C212 Midterm Exam (80 points)
Mar 5, 2025

PLEASE READ ALL DIRECTIONS BEFORE STARTING YOUR EXAM. DO NOT OPEN UNTIL YOU ARE TOLD TO DO SO.

This is a closed-note exam aside from your aid sheet. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe," i.e., write out documentation comments and tests. Of course, doing so may aid you in creating your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

The exam has 80 total points. Answer all three problems for full credit.

When you are finished, turn in your exam and aid sheet if you have one, then quietly exit.

You have 75 minutes to complete the exam.

Good luck!

Question	Points	Score
1	20	
2	30	
3	30	
Total:	80	

Name: _____

IU Email: _____

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.

```
class IceCreamTester {

  @Test
  void testComputeIceCreamCost() {
    assertEquals(______, computeIceCreamCost(_____________________________________));
    assertEquals(______, computeIceCreamCost(_____________________________________));
    assertEquals(______, computeIceCreamCost(_____________________________________));
  }
}

class IceCream {

  /**
   *
   *
   * @param
   * @param
   * @param
   * @param
   * @return
   */
  static double computeIceCreamCost(int c, int s, String f, String t) {









  }
}
```

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.

```
static String reverse(String s) {




}
```

   (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!

```
static String reverseTR(String s) {


}
// Use the rest of the space to write your helper method!
```

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

```
static String reverseLoop(String s) {









}
```

Before proceeding...

For the last question, did you use the translation pipeline to solve the problem? It was not required; I am just curious!

_____ Yes

_____ No

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.

```
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.
   */
  __________ __________ _____________________ orderByKeys(List<K> ks,
                                                          List<V> vs,
                                                          Comparator<K> cmp) {
   Map<_________________> resMap = ________________________________________;
    for (int i = 0; i < ______________________; i++) {
      // If the key doesn't exist, return a new LinkedHashSet.
      Set<________> currSet = resMap.getOrDefault(ks.get(i), ______________________);
      currSet.add(_________________________________);
      resMap.put(_________________, _________________);
    }
    return _________________;
  }
}
```

Scratch work

Scratch work