Free-Response Section

Scoring Guidelines

Applying the Scoring Criteria

Question scoring criteria include one or more algorithm points, which involve assembling the necessary pieces for a complete solution. Non-algorithm points are assessed with a narrower focus and can be earned even if other errors are present in the response. Algorithm points can be earned when other pieces are present but incorrect, but are not earned when required pieces are entirely omitted, when the pieces are assembled incorrectly, or when additional code not assessed in other points makes the solution incorrect.

Algorithm Penalties

Some responses include or omit elements that are not explicitly assessed in the rubric, but which make the answer substantially incorrect. Assess them as errors on the appropriate algorithm point. Such errors include:

- Array/collection access confusion (`[]` `.get`) when not otherwise assessed in the question
- Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
- Destruction of persistent data (e.g., changing value referenced by parameter)
- Void method or constructor that returns a value
- Provided method header rewritten with different number or type of parameters

No Penalty

Some errors are considered minor and either no penalty is assessed, or a penalty is only assessed when a scoring guideline explicitly requires it. Such errors include:

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared (unless scoring guidelines specifically requires it)
- Local variable declared in narrower scope than necessary (inner { })
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Provided method header rewritten with different variable names
- 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 counterexample, note that if the code declares* `"int G=99, g=0;"`*, then uses* `"while (G < 10)"` *instead of* `"while (g < 10)"`*, the context does **not** allow for the reader to assume the use of the lowercase variable.*

Question 1: Methods and Control Structures — 7 points

Canonical solution

(a) — 4 points

```
public String findWinner(int numLaps)
{
   double car1Time = 0;
   double car2Time = 0;

   for(int j = 1; j <= numLaps; j++)
   {
      car1Time += carOne.getLapTime(j);
      car2Time += carTwo.getLapTime(j);
   }

   if (car1Time < car2Time)
   {
      return "Car 1 Wins!";
   }
   else if(car2Time < car1Time)
   {
      return "Car 2 Wins!";
   }
   else
   {
      return "Tie!";
   }
}
```

(b) — 3 points

```
public String shortenMessage(String message, String str)
{
   String result = message;
   while (result.indexOf(str) >= 0)
   {
      int idx = result.indexOf(str);
      result = result.substring(0, idx) +
               result.substring(idx + str.length());
   }
   return result;
}
```

(a) `findWinner`

	Scoring Criteria	Decision Rules	
1	Calls `getLapTime` for at least one of `carOne` or `carTwo`	Responses **will not** earn the point if they • make any incorrect call to `getLapTime`	**1 point**
2	Loops `numLaps` times		**1 point**
3	Calculates the total amount of time for at least one race car	Responses **can** still earn the point even if they • fail to use the total Responses **will not** earn the point if they • fail to initialize a variable for the total • accumulate times for both cars into the same variable	**1 point**
4	Returns a correct string based on the calculated total time for each race car (*algorithm*)	Responses **will not** earn the point if they • fail to compare lap times • fail to return the string • print the string instead of or in addition to returning it	**1 point**
		Total for part (a)	**4 points**

(b) `shortenMessage`

	Scoring Criteria	Decision Rules	
5	Loop accesses all necessary parts of the message string (*no bounds error*)	Responses **will not** earn the point if they • use either a condition or an update that would cause an infinite loop or skipped elements in some cases	**1 point**
6	Identifies a location of `str` within `message` if there is one (and all `String` method calls are syntactically valid)	Responses **can** still earn the point even if they • fail to use the identified location	**1 point**
7	Builds and returns the correct string derived from first parameter with substring removed as many times as necessary (*algorithm*)	Responses **will not** earn the point if they • fail to return the string • print the string instead of or in addition to returning it	**1 point**
		Total for part (b)	**3 points**
		Total for question 1	**7 points**

Question 2: Class Design — 7 points

Canonical solution

7 points

```
public class CubePair
{
   private Cube cube1;
   private Cube cube2;

   public CubePair(String type)
   {
      cube1 = new Cube(type);
      cube2 = new Cube(type);
   }

   public CubePair(String type1, String type2)
   {
      cube1 = new Cube(type1);
      cube2 = new Cube(type2);
   }

   public String rollCubes()
   {
      String result = "";
      result += cube1.roll();
      result += cube2.roll();
      return result;
   }
}
```

`CubePair`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class CubePair`	Responses **will not** earn the point if they • declare the class as `private` • include extraneous code outside the class	**1 point**
2	Declares two `private Cube` instance variables to represent each cube	Responses **will not** earn the point if they • declare any instance variable `static` • declare a variable outside the class	**1 point**

3	Declares two constructor headers:
	`CubePair(String ___)`
	and
	`CubePair(String ___,`
	`String ___)`
		Responses **will not** earn the point if they
		• declare either constructor as `private`
			1 point

4	Constructor assigns correct initial values to all instance variables	Responses **will not** earn the point if they • fail to declare or initialize instance variables for the cubes • make any invalid call to the `Cube` constructor within the `CubePair` class	**1 point**
5	Declares method header: `String rollCubes()`	Responses **will not** earn the point if they • use incorrect method name • declare method as something other than `public`	**1 point**
6	Method calls `roll` on each cube and concatenates result (*algorithm*)	Responses **can** still earn the point even if they • fail to return the string representing the two rolls • declare instance variables outside the class, or in the class within a method or constructor • construct new cubes for each roll, as long as they are of the appropriate cube type	**1 point**
7	Method returns concatenated string	Responses **can** still earn the point even if they • return an incorrect string Responses **will not** earn the point if they • return a single `String` literal with no computation • return a non-`String` value • fail to return a value in some cases	**1 point**
		Total for question 2	**7 points**

Question 3: Data Analysis with `ArrayList` — 5 points

Canonical solution

5 points

```
public ArrayList<String> getHighRangeVehicles(int target)
{
   ArrayList<String> highRange = new ArrayList<String>();

   for (ElectricVehicle ev: vehicleList)
   {
      if (ev.getRange() > target)
      {
         boolean found = false;
         for (String s : highRange)
         {
            if (s.equals(ev.getModelName()))
            {
               found = true;
            }
         }
         if (!found)
         {
            highRange.add(ev.getModelName());
         }
      }
   }
   return highRange;
}
```

`getHighRangeVehicles`

	Scoring Criteria	Decision Rules	
1	Accesses* all vehicles in `vehicleList` (*no bounds errors*)	Responses **can** still earn the point even if they • fail to use the accessed value Responses **will not** earn the point if they • use an indexed loop with correct bounds but fail to access an item inside the loop • access the elements of `vehicleList` incorrectly • use array syntax to access `ArrayList` elements (`[ ]`)	**1 point**
2	Calls `getRange` and `getModelName` on an element from the list	Responses **can** still earn the point even if they • use array syntax to access `ArrayList` elements (`[ ]`) Responses **will not** earn the point if they • call either method on an object other than `ElectricVehicle`	**1 point**
3	Determines whether a vehicle has sufficient range and not already included in the list	Responses **can** still earn the point even if they • access `ElectricVehicle` or `ArrayList` incorrectly • call `getRange` incorrectly Responses **will not** earn the point if they • use incorrect syntax or logic for the condition (e.g., `>` in place of `>=`) • fail to call `getRange`	**1 point**
4	Compares model names of two vehicles	Responses **can** still earn the point even if they • call `String` methods incorrectly Responses **will not** earn the point if they • compare model names using `==` • fail to call `getModelName`	**1 point**
5	Returns list containing all and only necessary model names (*algorithm*)	Responses **can** still earn the point even if they • call `ElectricVehicle` or `ArrayList` methods incorrectly • use array syntax to access `ArrayList` elements (`[ ]`) • add model name to the list based on incorrect determination of whether the vehicle is in range Responses **will not** earn the point if they • fail to initialize the result list • fail to return the built `ArrayList` • modify `vehicleList`	**1 point**

*An enhanced for loop inherently accesses all elements of an `ArrayList`.

Total for question 3 **5 points**

Question 4: 2D Arrays — 6 points

Canonical solution

6 points

```
public int numberOfPies(double poundsPerPie)
{
   double weight = 0;

   for (int r = 0; r < apples.length; r++)
   {
      for (int c = 0; c < apples[0].length; c++)
      {
         boolean bad = apples[r][c].isRotten();

         if (r > 0 && apples[r - 1][c].isRotten())
         {
            bad = true;
         }
         if (r < apples.length - 1 &&
             apples[r + 1][c].isRotten())
         {
            bad = true;
         }
         if (c > 0 && apples[r][c - 1].isRotten())
         {
            bad = true;
         }
         if (c < apples[0].length - 1 &&
             apples[r][c + 1].isRotten())
         {
            bad = true;
         }

         if (!bad)
         {
            weight += apples[r][c].getWeight();
         }
      }
   }
   return (int)(weight / poundsPerPie);
}
```

`numberOfPies`

	Scoring Criteria	Decision Rules	
1	Accesses all elements of `apples`	Responses **can** still earn the point if they • access elements of `apples` out of bounds as part of adjacency checking, as long as loop bounds support traversing every element Responses **will not** earn the point if they • fail to access elements of `apples` correctly	**1 point**
2	Calls `isRotten` and `getWeight` on an `Apple` object	Responses **can** still earn the point even if they • access elements of `apples` out of bounds • fail to access elements of `apples` correctly Responses **will not** earn the point if they • call `isRotten` or `getWeight` incorrectly	**1 point**
3	Guards sum with a check for rotten apples	Responses **can** still earn the point even if they • fail to check all adjacent locations • combine the check for rotten apples with incorrect logic	**1 point**
4	Initializes sum variable and accumulates sum within a loop	Responses **will not** earn the point if they • declare the sum variable as `int`	**1 point**
5	Identifies non-usable apples (*algorithm*)	Responses **can** still earn the point even if they • call `isRotten` incorrectly Responses **will not** earn the point if they • fail to call `isRotten` • fail to check current location for rot • fail to check all adjacent locations • combine the check for rotten apples with incorrect logic • fail to guard access of adjacent elements to prevent bounds error	**1 point**
6	Returns correct integer number of pies (*algorithm*)	Responses **can** still earn the point even if they • identify non-usable apples incorrectly • call `getWeight` incorrectly Responses **will not** earn the point if they • return early • fail to return the computed value • fail to call `getWeight`	**1 point**
		Total for question 4	**6 points**