COMPUTER SCIENCE A
SECTION II

Time—1 hour and 45 minutes
Number of questions—4
Percent of total score—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix 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 that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. An organization raises money by selling boxes of cookies. A cookie order specifies the variety of cookie and the number of boxes ordered. The declaration of the `CookieOrder` class is shown below.

```
public class CookieOrder
{
  /** Constructs a new CookieOrder object. */
  public CookieOrder(String variety, int numBoxes)
  {  /* implementation not shown */  }

  /** @return the variety of cookie being ordered
   */
  public String getVariety()
  {  /* implementation not shown */  }

  /** @return the number of boxes being ordered
   */
  public int getNumBoxes()
  {  /* implementation not shown */  }

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

The `MasterOrder` class maintains a list of the cookies to be purchased. The declaration of the `MasterOrder` class is shown below.

```
public class MasterOrder
{
   /** The list of all cookie orders */
   private List<CookieOrder> orders;

   /** Constructs a new MasterOrder object. */
   public MasterOrder()
   {  orders = new ArrayList<CookieOrder>();  }

   /** Adds theOrder to the master order.
    *  @param theOrder the cookie order to add to the master order
    */
   public void addOrder(CookieOrder theOrder)
   {  orders.add(theOrder);  }

   /** @return the sum of the number of boxes of all of the cookie orders
    */
   public int getTotalBoxes()
   {   /* to be implemented in part (a) */   }

   /** 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)
   {   /* to be implemented in part (b) */   }

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

(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.

```
   /** @return the sum of the number of boxes of all of the cookie orders
    */
   public int getTotalBoxes()
```

GO ON TO THE NEXT PAGE.

(b) The `removeVariety` method updates the master order by removing all of the cookie orders in which the variety of cookie matches the parameter `cookieVar`. The master order may contain zero or more cookie orders with the same variety as `cookieVar`. The method returns the total number of boxes removed from the master order.

For example, consider the following code segment.

```
MasterOrder goodies = new MasterOrder();
goodies.addOrder(new CookieOrder("Chocolate Chip", 1));
goodies.addOrder(new CookieOrder("Shortbread", 5));
goodies.addOrder(new CookieOrder("Macaroon", 2));
goodies.addOrder(new CookieOrder("Chocolate Chip", 3));
```

After the code segment has executed, the contents of the master order are as shown in the following table.

"Chocolate Chip" 1	"Shortbread" 5	"Macaroon" 2	"Chocolate Chip" 3

The method call `goodies.removeVariety("Chocolate Chip")` returns 4 because there were two Chocolate Chip cookie orders totaling 4 boxes. The master order is modified as shown below.

"Shortbread" 5	"Macaroon" 2

The method call `goodies.removeVariety("Brownie")` returns 0 and does not change the master order.

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)
```

GO ON TO THE NEXT PAGE.

2. An `APLine` is a line defined by the equation $ax + by + c = 0$, where *a* is not equal to zero, *b* is not equal to zero, and *a*, *b*, and *c* are all integers. The slope of an `APLine` is defined to be the `double` value $-a/b$. A point (represented by integers *x* and *y*) is on an `APLine` if the equation of the `APLine` is satisfied when those *x* and *y* values are substituted into the equation. That is, a point represented by *x* and *y* is on the line if $ax + by + c$ is equal to 0. Examples of two `APLine` equations are shown in the following table.

Equation	**Slope (–*a* / *b*)**	**Is point (5, -2) on the line?**
$5x + 4y - 17 = 0$	-5 / 4 = -1.25	Yes, because 5(5) + 4(-2) + (-17) = 0
$-25x + 40y + 30 = 0$	25 / 40 = 0.625	No, because -25(5) + 40(-2) + 30 $\neq$ 0

Assume that the following code segment appears in a class other than `APLine`. The code segment shows an example of using the `APLine` class to represent the two equations shown in the table.

```
APLine line1 = new APLine(5, 4, -17);
double slope1 = line1.getSlope();        // slope1 is assigned -1.25
boolean onLine1 = line1.isOnLine(5, -2); // true because 5(5) + 4(-2) + (-17) = 0


APLine line2 = new APLine(-25, 40, 30);
double slope2 = line2.getSlope();        // slope2 is assigned 0.625
boolean onLine2 = line2.isOnLine(5, -2); // false because -25(5) + 40(-2) + 30 ≠ 0
```

Write the `APLine` class. Your implementation must include a constructor that has three integer parameters that represent *a*, *b*, and *c*, in that order. You may assume that the values of the parameters representing *a* and *b* are not zero. It must also include a method `getSlope` that calculates and returns the slope of the line, and a method `isOnLine` that returns `true` if the point represented by its two parameters (`x` and `y`, in that order) is on the `APLine` and returns `false` otherwise. Your class must produce the indicated results when invoked by the code segment given above. You may ignore any issues related to integer overflow.

GO ON TO THE NEXT PAGE.

The declaration of the `Trail` class is shown below. You will write two unrelated methods of the `Trail` class.

```
public class Trail
{
  /** Representation of the trail. The number of markers on the trail is markers.length. */
  private int[] markers;

  /** Determines if a trail segment is level. A trail segment is defined by a starting marker,
   *  an ending marker, and all markers between those two markers.
   *  A trail segment is level if it has a difference between the maximum elevation
   *  and minimum elevation that is less than or equal to 10 meters.
   *  @param start the index of the starting marker
   *  @param end the index of the ending marker
   *         Precondition: 0 <= start < end <= markers.length - 1
   *  @return true if the difference between the maximum and minimum
   *          elevation on this segment of the trail is less than or equal to 10 meters;
   *          false otherwise.
   */
  public boolean isLevelTrailSegment(int start, int end)
  { /* to be implemented in part (a) */ }

  /** Determines if this trail is rated difficult. A trail is rated by counting the number of changes in
   *  elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
   *  with 3 or more such changes is rated difficult.
   *  @return true if the trail is rated difficult; false otherwise.
   */
  public boolean isDifficult()
  { /* to be implemented in part (b) */ }

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

(a) Write the `Trail` method `isLevelTrailSegment`. A trail segment is defined by a starting marker, an ending marker, and all markers between those two markers. The parameters of the method are the index of the starting marker and the index of the ending marker. The method will return `true` if the difference between the maximum elevation and the minimum elevation in the trail segment is less than or equal to 10 meters.

For the trail shown at the beginning of the question, the trail segment starting at marker 7 and ending at marker 10 has elevations ranging between 70 and 80 meters. Because the difference between 80 and 70 is equal to 10, the trail segment is considered level.

The trail segment starting at marker 2 and ending at marker 12 has elevations ranging between 50 and 120 meters. Because the difference between 120 and 50 is greater than 10, this trail segment is not considered level.

GO ON TO THE NEXT PAGE.

Complete method `isLevelTrailSegment` below.

```
/** Determines if a trail segment is level. A trail segment is defined by a starting marker,
 *  an ending marker, and all markers between those two markers.
 *  A trail segment is level if it has a difference between the maximum elevation
 *  and minimum elevation that is less than or equal to 10 meters.
 *  @param start the index of the starting marker
 *  @param end the index of the ending marker
 *         Precondition: 0 <= start < end <= markers.length - 1
 *  @return true if the difference between the maximum and minimum
 *               elevation on this segment of the trail is less than or equal to 10 meters;
 *               false otherwise.
 */
public boolean isLevelTrailSegment(int start, int end)
```

(b) Write the `Trail` method `isDifficult`. A trail is rated by counting the number of changes in elevation that are at least 30 meters (up or down) between two consecutive markers. A trail with 3 or more such changes is rated difficult. The following table shows trail elevation data and the elevation changes between consecutive trail markers.

Trail Elevation (meters)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Elevation	100	150	105	120	90	80	50	75	75	70	80	90	100

```
                   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /
Elevation change    50   -45    15   -30   -10   -30    25     0    -5    10    10    10
```

This trail is rated difficult because it has 4 changes in elevation that are 30 meters or more (between markers 0 and 1, between markers 1 and 2, between markers 3 and 4, and between markers 5 and 6).

Complete method `isDifficult` below.

```
/** Determines if this trail is difficult. A trail is rated by counting the number of changes in
 *  elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
 *  with 3 or more such changes is rated difficult.
 *  @return true if the trail is rated difficult; false otherwise.
 */
public boolean isDifficult()
```

GO ON TO THE NEXT PAGE.

4. This question involves reasoning about the GridWorld case study. Reference materials are provided in the Appendix.

   In this question, you will write two unrelated methods of the `GridChecker` class that will process a `BoundedGrid<Actor>` object. Recall that the `BoundedGrid` class implements the `Grid` interface. Also note that the methods in the `Grid` interface that return an array list will return an empty array list when no objects meet the return criteria.

   The declaration of the `GridChecker` class is shown below.

```
public class GridChecker
{
  /**   The grid to check; guaranteed never to be null   */
  private BoundedGrid<Actor> gr;

  /** @return an Actor in the grid gr with the most neighbors; null if no actors in the grid.
   */
  public Actor actorWithMostNeighbors()
  {   /* to be implemented in part (a) */   }

  /** Returns a list of all occupied locations in the grid gr that are within 2 rows
   *  and 2 columns of loc. The object references in the returned list may appear in any order.
   *  @param loc a valid location in the grid gr
   *  @return a list of all occupied locations in the grid gr that are within 2 rows
   *          and 2 columns of loc.
   */
  public List<Location> getOccupiedWithinTwo(Location loc)
  {   /* to be implemented in part (b) */   }

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

GO ON TO THE NEXT PAGE.

Question 1: Master Order

Part (a)	`getTotalBoxes`	**3 points**

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
- **+1/2** 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

Part (b)	`removeVariety`	**6 points**

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
- **+1/2** 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):

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

Part (b):

```
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):

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.