log Message> out=new ArrayList<Logmessage>();

    for (int i=0; i< messageList.size(); i++)
        if (messageList.get(i).containsWord(keyword)){
            Out.add(messageList.remove(i));
            i--;
        }
  return out;
}

/* sorry about the stray marks,
 * it's wierd doing this w/o an IDE
 * so I had to re-work it a few times... :(
 */
```

GO ON TO THE NEXT PAGE.

2Ba

(a) Write the constructor for the `LogMessage` class. It must initialize the private data of the object so that `getMachineId` returns the *machineId* part of the message and `getDescription` returns the *description* part of the message.

Complete the `LogMessage` constructor below.

```
/** Precondition: message is a valid log message. */
public LogMessage(String message)
{
int index = message.indexOf(":");
machineId = message.substring(0,index);
description = message.substring(index +1 );
}
```

Part (b) begins on page 10.

GO ON TO THE NEXT PAGE.

Assume that the `LogMessage` constructor works as specified, regardless of what you wrote in part (a).

Complete method `containsWord` below.

```
/** Returns true if the description in this log message properly contains keyword;
 *          false otherwise.
 */
public boolean containsWord(String keyword)
{
 if(description.indexOf(" "+keyword+" ")<0)
   return false;
 else
   return true;
}
```

Part (c) begins on page 12.

GO ON TO THE NEXT PAGE.

2Bc

Assume that the `LogMessage` class works as specified, regardless of what you wrote in parts (a) and (b). You must use `containsWord` appropriately to receive full credit.

Complete method `removeMessages` below.

```
/** Removes from the system log all entries whose descriptions properly contain keyword,
 *  and returns a list (possibly empty) containing the removed entries.
 *  Postcondition:
 *    – Entries in the returned list properly contain keyword and
 *      are in the order in which they appeared in the system log.
 *    – The remaining entries in the system log do not properly contain keyword and
 *      are in their original order.
 *    – The returned list is empty if no messages properly contain keyword.
 */
public List<LogMessage> removeMessages(String keyword)
```

```
{ ArrayList<LogMessage> final = new ArrayList<LogMessages>();

  for(int i=messageList.size()-1; i>=0; i--)
  { if(messageList.get(i).containsWord(keyword))
    { final.add(messageList.get(i));
      messageList.remove(i);
    }
  }
  return final;
}
```

GO ON TO THE NEXT PAGE.

2Ca

(a) Write the constructor for the `LogMessage` class. It must initialize the private data of the object so that `getMachineId` returns the *machineId* part of the message and `getDescription` returns the *description* part of the message.

Complete the `LogMessage` constructor below.

```
/** Precondition: message is a valid log message. */
public LogMessage(String message){
   int index= message.indexOf(":");
   machineId= message.subString(0, index);
   description= message.substring(index+1);

}
```

Part (b) begins on page 10.

GO ON TO THE NEXT PAGE.

2Cb

Assume that the `LogMessage` constructor works as specified, regardless of what you wrote in part (a).

Complete method `containsWord` below.

```
/** Returns true if the description in this log message properly contains keyword;
 *          false otherwise.
 */
public boolean containsWord(String keyword){
    int index = description.indexOf(keyword);
    if(index != -1){
        return true;

    }else{

        return false;

    }

}
```

Part (c) begins on page 12.

GO ON TO THE NEXT PAGE.

2Cc

Assume that the `LogMessage` class works as specified, regardless of what you wrote in parts (a) and (b). You must use `containsWord` appropriately to receive full credit.

Complete method `removeMessages` below.

```
/** Removes from the system log all entries whose descriptions properly contain keyword,
 *  and returns a list (possibly empty) containing the removed entries.
 *  Postcondition:
 *    – Entries in the returned list properly contain keyword and
 *      are in the order in which they appeared in the system log.
 *    – The remaining entries in the system log do not properly contain keyword and
 *      are in their original order.
 *    – The returned list is empty if no messages properly contain keyword.
 */
public List<LogMessage> removeMessages(String keyword){
  List<LogMessage> removed = new List<LogMessage>();
  for (int i = 0; i <  this.size(); i++){
     if ( this.get(i).getDescription. containsWord(keyword){

           this. remove(i);
           removed.add(this.get(i));

     }
   return removed;
 }
```

1

AP® COMPUTER SCIENCE A
2016 SCORING COMMENTARY

Question 2

Overview

This question tested the students' ability to use `String` methods from the AP Java subset to perform processing of both `String` parameters and instance variables. This problem also involved interacting classes.

In part (a) students were asked to write a constructor that split a `String` into two parts based on the colon position and initialized instance variables with those parts.

In part (b) students were asked to write a method to test a `String` for certain characteristics and return `true` or `false` depending on the result of those tests.

In part (c) students were asked to write a method to create, fill, and return a new list. They needed to traverse an existing list, use a previously implemented method to identify certain elements, and then delete the identified elements from the original list while adding them to a new list.

Sample: 2A
Score: 9

In part (a) the position of the `":"` in `message` is located correctly, and the two instance variables are initialized with the correct `message` substrings. The use of the `trim` method is inconsequential as `message` is assumed not to have leading or trailing spaces. Part (a) earned 2 points.

In part (b) `description` is assigned to the local variable `s`, and the use of the `trim` method is also inconsequential here. Then all properly-delimited substrings of `s` are determined, each of which is stored as an element of a new `ArrayList` of `String` objects named `words`. Lastly, `words` is traversed, and each element is compared to `keyword`. If a properly-delimited substring equal to `keyword` is found, then `true` is returned. Otherwise, after all the properly-delimited substrings are compared with `keyword`, the loop exits and `false` is returned. Part (b) earned 2 points.

In part (c) a new `ArrayList` named `out` is created to hold the `LogMessage` objects that are removed from `messageList`. Then `messageList` is traversed, and each element is used to call `containsWord` to determine if `keyword` is properly contained in its `description`. When `containsWord` returns `true`, the element is removed from `messageList` and added to `out`. The loop control index `i` is decremented so that no elements are skipped in the traversal. Part (c) earned 5 points.

Sample: 2B
Score: 7

In part (a) the position of the `":"` in `message` is located correctly, and the two instance variables are initialized with the correct `message` substrings. Part (a) earned 2 points.

In part (b) the `indexOf` method is used to determine if `keyword` is properly contained by surrounding spaces in `description`. This earned the "identifies one" point. However, the solution does not attempt to determine if `keyword` appears at the beginning or end of `description`. Part (b) earned 1 point.

In part (c) the `ArrayList final` is constructed to hold the `LogMessage` objects that are removed from `messageList`. A backward traversal of `messageList` is used to avoid skipping elements after

Question 2 (continued)

removals. The solution correctly removes elements and adds them to the end of the final list. However, the removed elements needed to be added to the front of final to account for the backward traversal, so the "adds all identified" point was not earned. Part (c) earned 4 points.

Sample: 2C
Score: 2

In part (a) the position of the **":"** in message is located correctly, and the two instance variables are initialized with the correct message substrings. Part (a) earned 2 points.

In part (b) no attempt is made to locate a properly-contained keyword in description. Part (b) did not earn any points.

In part (c) the "constructs and returns new ArrayList<LogMessage>" point was not earned because List is an interface which cannot be instantiated with new List. The solution attempts to traverse this instead of messageList. As a result, neither the "accesses all" point nor the "removes all identified" point was earned. Also, the "identify" point was not earned because containsWord is called using description instead of the messageList element. Lastly, there is no attempt to add an element to a new list, so the "adds all identified" point was not earned. Part (c) did not earn any points.