Practice Exam 2 Answers and Explanations

Part I Answers and Explanations (Multiple Choice)

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is B.

- The outer loop will start at h = 5 and then, on successive iterations, h = 3 and h = 1.
- The inner loop will start at k = 0, then 1 and 2.
- The `if` statement will print h + k only if their sum is even (since % 2 = 0 is true only for even numbers).
- Since h is always odd, h + k will be even only when k is also odd. That only happens once per inner loop, when k = 1. Adding h and k, we get 5 + 1 = 6, 3 + 1 = 4, and 1 + 1 = 2, so 6 4 2 is printed.

2. The answer is A.

- Let's picture our `ArrayList` as a table. After the first three add statements we have:

Hummus	Soup	Sushi

- The set statement changes element 1:

Hummus	Empanadas	Sushi

- Add at index 0 gives us:

Salad	Hummus	Empanadas	Sushi

- Remove the element at index 1:

Salad	Empanadas	Sushi

- And finally, add "Curry" to the end of the list:

Salad	Empanadas	Sushi	Curry

3. The answer is B.

- Option I is incorrect. The outer loop begins at i = letters.length. Remember that the elements of an array start at index = 0 and end at index = length − 1. Starting the loop at i = letters.length will result in an `ArrayIndexOutOfBoundsException`.
- Option II is correct. The outer loop takes each letter from the string in turn, and the inner loop prints it four times, as long as it is not an "S".
- Option III is incorrect. It correctly traverses each element in the array, but then only prints each element once, not four times.

4. The answer is D.

- The constructor call creates a new Boat and the super call sets the fuel variable in the `Vehicle` class to 20.
- The call to the `Boat` class start method:
  - begins by calling the `Vehicle` class start method, which prints "Vroom",
  - then calls useFuel, which calls the useFuel method in the Boat class, which calls the `Vehicle` class `changeFuel` method, which subtracts 2 from the fuel variable, and
  - finally, the start method prints "Remaining fuel" followed by the value in the `Vehicle` class fuel variable, which is now 18.

5. The answer is C.

- The type on the left side of the assignment statement can be the same type or a super type of the object constructed on the right side of the assignment statement. The object constructed should have an *is-a* relationship with the declared type.
- Option I is correct. A Boat *is-a* Boat.
- Option II is correct. A Boat *is-a* Vehicle.
- Option III is incorrect. Vehicle is a super class of Boat. A Boat *is* a Vehicle, but a Vehicle doesn't have to be a Boat. The super class must be on the left side of the assignment statement.

6. The answer is E.

- This is a recursive method. The first call results in:

  weird(3) = weird(3 − 2) + weird(3 − 1) + weird(3)
  = weird(1) + weird(2) + weird(3)

  A recursive method must approach the base case or it will infinitely recurse (until the Stack overflows). The call to weird(3) calls weird(3) again, which will call weird(3) again, which will call weird(3) again. . . .

7. The answer is D.

- It is important to notice that the `for` loop is traversing the array from the greatest index to the least. The array is correct if every element is greater than or equal to the one before it.
- We will return true if we traverse the entire array without finding an exception. If we find an exception, we return false, so we are looking for that exception, or values[index − 1] > values[index].
- Option A is incorrect because of the =.
- Note that since we must not use an index that is out of bounds, we can immediately eliminate any answer that accesses values[index + 1] which eliminates options B and E.

8. The answer is D.

- The first nested loop initializes all the values in the array in row-major order. The first value entered is 0, then 1, 2, etc. After these loops terminate, the matrix looks like this:

0	1	2
3	4	5
6	7	8

- The second nested loop (for-each loops this time) traverses the array and adds all the values into the sum variable. 0 + 1 + 2 + . . . + 8 = 36.

9. The answer is D.

- Option I is correct. It calls the super constructor, correctly passing on 2 parameters, and then sets its own instance variables, one to the passed parameter, one to a default value.
- Option II is incorrect. It attempts to call the student class 2 parameter constructor, passing default values, but the year is being passed as a String, not as an int.
- Option III is correct. There will be an implicit call to Student's no-argument constructor.

10. The answer D.

- Integer.MAX_VALUE is a large number, but repeatedly dividing by 2 will eventually result in an answer of 0 and the loop will terminate. Remember that this is integer division. Each quotient will be truncated to an integer value. There will be no decimal part.
- In the second loop, the initial condition sets i = 0, which immediately fails the condition. The loop never executes, so nothing is printed, but the code segment terminates successfully.

11. The answer is A.

- The for-each loop looks at every item in the ArrayList and adds it to sum if:
  - the value of the item > 1 (all values except 0 and 1) AND
  - the size of the list − 3 > the value of the item. Since the list has 10 elements, we can rewrite this as the value of the item < 7, which is true for all elements 6 and below.
  - Both conditions are true for 2, 3, 4, 5, 6; so the sum is 20.

12. The answer is A.

- The `indexOf(str)` method returns the index of the first occurrence or `-1` if not found.
- The index position begins counting at 0. "e" is found at position 9.

13. The answer is C.

- Both of the conditions in the original segment add 1 or 2, which is value, to count.
- The conditions can be combined using an || statement.

14. The answer is D.

- Option I is correct. Note that the final condition, (credit >= 590 && coll), doesn't have to be in parentheses. Since AND has a higher priority than OR, it would be executed first, even without the parentheses. However, the parentheses make the code clearer, and that is always our goal.
- Option II is incorrect. While it seems reasonable at first glance, De Morgan's theorem tells us we can't just distribute the !. We also have to change OR to AND.
- Option III is correct. As soon as a condition evaluates to true, the method will return and no more statements will be executed. Therefore no else clauses are required. The method will return false only if none of the conditions are true.

15. The answer is E.

- The loop will continue to execute until either value <= 5 or calculate = false.
- When we enter the loop, value = 33, calculate = true.
  - Since 33 % 3 = 0 we execute the first if clause and set value = 31.
  - 31 / 4 = 7, which is not < 3, so we do not execute the second if clause.
- The second iteration of the loop begins with value = 31 and calculate = true.
  - Since 31 % 3 != 0, we do not execute the first if clause.
  - Since 31 / 4 = 7, which is not < 3, we do not execute the second if clause.
- At this point, we notice that nothing is ever going to change. Value will always be equal to 31 and calculate will always be true. This is an infinite loop.

16. The answer is E.

- Remember, we only need to find one error to eliminate an option.
- Option A is incorrect. index starts in the middle of the array and goes down to zero. We swap the item at index with the item at index + 1. We are never going to address the upper half of the array.
- Option B is incorrect. The swap code is incomplete. We put planets[index] into s, but then never use s.
- Option C is incorrect. It's OK to use another array, but this code forgets to switch the order when it moves the elements into the new array. The new array is identical to the old one.
- Option D is incorrect. At first glance, there's no obvious error. The swap code looks correct. Although planets.length − 1 − index seems a bit complex, a few examples will show us that it does, indeed, give the element that should be swapped with the element at index: 0 → 7 − 0 = 7, 1 → 7 − 1 = 6, 2 → 7 − 2 = 5, 3 → 7 − 3 = 4, and so on. The trouble with this option is that "and so on" part. It swaps all the pairs and then continues on to swap them all again. After completing the four swaps listed above, we are done and we should stop.
- Option E is correct. The code is the same as option D, except it stops after swapping half the elements. The other half were the partners in those first swaps, so now everything is where it should be.

17. The answer is C.

- The loop begins at whatever value is stored in `start` and will continue until the value stored in `stop` is reached.
- Try substituting start = 1 and stop = 5. The loop will execute 5 times.
- Substituting 1 and 5 into the answer options, A, B, and D are eliminated.
- Choosing different values for start and stop such as 2 and 10 will result in option C.

18. The answer is B.

- The general form for generating a random number between *high* and *low* is

```
(int)(Math.random() * (high - low + 1)) + low
```

- high − low + 1 = 30, low = 20, so high = 49
- After the first statement, num is an integer between 20 and 49 (inclusive).
- In the second statement, we add 5 to num. Now num is between 25 and 54 (inclusive).
- In the third statement we divide by 5, remembering integer division. Now num is between 5 and 10 (inclusive).

19. The answer is A.

- This loop takes pieces of the String "computer" and places them in an ArrayList of String objects.
- The loop starts at index k = 0, and adds the word.substring(0), or the entire word, to the ArrayList. The first element of the ArrayList is "computer".
- Then k is incremented by the k++ inside the loop, k = 1, and immediately incremented again by the instruction in the loop header, k = 2. It is bad programming style to change a loop index inside the loop, but we need to be able to read code, even if it is poorly written.
- The second iteration of the loop is k = 2, and we add word.substring(2) or "mputer" to the ArrayList.
- We increment k twice and add word.substring(4) or "uter".
- We increment k twice and add word.substring(6) or "er".
- We increment k twice and fail the loop condition. The loop terminates and the `ArrayList` is printed.

20. The answer is E.

- We need to plug the values of x and y into the given statements to see if they always generate the correct return value.
- Option A is incorrect. 3 > 6 is false, but should be true.
- Option B is incorrect. 1 % 0 will fail with a division by zero error.
- Option C is incorrect. (1 + 0) % 2 == 0 is false, but should be true.
- Option D is incorrect. (3 + 6) % 3 == 6 is false, but should be true.
- Option E works for all the given values.

21. The answer is B.

- The loop will continue until numWord = words.size(), which will cause an `ArrayIndexOutOfBoundsException`.
- Notice that numWord is only incremented if nothing is removed from the `ArrayList`. That is correct because, if something is removed, the indices of the elements after that element will change. If we increment numWord each time, we will skip elements.

22. The answer is C.

- This is an especially difficult example of recursion because the recursive call is not the last thing in the method. It's also not the first thing in the method. Notice that the print happens *after* the call. So each iteration of the method will print one letter less than its parameter.
- Let's trace the code. The parts in *italics* were filled in on the way back up. That is, the calls in the plain type were written top to bottom until the base case returned a value. Then the answers were filled in *bottom to top.*

  wackyOutput("APCS") → *print "PCS"* (printed fourth)
  wackyOutput("PCS") → *print "CS"* (printed third)
  wackyOutput("CS") → *print "S"* (printed second)
  wackyOutput("S") → *print ""* (printed first)
  wackyOutput("") → Base Case. Return without printing.

  Remember that the returns are read bottom to top, so the output is "SCSPCS" in that order.
- Note that "S".substring(1,1) is not an error. It returns the empty string.

23. The answer is D.

- Option I is correct. It uses De Morgan's theorem to modify the boolean condition.

```
!(num1 >= 0 && num2 >= 0)
!(num1 >= 0 ) || !(num2 >= 0)
num1 < 0 || num2 < 0
```

- Option II is correct. It tests the two parts of the condition separately. If either fails, it returns "Numbers are not valid".
- Option III is incorrect. If num1 is a positive number and num2 is a negative number, but the absolute value of num2 < num1 (num1 = 5 and num2 = -2, for example), option III will return "Numbers are valid" although clearly both numbers are not >= 0.

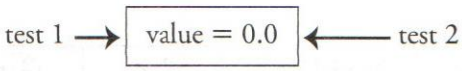
24. The answer is A.

- test1 references a `TestClass` object with value = 9.0 and test2 references a `TestClass` object with value = 17.5.

- After the addValue and reduceValue calls, our objects look like this:

- The assignment statement changes test2 to reference the same object as test1.

- At this point, the object with value = 19.0 is inaccessible (unless another reference variable we don't know about is pointing to it). The next time Java does garbage collection, it will be deleted.
- The next reduceValue call changes our object to this:



- The getValue statements in the next line get the same value, as test1 and test2 are pointing to the same object. 0.0 + 0.0 = 0.0.

25. The answer is E.

- Let's consider what the expression is telling us.

```
(truth1 && truth2) || ((!truth1) && (!truth2))
```

- The expression is true if both truth1 and truth2 are true OR if both truth1 and truth2 are false.
- Therefore, the expression is true as long as truth1 == truth2.

26. The answer is B.

This method implements a standard insertion sort. In this sort, each item is taken in turn and placed in its correct position among the items to its left (with lower indices).

- The first time through the loop, i = 1, the algorithm looks at the first two values {4, 3...} and inserts the 3 in the right place {3, 4...}. The rest of the array is untouched.
- The second time through, i = 2, the algorithm looks at the first three values {3, 4, 7...}. Since the 7 is already in the right place, no changes are made.
- The third time through, i = 3, the algorithm looks at the first four values {3, 4, 7, 6...} and inserts the 6 in the right place {3, 4, 6, 7...}. Our answer is {3, 4, 6, 7, 1, 5, 2}.
- If we were to continue, the final three values, {...1, 5, 2} would be inserted in their correct locations during the remaining 3 iterations of the outside loop.

Note: Answer C is the result of the third pass of a standard selection sort algorithm.

27. The answer is A.

- In order for a rectangle to contain a point, the following conditions must be true:

  topLeftX < x < bottomRightX and topLeftY < y < bottomRightY
- Option A is correct. These are the exact conditions used in option A, though they are in a different order.
- Option B is incorrect because it will return true if any one of the four conditions is true.
- Options C and D are incorrect because they attempt to make invalid calls to super or `Polygon` methods that do not exist.
- Option E is incorrect. It is similar to option B, except that it will return true if two of the required conditions are true.

28. The answer is A.

- Let's show the contents of the `ArrayList` graphically and trace the code. The method is called with this `ArrayList`.

Nora	Charles	Madeline	Nate	Silja	Garrett

- The first time through the loop, i = 3. The element at index 3 is removed and added to the end of the `ArrayList`.

Nora	Charles	Madeline	Silja	Garrett	Nate

- Decrement i, i = 2, which is >= 0, so execute the loop again. The element at index 2 is removed and added to the end.

Nora	Charles	Silja	Garrett	Nate	Madeline

- Decrement i, i = 1, which is >= 0, so execute the loop again. The element at index 1 is removed and added to the end.

Nora	Silja	Garrett	Nate	Madeline	Charles

- Decrement i, i = 0, which is >= 0, so execute the loop again. The element at index 0 is removed and added to the end.

Silja	Garrett	Nate	Madeline	Charles	Nora

- Decrement i, i = -1, which fails the loop condition. The loop terminates and the `ArrayList` is returned.

29. The answer is C.

- The loop will execute while varA != 0 OR varB > 0. Another way to say that, using De Morgan's theorem, is that the loop will *stop* executing when varA == 0 AND varB <= 0.
- The only answer that fits that condition is C: varA = 0 and varB = -4.
- Instead of thinking through the logic, we could trace the execution of the loop, but it's a lot of work. Here's a table of the values of varA and varB at the end of every iteration of the loop until the loop terminates.

varA	varB
-27	21
-24	18
-21	15
-18	12
-15	9
-12	6
-9	3
-6	0
-3	-3
0	-4

30. The answer is C.

- The nested loops go through grid in row-major order, but assign into newGrid in column-major order. (Notice that newGrid is instantiated with its number of rows equal to the grid's number of columns and its number of columns equal to the grid's number of rows.) After executing the loops, newGrid looks like this:

```
{ {0, 3, 6,  9 },
  {1, 4, 7, 10 },
  {2, 5, 8, 11 } }
```

- newGrid [2][1] = 5

31. The answer is E.

- The nested loops traverse the entire array. Each time through, the three assignment statements are executed. The key to the problem is the order in which these statements are executed. We need to pay attention to which values will overwrite previously assigned values.
- On entering the loops, our array looks like this:

0	0	0
0	0	0
0	0	0

- We will execute the outer loop three times, and for each of those three times, we will execute the inner loop three times.
- The first iteration of both loops, we are changing element [0][0].
  - The first statement makes it a 1.
    - Note that this statement changes the array in row-major order.
  - The second statement makes it a 2.
    - Note that this statement changes the array in column-major order.
  - The third statement makes it a 3, so it will remain a 3.
    - Note that this statement only changes the diagonals.
- Completing the first cycle of the inner loop, the first statement assigns a 1 to [0][1] and [0][2] and the second statement assigns a 2 to [1][0] and [2][0]. The third statement repeatedly sets [0][0] to 3, and since that statement is last, it will remain a 3.

3	1	1
2	0	0
2	0	0

- The next cycle of the inner loop will assign a 1 to [1][0], [1][1], and [1][2]. A 2 will be assigned to [0][1], [1][1], and [2][1]. Then element [1][1] will be overwritten with a 3.

3	2	1
1	3	1
2	2	0

- The third (and last) cycle of the inner loop will assign a 1 to [2][0], [2][1], and [2][2]. A 2 will be assigned to [0][2], [1][2], and [2][2]. Then element [2][2] will be overwritten with a 3.

3	2	2
1	3	2
1	1	3

32. The answer is E.

- If this were a sorting algorithm, there would be only one parameter, the array to be sorted, and a sorted array would be returned, so we can eliminate the sorting algorithms. In addition, Merge Sort is recursive, and selection and insertion sorts require nested loops, neither of which appear in this code.
- Binary search repeatedly divides the remaining section of the array in half. There's no code that does anything like that. This is not a binary search.
- This code goes through the array one step at a time in order. This is a sequential search algorithm.

33. The answer is E.

- Each pass of the loop needs to check one letter to see if it is the desired key.
- The `substring(i, i+1)` method is used to look at one letter at a time.
- Each time the letter matches the key, then `count` is incremented by 1.

34. The answer is D.

- The `compareTo(other)` method returns `0` if the `string` object is less than other (which means it comes first alphabetically)
- Strings cannot be compared using ==, < , or > operators.

35. The answer is D.

- `Math.random()` doesn't take any parameter, so options A and B can be eliminated.
- `Math.random()` returns a value greater than or equal to 0.0 and less than 1.0.
- Once the random `double` is returned, it must be multiplied by the size of the array and then cast into an `integer`.

36. The answer is D.

- Only I and III evaluate to true.

37. The answer is D.

- index starts at 0 and goes to the end of the String. We want a condition that will look at the letter in word at index, compare it to the letter parameter, and count it if it is the same.
- Option A is incorrect. The one-parameter substring includes everything from the starting index to the end of the word. We need one letter only.
- Option B is incorrect. We cannot compare Strings with ==.
- Option C is incorrect. indexOf will tell us whether a letter appears in a word and where it appears, but not how many times. It is possible to count the occurrences of letter using indexOf, but not this way. Option C does not change the if condition each time through the loop. It just asks the same question over and over.
- Option D is correct. It selects one letter at index and compares it to letter using the `equals` method, incrementing count if they match.
- Option E is incorrect. It will not compile. It tries to compare letter to the int returned by indexOf.

38. The answer is C.

- You might expect that the loop will remove any animal whose name comes before pink fairy armadillo lexicographically, but removing elements from an ArrayList in the context of a loop is tricky. Let's walk through it.
- We start with:

okapi	aye-aye	cassowary	echidna	sugar glider	jerboa

- The `for` loop begins, i = 0. Compare "okapi" with "pink fairy armadillo", and since "o" comes before "p", remove "okapi".

aye-aye	cassowary	echidna	sugar glider	jerboa

- The next iteration begins with i = 1. The remove operation has caused all the elements' index numbers to change. We skip "aye-aye", which is now element 0, and look at "cassowary", which we remove.

aye-aye	echidna	sugar glider	jerboa

- The next iteration begins with i = 2. Again the elements have shifted, so we skip "echidna", compare to "sugar glider" but that comes after "pink fairy armadillo" so it stays put leaving the ArrayList unchanged.

aye-aye	echidna	sugar glider	jerboa

- The next iteration begins with i = 3. Remove "jerboa".

aye-aye	echidna	sugar glider

- i = 4, which is not < animals.size(). We exit the loop and print the ArrayList.
- Remember that if you are going to remove elements from an ArrayList in a loop, you have to adjust the index when an element is removed so that no elements are skipped.
- Note that if we used the loop condition i < 6 (the size of the original ArrayList), then there would have been an `IndexOutOfBoundsException`, but because we used animals.size(), the value changed each time we removed an element.

39. The answer is E.

- Option I is incorrect. Among other errors, getSqFeet does not take a parameter.
- Option II is incorrect. Square brackets are used to access elements of arrays, but not of `ArrayLists`.
- Option III is correct.
- Option IV is incorrect. list is the name of the `ArrayList`, not the name being used for each element of the `ArrayList`.
- Option V is correct.

40. The answer is B.

- The array holds `Building` objects. As long as an object *is-a* Building, it can go into the array.
- The runtime environment will look at the type of the object and call the version of getSize written specifically for that object.
  - list[0] is a Building so the `Building` class getSize will be used to generate the String for list[0].
  - list[1] is a House so the `House` class getSize will be used to generate the String for list[1].
  - list[2] references the same object as list[1]. This will generate a duplicate of the list[1] response.

Part II Answers and Explanations (Free Response)

Please keep in mind that there are multiple ways to write the solution to a Free-response question, but the general and refined statements of the problem should be pretty much the same for everyone. Look at the algorithms and coded solutions, and determine if yours accomplishes the same task.

General Penalties (assessed only once per problem):
-1 using local variables without first declaring them
-1 returning a value from a void method or constructor
-1 accessing an array or `ArrayList` incorrectly
-1 overwriting information passed as a parameter
-1 including unnecessary code that causes a side effect such as a compile error or console output

1. Location Numerals

(a) **General Problem:** Write the `getLetterValue` method that returns the numerical value of the given letter.

Refined Problem: Find the parameter letter's position in the `String` alphabet. Return 2 raised to that power.

Algorithm:
- Use `indexOf` to find the position of letter in the `String` alphabet.
- Use `Math.pow` to raise 2 to the power of the position found.
- Return the result.

Java Code:

```
public int getLetterValue(String letter)
{
    int position = alphabet.indexOf(letter);
    return (int) Math.pow(2, position);
}
```

(b) **General Problem:** Write the `getDecimalValue` method that takes a Location Numeral and returns its decimal equivalent.

Refined Problem: Find the value of each letter in turn and add it to running total. Return the final answer.

Algorithm:
- Create a variable to hold the running total.
- Loop through the Location Numeral `String` from the letter at index 0 to the end of the `String`.
  - Isolate each letter, using `substring`.
  - Call the method `getLetterValue`, passing the isolated letter.
  - Add the result to the running total.
  - When the loop is complete, return the total.

Java Code:

```
public int getDecimalValue(String numeral)
{
   int total = 0;
   for (int i = 0; i < numeral.length(); i++)
   {
      String letter = numeral.substring(i, i + 1);
      total += getLetterValue(letter);
   }
   return total;
}
```

Common Errors:

- Don't end the loop at `numeral.length() - 1`. When you notice the expression `numeral.substring(i, i + 1)`, you may think that the i + 1 will cause an out of bounds exception. It would if it were the first parameter in the substring, but remember that the substring will stop *before* the value of the second parameter, so it will not index out of bounds. If you terminate the loop at `numeral.length() - 1`, you will miss the last letter in the `String`.
- Be sure that you use the method you wrote in part (a). There is generally a penalty for duplication of code if you rewrite the function in another method. The problem usually has a hint when a previously written method is to be used. In this case, the problem says, "You may assume that `getLetterValue` works as intended, regardless of what you wrote in part (a)." That's a surefire indication that you'd better use the method to solve the current problem.

(c) **General Problem:** Write the `buildLocationNumeral` method that returns the Location Numeral representation of the decimal value passed as a parameter.

Refined Problem: Determine which powers of 2 sum together to give the value of the parameter. Build the Location Numeral by determining the corresponding letters of the alphabet.

Algorithm:

- Initialize a variable to keep track of the position in the alphabet string starting at the end.
- Initialize a `string` variable to the empty string and add letters as appropriate.
- As long as more simplification can be done:
  - Determine the value of $2^{current}$ position in alphabet string.
    - If the power of 2 value is larger than the current value variable
    - Concatenate the corresponding alphabetic letter onto the string variable.
  - Decrease the value by the power of 2.
  - Decrease the position variable by 1.
- Return the final result.

Java Code:

```
public String buildLocationNumeral(int value)
{
   int position = 26;
   int powerOf2;
   String temp = "";
   while (value > 0)
   {
      powerOf2 = (int) Math.pow(2, position);
      if (value >= powerOf2)
      {    temp += alphabet.substring(position, position + 1);
           value -= powerOf2;
      }
     position --;
   }
   return temp;
}
```

Common Errors:

- This is a tricky piece of code. There are many places where errors can sneak in. Remember that you can earn most of the points for a question even with errors or missing sections in your code.
- The key is continuously subtracting powers of 2 from value until value becomes 0 and determining the letter of the alphabet that corresponds to that power of 2.

Scoring Guidelines: Location Numerals

Part (a) **getLetterValue** **2 points**

+1 Determines the correct position in the alphabet string
+1 Returns the correct value

Part (b) **getDecimalValue** **3 points**

+1 Accesses every individual letter in numeral; no bounds errors, no missing values
+1 Calls getLetterValue for each letter
+1 Accumulates and returns the correct total value

Part (c) **buildLocationNumeral** **4 points**

+1 Initializes appropriate variables
+1 Determines the appropriate power of 2 for each time through the loop
+1 Adds the appropriate letter of the alphabet to the string for each time through the loop
+1 Returns the accumulated string

Sample Driver:

There are many ways to write these methods. Maybe yours is a bit different from our sample solutions and you are not sure if it works. Here is a sample driver program. Running it will let you see if your code works and will help you debug it if it does not.

Copy the `LocationNumeralDriver` into your IDE along with the `LocationNumeral` class (including your solutions).

- Here's the `LocationNumeralDriver`:

```
public class LocationNumeralDriver
{
   public static void main(String[] args)
   {
   LocationNumeral num1 = new LocationNumeral();
   System.out.println(num1.getLetterValue("E"));
   System.out.println(num1.getDecimalValue("ECA"));
   System.out.println(num1.buildLocationNumeral(43));

   LocationNumeral num2 = new LocationNumeral();
   System.out.println(num2.getLetterValue("B"));
   System.out.println(num2.getDecimalValue("CBA"));
   System.out.println(num2.buildLocationNumeral(17));
   }
}
```

2. Quadratic

(a) **General Problem:** Write the `Quadratic` class.

Refined Problem: Create the `Quadratic` class with three instance variables and one three parameter constructor. The three methods `getDiscriminant`, `getRoot1`, and `getRoot2` will need to be defined.

Algorithm:

- Write a class header for `Quadratic`.
- Declare three instance variables to store the values of *a*, *b*, and *c*.
- Write a constructor with three parameters that represent the coefficients of the quadratic equation and assign the parameters to the corresponding instance variables.
- Write method `getDiscriminant`.
- Write methods `getRoot1` and `getRoot2`.

Java Code:

```
public class Quadratic
{
   private double a, b, c;

   public Quadratic(double a, double b, double c)
   {
      this.a = a;
      this.b = b;
      this.c = c;
   }

    public double getDiscriminant()
    {
      return b * b - 4 * a * c;
    }

    public int getRoot1()
    {
      return (-b + Math.sqrt(getDiscriminant()))/(2 * a);
    }

    public int getRoot2()
    {
      return (-b - Math.sqrt(getDiscriminant()))/(2 * a);
    }
}
```

Common Errors:

- The instance variables must be declared private.

Scoring Guidelines: Quadratic

Quadratic class **1 point**

+1 Complete, correct header for `Quadratic`

state maintenance **1 point**

+1 Declares at least three private instance variables capable of maintaining the coefficients of the quadratic equation

Quadratic Constructor **2 points**

+1 Correct method header
+1 Sets appropriate state variables based on parameters

getDiscriminant **2 points**

+1 Correct method header
+1 Returns the correctly computed value

getRoot1, getRoot2 **3 points**

+1 Correct method headers for `getRoot1` and `getRoot2`
+1 Correctly calculates and returns `root1`
+1 Correctly calculates and returns `root2`

Sample Driver:

There are many ways to write these methods. Maybe yours is a bit different from the sample solutions shown here and you are not sure if it works. Here is a sample driver program. Running it will let you see if your code works, and will help you debug it if it does not.

Copy `QuadraticDriver` into your IDE along with the complete `Quadratic` class (including your solutions). You also might want to add the `toString` method to your `Quadratic` class, which will print the quadratic equation for you.

```
public String toString()
{
  return a + "x^2 + " + b + "x + " + c;
}
```

```
public class QuadraticDriver
{
  public static void main(String[] args)
  {
      Quadratic q1 = new Quadratic(1, 0, -25);
      double discrim = q1.getDiscriminant();
      System.out.println("The quadratic equation " + q1 + " with discriminant " +
                         discrim);
      if (discrim > 0)
        System.out.println("has two real roots: " + q1.root1() + " and " +
                           q1.root2());
      else if (discrim == 0)
        System.out.println("has one real root: " + q1.root1());
      else
        System.out.println("has no real roots");
      System.out.println();

      Quadratic q2 = new Quadratic(1.2, 3.6, 2.7);
      discrim = q2.getDiscriminant();
      System.out.println("The quadratic equation " + q2 + " with discriminant " +
                         discrim);
      if (discrim > 0)
        System.out.println("has two real roots: " + q2.root1() + " and " +
                           q2.root2());
      else if (discrim == 0)
        System.out.println("has one real root: " + q2.root1());
      else
        System.out.println("has no real roots");
      System.out.println();

      Quadratic q3 = new Quadratic(2, -4.1, 5.2);
      discrim = q3.getDiscriminant();
      System.out.println("The quadratic equation " + q3 + " with discriminant " +
                         discrim);
      if (discrim > 0)
        System.out.println("has two real roots: " + q3.root1() + " and " +
                           q3.root2());
      else if (discrim == 0)
        System.out.println("has one real root: " + q3.root1());
      else
        System.out.println("has no real roots");
      }
}
```

3. Printing Factory

(a) **General Problem:** Write the `replacePaper` method of the `Machine` class.

Refined Problem: Take the new PaperRoll passed as a parameter and use it to replace the current roll. Return the used roll to the caller. Notice that this is a type of swapping and will require a temp variable.

Algorithm:

- Create a temp variable of type PaperRoll.
- Assign the machine's PaperRoll paper to temp.
- Assign the new PaperRoll passed as a parameter to the machine's PaperRoll.
- Return the PaperRoll in temp.

Java Code:

```
public PaperRoll replacePaper(PaperRoll pRoll)
{
   PaperRoll temp = paper;
   paper = pRoll;
   return temp;
}
```

Common Errors:

- It may seem like you want to do this:

  ```
  return paper;
  paper = pRoll;
  ```

  But once the return statement is executed, flow of control passes back to the calling method. The second statement will never be executed.

(b) **General Problem:** Write the `replacePaperRolls` method of the `PrintingFactory` class.

Refined Problem: Traverse the machines `ArrayList` checking for `Machine` object with `PaperRoll` objects that contain less than 4.0 meters of paper. Any `PaperRoll` objects containing less than 4.0 meters of paper need to be replaced. Get a new `PaperRoll` object from the newRolls `ArrayList`, and pass it to the `Machine` class `replacePaper` method. Place the returned used `PaperRoll` object on the usedRolls `ArrayList`.

Algorithm:

- Write a `for-each` loop to traverse the machines array.
  - If the current `Machine` object's `PaperRoll` object contains < 4.0 m of paper:
    - Take a `PaperRoll` object off of the newRolls `ArrayList`.
    - Call replacePaper passing the new roll as a parameter.
    - replacePaper will return the old `PaperRoll` object; put it on the usedRolls `ArrayList`.

Java Code:

```
public void replacePaperRolls()
{
   for (Machine m : machines)
   {
      if (m.getPaperRoll().getMeters() < 4.0)
      {
         PaperRoll newRoll = newRolls.remove(0);
         PaperRoll oldRoll = m.replacePaper(newRoll);
         usedRolls.add(oldRoll);
      }
   }
}
```

Common Errors:

- The syntax for accessing the amount of paper remaining is complex, especially when using a `for` loop instead of a `for-each` loop. Remember that the `getMeters` method is not a method of the `Machine` class. We have to ask each `Machine` object for access to its `PaperRoll` object and then ask the `PaperRoll` object how much paper it has left.

Java Code Alternate Solution #1:

Use `for` loops instead of `for-each` loops.

Use just one variable for the old and new rolls.

```
public void replacePaperRolls()
{
   for (int i = 0; i < machines.length; i++)
   {
      if (machines[i].getPaperRoll().getMeters() < 4.0)
      {
         PaperRoll roll = newRolls.remove(0);
         roll = machines[i].replacePaper(roll);
         usedRolls.add(roll);
      }
   }
}
```

Java Code Alternate Solution #2:

Combine the statements and eliminate the variables altogether. This could also be done in the `for` loop version.

```
public void replacePaperRolls()
{
   for (Machine m : machines)
      if (m.getPaperRoll().getMeters() < 4.0)
          usedRolls.add(m.replacePaper(newRolls.remove(0)));
}
```

(c) **General Problem:** Write the `getPaperUsed` method of the `PrintingFactory` class.

Refined Problem: The amount of paper used on each roll is 1000 minus the amount of paper remaining. First traverse the usedRolls `ArrayList` and add up the paper used. Next, traverse the machines array and add up the paper used. These two amounts represent the total paper used. Return the sum.

Algorithm:

- Write a `for-each` loop to traverse the usedRolls `ArrayList`.
  - Add 1000 minus paper remaining on the current roll to a running sum.
- Write a `for-each` loop to traverse the machines array.
  - Add 1000 minus paper remaining on the current machine's roll to the same running sum.
- Return the sum.

Java Code:

```
public double getPaperUsed()
{
   double sum = 0.0;
   for (PaperRoll p : usedRolls)
      sum = sum + (1000 - p.getMeters());

   for (Machine m : machines)
      sum = sum + (1000 - m.getPaperRoll().getMeters());

   return sum;
}
```

Java Code Alternate Solution #1:

Use `for` loops instead of `for-each` loops.

Replace sum = sum + ... with sum += ...

```
public double getPaperUsed()
{
   double sum = 0.0;
   for (int i = 0; i < usedRolls.size(); i++)
      sum += 1000 - usedRolls.get(i).getMeters();

   for (int i = 0; i < machines.length; i++)
      sum += 1000 - machines[i].getPaperRoll().getMeters();

   return sum;
}
```

Common Errors:

- Remember that `ArrayLists` and arrays use different syntax both to access elements and to find their total number of elements.

Java Code Alternate Solution #2:

There are other ways to compute the paper use. Here we add up all the leftover paper and then subtract the total from 1000 * the total number of rolls. This technique could be combined with the `for` loop version, as well as the `for-each` loop as shown here.

```
public double getPaperUsed()
{
   double totalLeft = 0.0;
   double totalUsed = 0.0;

   for (PaperRoll p : usedRolls)
      totalLeft += p.getMeters();

   for (Machine m : machines)
      totalLeft += m.getPaperRoll().getMeters();

   totalUsed = 1000 * (usedRolls.size() + machines.length) - totalLeft;

   return totalUsed;
}
```

Scoring Guidelines: Printing Factory

Part (a) **replacePaper** **2 points**

+1 Assigns the `PaperRoll` object passed as a parameter to the `Machine` object's PaperRoll variable; specifically: `paper = pRoll;`

+1 Returns the used `PaperRoll` object

Part (b) **replacePaperRolls** **3 points**

+1 Accesses all elements of machines; no bounds errors, no missed elements

+2 Processes low-paper rolls correctly

- +1 Removes a new roll from the newRolls `ArrayList`; calls the `replacePaper` method; passing the new roll as a parameter
- +1 Adds the used roll returned from the `replacePaper` method to the usedRolls `ArrayList`

Part (c) getPaperUsed 4 points

+1 Accesses the amount of paper remaining for all elements of the usedRolls `ArrayList`; no bounds errors, no missed elements

+1 Accesses all elements of the machines array; no bounds errors, no missed elements

+1 Accesses the amount of paper remaining on a `Machine` object's PaperRoll

+1 Correctly computes and returns total paper used

4. Hex Grid

(a) **General Problem:** Write the `getGamePieceCount` method that counts how many `GamePiece` objects are located on the grid.

Refined Problem: Traverse the 2D array of `GamePiece` objects and count the number of cells that are not null.

Algorithm:

- Create a variable to hold the count.
- Traverse the 2D array of values using nested loops.
  - Inside the loops, evaluate each cell of the grid to see if it is null. When we find an element that is not null, increment the count variable.
- Once the entire array has been traversed, the value of the count variable is returned.

Java Code:

```
public int getGamePieceCount()
{
   int count = 0;
   for (int row = 0; row < grid.length; row++)
      for (int col = 0; col < grid[row].length; col++)
         if (grid[row][col] != null)
            count++;
   return count;
}
```

Common Errors:

- Remember that the number of rows is grid.length and the number of columns is grid[row].length or, since all columns are the same length, grid[0].length.
- The return statement has to be outside both loops.

Java Code Alternate Solution:

You can use `for-each` loops to traverse a 2D array.

```
public int getGamePieceCount()
{
   int count = 0;
   for (GamePiece[] row : grid)
      for (GamePiece g : row)
         if (g != null)
            count++;
   return count;
}
```

(b) **General Problem:** Write the `isAbove` method that returns the `GamePiece` objects that are above an object at a given location on the grid.

Refined Problem: Create an `ArrayList` of `GamePiece` objects that are above the position passed in the parameter, where "above" is defined as having the same column number and a lower row number than another object. If there is no object at the location passed in, null is returned. If there are no objects above the parameter location, an empty `ArrayList` is returned.

Algorithm:

- Determine whether there is a `GamePiece` object at the location specified. If there is not, return null.
- Create an `ArrayList` to hold the `GamePiece` objects to be returned.
- Look in the specified column and the rows from 0 up to the specified row. If any of those cells contain `GamePiece` objects, add them to the `ArrayList`.
- Return the `ArrayList`.

Java Code:

```
public ArrayList<GamePiece> isAbove(int row, int col)
{
  if (grid[row][col] == null)
     return null;
  ArrayList<GamePiece> above = new ArrayList<GamePiece>();
  for (int r = 0; r < row; r++)
     if (grid[r][col] != null)
        above.add(grid[r][col]);
  return above;
}
```

Common Errors:

- Do not count the object itself. That means the loop must stop at r < row, not r <= row.

(c) **General Problem:** Write the `addRandom` method that adds a specified number of `GamePiece` objects to the grid in random locations.

Refined Problem: Check to see if there are enough blank cells to add the requested objects. If there are, use a random number generator to generate a row and column number. If the cell at that location is empty, add the object; otherwise, generate a new number. Repeat until all objects have been added.

Algorithm:

- Determine whether there are enough empty cells to hold the requested number of objects.
  - If there are not, return false.
- Loop until all objects have been added.
  - Generate a random int between 0 and number of rows.
  - Generate a random int between 0 and number of columns.
  - Check to see if that cell is free. If it is, add object and decrease number to be added by 1.
- Return true.

Note that this algorithm may take a very long time to run if the grid is large and mostly full. If that is the case, a more efficient algorithm is to put all available open spaces into a list and then choose a random element from that list.

Java Code:

```
public boolean addRandom(int number)
{
   if (getGamePieceCount() + number > grid.length * grid[0].length)
      return false;
   while (number > 0)
   {
      int row = (int)(Math.random() * grid.length);
      int col = (int)(Math.random() * grid[0].length);
      if(grid[row][col] == null)
      {
         grid[row][col] = new GamePiece();
         number--;
      }
   }
   return true;
}
```

Scoring Guidelines: Hex Grid

Part (a) **getGamePieceCount** **2 points**

+1 Compares every element to null; no bounds errors, no missed elements

+1 Returns the correct count

Part (b) **isAbove** **3 points**

+1 Returns null if the specified location is null

+1 Compares to null all elements in the given column with a row number from 0 up to but not including the given row number; no bounds errors, no missed elements

+1 Creates and returns `ArrayList` of all elements "above" the given location

Part (c) **addRandom** **4 points**

+1 Returns false if there is not sufficient room to add requested elements; must use getGamePieceCount; returns true after requested number of elements have been added

+1 Generates a random location; no bounds errors, no missed elements

+1 Adds the element only if the generated location is null; continues to generate locations in the context of a loop until an element is successfully added (a non-null location is found)

+1 Continues to add elements in the context of a loop until requested number of elements has been added