**AP® COMPUTER SCIENCE A**  
**2010 GENERAL SCORING GUIDELINES**

**Apply the question-specific rubric first.** To maintain scoring intent, a single error is generally accounted for only once per question thereby mitigating multiple penalties for the same error. The error categorization below is for cases not adequately covered by the question-specific rubric. Note that points can only be deducted if the error occurs in a part that has earned credit via the question-specific rubric. Any particular error is **penalized only once** in a question, even if it occurs on different parts of that question.

### Nonpenalized Errors
- spelling/case discrepancies if no ambiguity*
- local variable not declared if others are declared in some part
- use keyword as identifier
- [] vs. () vs. <>
- = instead of == (and vice versa)
- length/size confusion for array, String, and ArrayList, with or without ()
- private qualifier on local variable
- extraneous code with no side effect; e.g., precondition check
- common mathematical symbols for operators (x • ÷ ≤ ≥ <> ≠)
- missing {} where indentation clearly conveys intent and {} used elsewhere
- default constructor called without parens; e.g., new Fish;
- missing () on parameterless method call
- missing () around if/while conditions
- missing ; when majority are present
- missing public on class or constructor header
- extraneous [] when referencing entire array
- extraneous size in array declaration, e.g., int[size] nums = new int[size];

### Minor Errors (1/2 point)
- confused identifier (e.g., len for length or left() for getLeft())
- local variables used but none declared
- modifying new in constructor call
- using equals or compareTo method on primitives, e.g., int x; ...x.equals(val)
- array/collection access confusion ([] get)
- assignment dyslexia, e.g., x + 3 = y; for y = x + 3;
- super(method) instead of super.method
- formal parameter syntax (with type) in method call, e.g., a = method(int x)
- missing public from method header when required
- "false"/"true" or 0/1 for boolean values
- "null" for null

### Major Errors (1 point)
- extraneous code that causes side effect; e.g., information written to output
- interface or class name instead of variable identifier; e.g., Bug.move() instead of aBug.move()
- aMethod(obj) instead of obj.aMethod()
- attempt to use private data or method when not accessible
- destruction of persistent data (e.g., changing value referenced by parameter)
- use class name in place of super in constructor or method call
- void method (or constructor) returns a value

* Spelling and case discrepancies for identifiers fall under the “nonpenalized” category only if the correction can be unambiguously inferred from context; for example, “ArrayList” instead of “ArrayList”. As a counter example, note that if a student 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: Master Order

<table>
<thead>
<tr>
<th>Part (a)</th>
<th>getTotalBoxes</th>
<th>3 points</th>
</tr>
</thead>
</table>

**Intent:** Compute and return the sum of the number of boxes of all cookie orders in this.orders

- +1 Considers all CookieOrder objects in this.orders
- +1/2 Accesses any element of this.orders
- +1/2 Accesses all elements of this.orders with no out-of-bounds access potential
- +1 1/2 Computes total number of boxes
  - +1 Creates an accumulator (declare and initialize)
  - +1/2 Invokes getNumBoxes on object of type CookieOrder
  - +1/2 Correctly accumulates total number of boxes
- +1/2 Returns computed total

<table>
<thead>
<tr>
<th>Part (b)</th>
<th>removeVariety</th>
<th>6 points</th>
</tr>
</thead>
</table>

**Intent:** Remove all CookieOrder objects from this.orders whose variety matches cookieVar; return total number of boxes removed

- +4 Identifies and removes matching CookieOrder objects
- +1/2 Accesses an element of this.orders
- +1/2 Compares parameter cookieVar with getVariety() of a CookieOrder object (must use .equals or .compareTo)
- +1 Compares parameter cookieVar with getVariety() of all CookieOrder objects in this.orders, no out-of-bounds access potential
- +1/2 Removes an element from this.orders
- +1/2 Removes only matching CookieOrder objects
- +1 Removes all matching CookieOrder objects, no elements skipped
- +1 1/2 Computes total number of boxes in removed CookieOrder objects
  - +1 Creates an accumulator (declare and initialize)
  - +1/2 Invokes getNumBoxes on object of type CookieOrder
  - +1/2 Correctly accumulates total number of boxes (must be in context of loop and match with cookieVar)
- +1/2 Returns computed total

**Usage:**
- –1 consistently references incorrect name instead of orders, of potentially correct type
- –1 1/2 consistently references incorrect name instead of orders, incorrect type (e.g., this, MasterOrder)
AP® COMPUTER SCIENCE A  
2010 CANONICAL SOLUTIONS

Question 1: Master Order

Part (a):

```java
public int getTotalBoxes() {
    int sum = 0;
    for (CookieOrder co : this.orders) {
        sum += co.getNumBoxes();
    }
    return sum;
}
```

Part (b):

```java
public int removeVariety(String cookieVar) {
    int numBoxesRemoved = 0;
    for (int i = this.orders.size() - 1; i >= 0; i--) {
        if (cookieVar.equals(this.orders.get(i).getVariety())) {
            numBoxesRemoved += this.orders.get(i).getNumBoxes();
            this.orders.remove(i);
        }
    }
    return numBoxesRemoved;
}
```

// Alternative solution (forward traversal direction):

```java
public int removeVariety(String cookieVar) {
    int numBoxesRemoved = 0;
    int i = 0;
    while (i < this.orders.size()) {
        if (cookieVar.equals(this.orders.get(i).getVariety())) {
            numBoxesRemoved += this.orders.get(i).getNumBoxes();
            this.orders.remove(i);
        } else {
            i++;
        }
    }
    return numBoxesRemoved;
}
```

These canonical solutions serve an expository role, depicting general approaches to a 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.
(a) The `getTotalBoxes` method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns 0.

Complete method `getTotalBoxes` below.

```java
/** @return the sum of the number of boxes of all of the cookie orders */
public int getTotalBoxes()
{
    if (orders.size() == 0)
    {
        return 0;
    }

    int sum = 0;
    for (int i = 0; i < orders.size(); i++)
    {
        CookieOrder x = order.get(i);
        sum += x.getNumBoxes();
    }

    return sum;
}
```

Part (b) begins on page 6.
Complete method `removeVariety` below.

```java
/** *
 * Removes all cookie orders from the master order that have the same variety of *
 * cookie as `cookieVar` and returns the total number of boxes that were removed.
 * *
 * @param cookieVar  the variety of cookies to remove from the master order
 * @return  the total number of boxes of `cookieVar` in the cookie orders removed
 */
public int removeVariety(String cookieVar) {
    int num = 0;
    for (int i = orders.size() - 1; i >= 0; i--)
    
        CookieOrder x = orders.get(i);
    
        if (x.getVariety().equals(cookieVar))
    
            num += x.getNumBoxes();

        orders.remove(i)

    return num
```
(a) The `getTotalBoxes` method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns 0.

Complete method `getTotalBoxes` below.

```java
/** @return the sum of the number of boxes of all of the cookie orders */
public int getTotalBoxes()
{
    if (orders.length == 0)
        return 0;
    int sum = 0;
    for (int k=0; k<orders.length; k++)
        sum += orders[k];
    return sum;
}
```

Part (b) begins on page 6.
Complete method `removeVariety` below.

```java
/**
 * Removes all cookie orders from the master order that have the same variety of
 * cookie as `cookieVar` and returns the total number of boxes that were removed.
 * @param cookieVar the variety of cookies to remove from the master order
 * @return the total number of boxes of `cookieVar` in the cookie orders removed
 */
public int removeVariety(String cookieVar)
{
    int sum = 0;
    for (int i = 0; i < orders.length; i++)
    {
        if (orders[i].getVariety().equals(cookieVar))
        {
            sum += orders[i].getNumBoxes();
            orders.removeOrder(i);
        }
    }
    return sum;
}
```

GO ON TO THE NEXT PAGE.
(a) The `getTotalBoxes` method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns 0.

Complete method `getTotalBoxes` below.

```java
/** @return the sum of the number of boxes of all of the cookie orders */
public int getTotalBoxes()
{
    int sum = 0;
    for (int i = 0; i < orders.length; i++)
    {
        sum += orders[i];
    }
    return sum;
}
```

Part (b) begins on page 6.
Complete method removeVariety below.

/** Removes all cookie orders from the master order that have the same variety of
 *  cookie as cookieVar and returns the total number of boxes that were removed.
 *  @param cookieVar the variety of cookies to remove from the master order
 *  @return the total number of boxes of cookieVar in the cookie orders removed
 */
public int removeVariety(String cookieVar)
{
    int removed = 0;
    int j = 0;
    for (int i = 0; i < orders.length(); i++)
    {
        if (orders[i][2] == cookieVar)
        {
            removed++;
            j++;
        }
    }
    return removed;
}
Question 1

Overview

This question focused on the ArrayList data structure, element access and removal, algorithms that required processing all elements, and using instance data. Students were provided with the frameworks for two classes, CookieOrder and MasterOrder, and were asked to implement two methods in the MasterOrder class. In part (a) students were required to implement the method getTotalBoxes that returns the sum of the number of boxes of all of the cookie orders in the ArrayList instance variable. This could be accomplished by invoking getNumBoxes on each element of the list, accumulating and returning the sum. In part (b) students were required to implement the removeVariety method, which removes from the ArrayList instance variable all CookieOrder objects that have the same variety as the parameter, maintains an accumulator of the number of boxes removed, and returns the accumulator's final value. This could be accomplished by first invoking getVariety on each element of the list and performing a string comparison with the parameter. If the two strings match, the result of invoking getNumBoxes would be added to an accumulator and the remove method invoked to delete that order from the list. The accumulated total needed to be returned at the end of the method.

Sample: 1A
Score: 9

In part (a) the initial check for a zero-length list is unnecessary, but it does not cause a problem with the solution. The student correctly declares and initializes an accumulator. The student then correctly uses an indexed for-loop to access every element of the ArrayList, calls getNumBoxes on each element and accumulates the sum. When the loop ends, the sum is returned. Part (a) earned all 3 points.

In part (b) the student correctly declares and initializes an accumulator and uses a descending indexed for-loop to access every element of the ArrayList. The student correctly invokes getVariety on each CookieOrder and compares the result with the cookieVar parameter. If they match, the student invokes getNumBoxes and adds the result to the accumulator. The student then uses remove to delete that cookie order from the list. After the loop, the student returns the accumulated total number of boxes removed. Part (b) earned all 6 points.

Sample: 1B
Score: 6

In part (a) the initial check for a zero-length list is unnecessary, but it does not cause a problem with the solution. The student correctly declares and initializes an accumulator. The student then correctly uses an indexed for-loop to access every element of the ArrayList. However, the student does not call getNumBoxes on each element, and the accumulator is not counting boxes, so those two ½ points were not earned. A computed total is returned, earning ½ point. Part (a) earned 2 points.

In part (b) the reference orders[i] earned ½ point for "Accesses an element of this.orders." The call to getVariety and the comparison to the cookieVar parameter are done correctly. The student uses orders.removeOrder(i) instead of orders.remove(k) and so did not earn the ½ point for "Removes an element." The attempted removal is appropriately guarded and earned the ½ point for "Removes only matching CookieOrder objects." The student uses an ascending index for-loop without
the necessary index correction after removals and so did not earn the “Removes all matching” point. The call to `getNumBoxes` is good, as are the accumulation and the final return of the total number of boxes. Part (b) earned 4½ points.

The student uses `orders[k]` and `orders[i]` instead of `orders.get(k)` and `orders.get(i)` throughout the solution. This array/collection access confusion (`[ ] get`) lost ½ point under the General Scoring Guidelines.

**Sample: 1C**  
**Score: 3**

In part (a) the missing call to `getNumBoxes` did not earn the ½ point for “Invokes `getNumBoxes`” or the ½ point for “Correctly accumulates.” Because `sum` gets a calculated value beyond its initialization, the statement “`return sum;`” earned ½ point for “Returns computed total.” Part (a) earned 2 points.

In part (b) the reference `orders[i]` earned ½ point for “Accesses an element of this.orders.” There is no use of parameter `cookieVar` and there is no call to `getVariety`, so the student did not earn any of the score points for the comparison. There is no attempt to remove items from `orders`, so none of the points under “Removes” was earned. The statement “`int removed = 0;`” earned ½ point for “Creates an accumulator.” The missing call to `getNumBoxes` did not earn the ½ point for “Invokes `getNumBoxes`” or the ½ point for “Correctly accumulates.” Because `removed` gets a calculated value beyond its initialization, the statement “`return removed;`” earned ½ point for “Returns computed total.” Part (b) earned 1½ points.

The student uses `orders[i]` instead of `orders.get(i)` throughout the solution. This array/collection access confusion (`[ ] get`) lost ½ point under the General Scoring Guidelines.