C212 Midterm Exam (80 points)
Mar 5, 2025

C212 Midterm Exam Rubric

1. (20 points) You're the owner of an ice-cream parlor in the middle of summer. You serve different flavors of ice-cream, alongside different toppings.

```
An IceCreamFlavor is one of:
- "Vanilla"
- "Chocolate"
- "Strawberry"

A Topping is one of:
- "Peanut"
- "Sprinkle"
```

Design the `double computeIceCreamCost(int c, int s, String f, String t)` method that, when given a number of cones c, a number of scoops per cone s, an `IceCreamFlavor` f, and a `Topping` t, returns the cost of the ice cream according to the following rules:

- Ice-cream cones have a "base" price of $3.50.
- If the `IceCreamFlavor` is `"Chocolate"` or `"Strawberry"`, add a charge of $0.75.
- If the `Topping` is `"Peanut"`, add a charge of $1.10. If it is `"Sprinkle"`, add a charge of $0.60.
- If the customer is buying more than 3 ice-cream cones, apply a 10% discount.
- Each scoop has a surcharge of $0.50.

In designing this method, follow the design recipe from class: write the signature, purpose statement, testing, and *then* do the implementation. You should probably use simple numbers for the inputs so you can calculate the values in your head. You only need to write one test for each flavor. You may assume that all inputs are well-formed: $c > 0$, $s > 0$, and f and t are one of the values in the above data definition.

The skeleton code is on the next page.

Solution.

Rubric:

- *(6 points)* +2 points per test. They must test each flavor. +0.5 points only for duplicated flavors.
- *(4 points)* +2 for the purpose. +0.25 for each `@param`. +1 for the `@return`
- *(1.5 points)* Base price is considered.
- *(1.5 points)* Charge is added for `"Chocolate"` or `"Strawberry"`.
- *(1 point)* Charge for `"Peanut"`.
- *(1 point)* Charge for `"Sprinkle"`.
- *(2.5 points)* More than 3 cones discount. -1 point if the percentage is applied incorrectly.
- *(2.5 points)* Surcharge.

```
class IceCreamTester {

  @Test
  void testComputeIceCreamCost() {
    assertEquals(6.85, computeIceCreamCost(1, 3, "Strawberry", "Peanut")); // 3.5 + 1.5 + .75 + 1.1(
    assertEquals(28.26, computeIceCreamCost(4, 6, "Chocolate", "Sprinkles")); // (6.85*4) * 0.9
    assertEquals(9.2, computeIceCreamCost(2, 1, "Vanilla", "Sprinkles")); //(4.6)*2
  }
}

class IceCream {

  /**
   * Computes the cost of an ice cream order.
   * @param c number of cones.
   * @param s number of scoops.
   * @param f flavor, one of "Vanilla", "Strawberry", "Chocolate".
   * @param t topping, one of "Peanut", "Sprinkles"
   * @return cost according to formula. Multiple cone orders are multiplied by c.
   */
  static double computeIceCreamCost(int c, int s, String f, String t) {
    double cost = 3.5;
    // Flavor check.
    if (f.equals("Strawberry") || f.equals("Chocolate")) { cost += 0.75; }

    // Topping check.
    if (t.equals("Peanut")) { cost += 1.1; }
    else { cost += 0.6; }

    // Scoops.
    cost += s * 0.5;
    // Multiply.
    cost *= c;
    // Discount?
    return c > 3 ? cost * 0.9 : cost;
  }
}
```

2. (30 points) This question has three parts.

   (a) Design the *standard recursive* `static String reverse(String s)` method that reverses a string s. This *must* be done using standard recursion. Do not use any data structures or other classes, e.g., `StringBuilder`, to reverse the string for you.

   Solution.

   Rubric:

   - *(3 points)* Correct base case. This can either be returning an empty string or broken up.
   - *(2 points)* Makes any recursive call on a smaller subproblem.
   - *(2 points)* Attempts to add any characters onto the outside of the string.
   - *(2 points)* Makes a recursive call on the *correct* subproblem.
   - *(1 point)* Correctly adds the first character onto the end of the recursive call.
   - *If the problem is not standard recursive, then award 0 points. This means they fundamentally did not understand the problem.*

```
static String reverse(String s) {
  if (s.length() <= 1) {
    return s;
  } else {
    return reverse(s.substring(1)) + s.charAt(0);
  }
}
```

   (b) Design the *tail recursive* `static String reverseTR(String s)` that solves the same problem as part (a), but instead uses tail recursion. You will need to design a helper method. Do not forget the relevant access modifier!

   Solution.

   Rubric:

   - *(1 point)* Driver method uses the correct accumulator.
   - *(1 point)* Helper method is private.
   - *(3 points)* Base case is correct.
   - *(2.5 points)* Method recurses correctly by shrinking the problem size or heading towards the base case in some way.
   - *(2.5 points)* Method correctly appends the first character onto the accumulator. If the order of `+` is flipped, award only +1.5 points.

```
static String reverseTR(String s) {
  return reverseTRHelper(s, "");
}

private static String reverseTRHelper(String s, String acc) {
  if (s.length() <= 1) {
    return acc + s;
  } else {
    return reverseTRHelper(s.substring(1), s.charAt(0) + acc);
  }
}
```

(c) Design the `static String reverseLoop(String s)` method that solves the same problem as (a) and (b), but instead uses a loop.

Solution.

Rubric:

- *(1 point)* Local variable declared.
- *(4 points)* Loop condition is correct.
- *(5 points)* Loop body is correct. Not sure how to award partial points. I imagine many people will just use a for loop in reverse order. That is fine.

```
static String reverseLoop(String s) {
  String acc = "";
  while (!(s.length() <= 1)) {
    acc = s.charAt(0) + acc;
    s = s.substring(1);
  }
  return acc + s;
}
```

3. (30 points) Design the generic `static <K, V> Map<K, Set<V>> orderByKeys(List<K> ks, List<V> vs, Comparator<K> cmp)` method that, when given a list of keys *ks* of type K, a list of values *vs* of type V, and a comparator *cmp* of type K, returns a map of keys to sets where the keys are ordered according to the comparator, and the elements of the sets are ordered according to their insertion order. Namely, the values are placed into buckets according to their respective keys. Values with the same keys are placed into the same bucket. Note that the i^{th} element of *ks* corresponds to the i^{th} element of *vs*. You may assume that $|ks| = |vs|$. (That is, the length of *ks* is the same as the length of *vs*).

   As an example, consider the following test case:

```
ks = [92, 85, 81, 92, 48, 1, 92, 48, 2, 85]
vs = ["Kyle", "Sujin", "Alan", "Peter", "Ransom",
      "Jack", "Simon", "Owen", "George", "Ian"]
cmp = Comparator.reverseOrder() // Sort in descending order.

orderByKeys(ks, vs, cmp) =>
  <92 : {Kyle, Peter, Simon},
   85 : {Sujin, Ian},
   81 : {Alan},
   48 : {Ransom, Owen},
    2 : {George},
    1 : {Jack}>
```

   You are not required to write test cases for this method. However, doing so may help you in its design.

   The skeleton code is on the next page.

Solution.

Rubric:

- Each blank is 2.5 points. Ask me about partial points.

```
class OrderByKeys {

  /**
   * Receives a list of keys, a list of values, and a comparator. Creates an
   * ordered map of sets, where the keys are ordered according to the given
   * comparator, and the values are sets of elements that are associated with
   * a shared key. The elements of those sets are ordered according to the
   * insertion order.
   * @param ks list of keys of type K.
   * @param vs list of values of type V.
   * @param cmp comparator to compare objects of type K.
   * @return map of keys of type k to values that are sets of type v.
   */
  static <K, V> Map<K, Set<V>> orderByKeys(List<K> ks,
                                           List<V> vs,
                                           Comparator<K> cmp) {
    Map<K, Set<V>> resMap = new TreeMap<>(cmp);
    for (int i = 0; i < ks.size(); i++) {
      Set<V> currSet = resMap.getOrDefault(ks.get(i), new LinkedHashSet<>());
      currSet.add(vs.get(i));
      resMap.put(ks.get(i), currSet);
    }
    return resMap;
  }
}
```

Scratch work

Scratch work