Apply the question-specific rubric first; the question-specific rubric always takes precedence.

**Penalties:** The penalty categorization below is for cases not covered by the question-specific rubric. Points can only be deducted in a part of the question that has earned credit via the question-specific rubric, and no section may have a negative point total. A given penalty can be assessed only once in a question, even if it occurs on different parts of that question. A maximum of 3 penalty points may be assessed over the entire question.

<table>
<thead>
<tr>
<th>Nonpenalized Errors</th>
<th>Minor Errors (½ point)</th>
<th>Major Errors (1 point)</th>
</tr>
</thead>
<tbody>
<tr>
<td>spelling/case discrepancies if no ambiguity*</td>
<td>confused identifier (e.g., len for length or left() for getLeft())</td>
<td>extraneous code that causes side effect; e.g., information written to output</td>
</tr>
<tr>
<td>local variable not declared if other variables are declared in some part</td>
<td>local variables used but none declared missing new in constructor call</td>
<td>interface or class name instead of variable identifier; e.g., Bug.move() instead of aBug.move()()</td>
</tr>
<tr>
<td>use of keyword as identifier</td>
<td>modifying a constant (final)</td>
<td>aMethod(obj) instead of obj.aMethod()</td>
</tr>
<tr>
<td>[ ] vs. () vs. &lt;&gt;</td>
<td>use of equals or compareTo method on primitives, e.g., int x; ...x.equals(val)</td>
<td>attempt to use private data or method when not accessible</td>
</tr>
<tr>
<td>= instead of == (and vice versa)</td>
<td>array/collection access confusion ([ ] get)</td>
<td>destruction of persistent data (e.g., changing value referenced by parameter)</td>
</tr>
<tr>
<td>length/size confusion for array, String, and ArrayList, with or without ()</td>
<td>assignment dyslexia, e.g., x + 3 = y; for y = x + 3; super (method()) instead of super.method()</td>
<td>use of class name in place of super in constructor or method call</td>
</tr>
<tr>
<td>private qualifier on local variable</td>
<td>formal parameter syntax (with type) in method call, e.g., a = method(int x)</td>
<td>void method (or constructor) returns a value</td>
</tr>
<tr>
<td>extraneous code with no side effect; e.g., precondition check</td>
<td>missing public from method header when required</td>
<td></td>
</tr>
<tr>
<td>common mathematical symbols for operators (x • x ≤ ≥ &lt; &gt; ≠)</td>
<td>&quot;false&quot;/&quot;true&quot; or 0/1 for boolean values</td>
<td></td>
</tr>
<tr>
<td>missing { } where indentation clearly conveys intent and { } used elsewhere</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default constructor called without parens; e.g., new Critter;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing ( ) on parameter-less method call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing ( ) around if/while conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing ; when majority are present</td>
<td></td>
<td></td>
</tr>
<tr>
<td>missing public on class or constructor header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extraneous [ ] when referencing entire array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[i,j] instead of [i][j]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extraneous size in array declaration, e.g., int[size] nums = new int[size];</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Spelling and case discrepancies for identifiers fall under the “nonpenalized” category only if the correction can be unambiguously inferred from context; for example, “ArayList” instead of “ArrayList”. As a counterexample, 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 3: Fuel Depot

Part (a)  nextTankToFill  5 points

**Intent:** Return index of tank with minimum level (<= threshold)

+4  Determine minimum element of tanks that is <= threshold, if any
+1½ Consider fuel levels of elements of tanks
+½  Accesses fuel level of an element of tanks
+½  Accesses at least one element of tanks in context of repetition (iteration/recursion)
+½  Accesses every element of tanks at least once
+2½ Identify minimum element of tanks that is <= threshold
+½  Compares fuel levels from at least two elements of tanks
+½  Implements algorithm to find minimum
+½  Identifies tank (object or index) holding identified minimum
+½  Determines element identified as minimum fuel level that is also <= threshold

+1  Return the index of the element satisfying the conditions, or the current index if no element does so
+½  Returns index of element identified as satisfying threshold & minimum conditions*
+½  Returns filler.getCurrentIndex() when no element satisfies conditions*

*Note: Point is not awarded if wrong data type is returned.

Part (b)  moveToLocation  4 points

**Intent:** Move robot to given tank location

+2  Ensure robot is pointing in direction of tank to be filled
+½  Determines direction filler is currently facing
+½  Changes filler’s direction for some condition
+1  Establishes filler’s direction as appropriate for all conditions

+2  Place robot at specified location
+½  Invokes moveForward method with a parameter
+½  Invokes moveForward method with a verified non-zero parameter
+1  Invokes filler.moveForward method with a correctly computed parameter
Question 3: Fuel Depot

Part (a):

```java
public int nextTankToFill(int threshold) {
    int minLevel = this.tanks.get(0).getFuelLevel();
    int minTankIndex = 0;
    for (int i = 1; i < this.tanks.size(); i++) {
        if (this.tanks.get(i).getFuelLevel() < minLevel) {
            minLevel = this.tanks.get(i).getFuelLevel();
            minTankIndex = i;
        }
    }
    if (minLevel <= threshold) {
        return minTankIndex;
    } else {
        return this.filler.getCurrentIndex();
    }
}

// Alternative solution
public int nextTankToFillA(int threshold) {
    int minTankIndex = this.filler.getCurrentIndex();
    for (int i = 0; i < this.tanks.size(); i++) {
        if (this.tanks.get(i).getFuelLevel() <= threshold &&
            this.tanks.get(i).getFuelLevel() <
            this.tanks.get(minTankIndex).getFuelLevel()) {
            minTankIndex = i;
        }
    }
    return minTankIndex;
}

Part (b):

```
Complete method `nextTankToFill` below.

```java
/**
 * Determines and returns the index of the next tank to be filled.
 * @param threshold fuel tanks with a fuel level \leq threshold may be filled
 * @return index of the location of the next tank to be filled
 * @postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
{
    int nextTank = filler.getCurrentIndex();
    int lowestLevel = threshold;
    FuelTank tank;
    for (int i = 0; i < tanks.size(); i++) {
        tank = tanks.get(i);
        if (tank.getFuelLevel() \leq lowestLevel) {
            lowestLevel = tank.getFuelLevel();
            nextTank = i;
        }
    }
    return nextTank;
}
```

Part (b) begins on page 16.
(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```java
/**
 * Moves the robot to location `locIndex`.
 * @param locIndex the index of the location of the tank to move to
 * @Precondition: 0 ≤ locIndex < tanks.size()
 * @Postcondition: the current location of the robot is `locIndex`
 */
public void moveToLocation(int locIndex)
{
    int currentBotPos = filler.getCurrentIndex();
    int movement = locIndex - currentBotPos;
    // if positive, move right; if negative, move left
    if (movement > 0) {
        // Make sure it's facing right
        if (!filler.isFacingRight()) filler.changeDirection();
    } else if (movement < 0) {
        if (filler.isFacingRight()) filler.changeDirection();
    } else { // movement == 0
        // Doesn't need to move
    }

    filler.moveForward(Math.abs(movement));
}
```

-16-
Complete method `nextTankToFill` below:

```java
/**
 * Determines and returns the index of the next tank to be filled.
 * @param threshold fuel tanks with a fuel level ≤ threshold may be filled
 * @return index of the location of the next tank to be filled
 * @postcondition the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
{
    for (int k=0; k < tanks.size(); k++)
        if (tanks.get(k).getFuelLevel() <= threshold)
            return k;
    return filler.getCurrentIndex();
}
```

Part (b) begins on page 16.
(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do _not_ move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```java
/** Moves the robot to location locIndex.
 * @param locIndex  the index of the location of the tank to move to
 * @see Precondition: 0 ≤ locIndex < tanks.size()
 * @see Postcondition: the current location of the robot is locIndex
 */
public void moveToLocation(int locIndex)
{
    int cur = filler.getCurrentIndex();  // the current location of the robot
    boolean right = filler.isFacingRight();
    if (cur == locIndex)
        return;
    if ((cur > locIndex && right) || (cur < locIndex && !right))
        filler.changeDirection();  // change direction if needed
    filler.moveForward(Math.abs(cur - locIndex));
} 
```

-16- GO ON TO THE NEXT PAGE.
Complete method `nextTankToFill` below.

```java
/**
 * Determines and returns the index of the next tank to be filled.
 * @param threshold fuel tanks with a fuel level \leq\ threshold\ may be filled
 * @return index of the location of the next tank to be filled
 * @postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
{
    int index = -1;
    int minimum = threshold;
    for (int x = 0; x < tanks.size(); x++)
    {
        if (tanks.get(x) <= minimum)
        {
            minimum = tanks.get(x);
            index = x;
        }
    }
    if (index == -1)
    {
        return locIndex;
    } else
    {
        locIndex = index;
        return locIndex;
    }
}
```

Part (b) begins on page 16.
(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```java
/**
 * Moves the robot to location `locIndex`.
 * @param locIndex the index of the location of the tank to move to
 * @Precondition: 0 ≤ locIndex < tanks.size()
 * @Postcondition: the current location of the robot is `locIndex`
 */
public void moveToLocation(int locIndex)
{
    if (locIndex ≥ 0 && locIndex < tanks.size())
    {
        // fill up
        move(locIndex);
```

-16- GO ON TO THE NEXT PAGE.
AP® COMPUTER SCIENCE A
2011 SCORING COMMENTARY

Question 3

Overview

This question involved the List interface and two provided interfaces: FuelTank and FuelRobot. It also contained a FuelDepot class, which represented a fuel depot that had a number of fuel tanks arranged in a line, and a robot that moved a filling mechanism back and forth along the line so that the tanks could be filled. The FuelDepot class contained two important instance variables: a FuelRobot named filler, and a List of FuelTanks named tanks. Students needed to use these variables to call methods that were specified in List and the two provided interfaces. The first part of this question focused on implementing an algorithm to traverse a List data structure and to find a minimum. The second part of this question focused on the state of the filler object. Students needed to query filler's state, utilize a combination of Boolean conditions to determine whether filler's state should be changed, and change filler's state as appropriate. Additionally, students had to satisfy a precondition of a method that they invoked (moveForward).

Two unrelated FuelDepot methods were to be implemented. In part (a) the method nextTankToFill was implemented and required the use of both filler and tanks instance variables and some of their methods as specified in the FuelRobot and FuelTank interfaces. Implementing nextTankToFill required an algorithm to find the minimum fuel level of any tank in the depot. This also required the use of List methods to access the elements of tanks. Finally, if the minimum fuel level was less than or equal to the value threshold, the index of the corresponding tank was returned. Otherwise, the index of the current tank was returned.

In part (b) the method moveToLocation was implemented and required the use of the filler instance variable. First, this method used the moveToLocation method's parameter and FuelRobot methods to determine the direction and number of spaces that filler must move. Then it used other FuelRobot methods to change direction and/or move filler as appropriate. It was important to be careful when calling FuelRobot's moveForward method so that the parameter was positive. This was a moveForward method precondition.

Sample: 3A
Score: 9

In part (a) the student uses a for loop to find the element of tanks that contains the least amount of fuel that is also less than or equal to the threshold. The variable lowestLevel contains the lowest fuel level found and nextTank contains the index of the element of tanks that contains that minimum fuel level. The nextTank index is returned. Note that initializing lowestLevel to threshold eliminates the need for a separate test to compare fuel levels with threshold. Also, initializing nextTank to filler.getCurrentIndex() eliminates the need for a check prior to the return. Part (a) earned all 5 points.

In part (b) filler is correctly used to call its methods. First, the location of filler is determined, and the number of locations to move is calculated. The sign of movement and filler's isFacingRight method are then used to determine whether filler is facing in the correct direction. If not, filler is turned by using filler.changeDirection. In the case where filler should not move, an explicit return is used to prevent calling moveForward with a 0 parameter. Otherwise, Math.abs is used to prevent calling moveForward with a negative parameter. Part (b) earned all 4 points.
Question 3 (continued)

Sample: 3B
Score: 6

Part (a) has two significant omissions. First, there is no attempt to find a minimum in the loop. Fuel levels are only being compared to the threshold. This lost the “compares fuel levels from at least two elements of tanks” ½ point as well as all ½-point items that mention minimum, for a total deduction of 2½ points. Also, there is an early return inside the loop. This lost the ½ point for “accesses every element of tanks” as well as all ½-point items that mention minimum, for a total deduction of 2½ points. Because the minimum ½ points could not be lost twice, these two omissions lost a total of 3 points. All possible remaining ½ points were earned for the proper use of getFuelLevel, accessing a tank in the context of repetition, comparing a tank’s fuel level to threshold, and returning filler.getCurrentIndex when no tank satisfies the conditions. Part (a) earned 2 points.

Part (b) has an explicit check that guards against calling moveForward with a 0 parameter. It also uses a single if statement to determine whether filler’s direction must be changed. Math.abs is used to prevent moveForward from being called with a negative parameter. Part (b) earned all 4 points.

Sample: 3C
Score: 2

In part (a) getFuelLevel is not used. This lost all ½-point items that mention fuel level, for a total deduction of 2 points. When students do not call getFuelLevel, the remaining items are scored as if the elements of tanks are fuel levels. As a result, this response earned each of the ½ points for “implements algorithm minimum,” “identifies tank holding minimum,” and “returns index of element.” The default “return” ½ point was lost because filler.getCurrentIndex is not returned. The array access operator [] is used instead of List’s get method. This very common array/collection access confusion results in a General Scoring Guidelines minor penalty of ½ point. Note that locIndex is not declared, but other local variables are properly declared, so that omission was not penalized. After the ½-point penalty was assessed, part (a) earned a total of 2 points.

Part (b) does not determine the direction filler is facing. This lost ½ point. It also does not determine the direction in which filler should face nor call filler.changeDirection when necessary. This lost 1½ points. The attempt to move filler earned none of the 2 possible placement points. The method name should be moveForward rather than move, and the parameter is not correct. This solution earned none of the points for part (b).