C212 Midterm Exam (80 points)
Oct 11, 2023

C212 Midterm Exam Rubric

1. (20 points) Design the `computeDiscount(double itemCost, int age, boolean isStudent)` method that computes a discount for some item based on their age and student status according to the following criteria:

   - If $age < 18$, apply a 20% discount.
   - If $18 \leq age \leq 25$ and they are a student, apply a 25% discount. If they are not a student, do not apply a discount.
   - If $age \geq 65$ and they are a student, apply a 30% discount. If they are not a student, apply a 15% discount.
   - All other cases should not have a discount applied.

   Your method should return the total cost of the item after applying the discount. In designing this method, follow the template from class; write the signature, purpose statement, testing, and *then* do the implementation. You should probably use simple numbers for the `itemCost` so you can calculate the discounts in your head.

Solution.

Rubric:

- (1 pt) example when $age < 18$
- (1 pt) example when $18 \leq$ age ≤ 25 and is a student
- (1 pt) example when $18 \leq$ age ≤ 25 and is not a student
- (1 pt) example when age ≥ 65 and is a student
- (1 pt) example when age ≥ 65 and is not a student
- (1 pt) any other example not covered by the above cases
- (2 pts) purpose statement sensible
- (3 pts) signature is correct
- (5 pts) conditionals are correct (-1 for each incorrect up to -5).
- (4 pts) return statements are correct (-1 for each incorrect up to -4).

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class ComputeDiscountTester {

  @Test
  void computeDiscountTest() {
    assertAll(
            () -> assertEquals(16, ComputeDiscount.computeDiscount(20, 15, false)),
            () -> assertEquals(20, ComputeDiscount.computeDiscount(20, 24, false)),
            () -> assertEquals(15, ComputeDiscount.computeDiscount(20, 24, true)),
            () -> assertEquals(7, ComputeDiscount.computeDiscount(10, 70, true)),
            () -> assertEquals(8.5, ComputeDiscount.computeDiscount(10, 70, false)),
            () -> assertEquals(100, ComputeDiscount.computeDiscount(100, 30, true))
    );
  }
}

class ComputeDiscount {

  /**
   * Computes the discount of a given item based on the age
   * and whether the person is a student.
   */
  static double computeDiscount(double itemCost, int age, boolean isStudent) {
    if (age < 18) { return itemCost * .8; }
    else if (isStudent) {
      if (age <= 25) { return itemCost * .75; }
      else if (age >= 65) { return itemCost * .7; }
    } else {
      if (age >= 65) { return itemCost * .85; }
    }
    return itemCost;
  }
}
```

2. (20 points) This question has three parts.

Solution.

Rubric:

- (1 pt) uses standard recursive.
- (2 pts) correct signature.
- (3 pts) correct return values.

(a) *(6 points)* Design the *standard recursive* `countdown` method, which receives an `int` $n \geq 0$ and returns a `String` containing a sequence of the even numbers from n down to 0 inclusive, separated by commas.

```
countdown(10) => "10,8,6,4,2,0"
countdown(23) => "22,20,18,16,14,12,10,8,6,4,2,0"
countdown(0)  => "0"
```

```
static String countdown(int n) {
  if (n <= 0) { return "0"; }
  else if (n % 2 == 0) { return n + "," + countdown(n - 2); }
  else { return countdown(n - 1); }
}
```

Rubric:

- (1 pt) correct driver method.
- (1 pt) tail recursive method uses `private` access modifier.
- (2 pts) correct conditionals.
- (3 pts) correctly updates accumulator and n.

(b) *(7 points)* Design the `countdownTR` and `countdownTRHelper` methods. The former acts as the driver to the latter; the latter solves the same problem as `countdown` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

```
static String countdownTR(int n) {
  return countdownTRHelper(n, "");
}

private static String countdownTRHelper(int n, String acc) {
  if (n <= 0) { return acc + "0"; }
  else if (n % 2 == 0) { return countdownTRHelper(n - 2, acc + n + ","); }
  else { return countdownTRHelper(n - 1, acc); }
}
```

Rubric:

- (1 pt) correct signature.
- (1 pt) localized accumulator.
- (2 pts) correct loop condition.
- (2 pts) correctly updates local variables.
- (1 pt) correct return value.

(c) *(7 points)* Design the `countdownLoop` method, which solves the problem using either a `while` or `for` loop.

```
static String countdownLoop(int n) {
  String acc = "";
  while (!(n <= 0)) {
    if (n % 2 == 0) { acc += n + ","; n -= 2; }
    else { n -= 1; }
  }
  return acc + "0";
}
```

3. (20 points) Design the `moreThanThree` method that, when given an `int[]` A, returns a new `HashSet<Integer>` of values containing those values from A that occur strictly more than three times. **You cannot use the Stream API.** In designing this method, follow the template from class; write the signature, purpose statement, testing, and *then* do the implementation.

Solution.

Rubric:

- (4 pts) at least two coherent examples.
- (2 pts) sensible purpose statement.
- (14 pts) definition works as expected.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class MoreThanThreeTest {

  @Test
  void moreThanThree() {
    assertAll(
      () -> assertEquals(new HashSet<>(List.of(3, 4)),
                         moreThanThree(new int[]{3, 3, 3, 4, 3, 4, 4, 2, 4})),
      () -> assertEquals(new HashSet<>(),
                         moreThanThree(new int[]{2,3,4,5,6,7,7,7}))
    );
  }
}

import java.util.*;

class MoreThanThree {

  /**
   * Returns a HashSet of all the integers that occur more than
   * three times in the given array.
   */
  static HashSet<Integer> moreThanThree(int[] A) {
    HashMap<Integer, Integer> M = new HashMap<>();
    for (int v : A) {
      if (M.containsKey(v)) { M.put(v, M.get(v) + 1); }
      else { M.put(v, 1); }
    }
    HashSet<Integer> S = new HashSet<>();
    for (int v : M.keySet()) {
      if (M.get(v) > 3) { S.add(v); }
    }
    return S;
  }
}
```

4. (20 points) Oh no! Sam's cat, Marmalade, has scratched part of this exam away and we need you to fix the missing code. Fill in the blanks to complete this generic method implementation. Additionally, write at least two examples where each example uses a different key type. You can write an instance of a `LinkedHashMap` as {<k_1, v_1>, <k_2, v_2>, ..., <k_n, v_n>} in your examples to compensate for time.

Solution.

Rubric:

- (6 pts) at least two examples with different key types.
- (3 pts) `<K extends Comparable<K>>` *(Points were not deducted for not including the* `extends Comparable<K>`*.*
- (2 pts) `K, String`
- (3 pts) `K t`
- (6 pts) `(map.get(t).compareTo(max) > 0)`

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class FindMaxStringTest {

  @Test
  void findMaxString() {
    LinkedHashMap<Integer, String> m1 = new LinkedHashMap<>();
    m1.put(3, "Hi");
    m1.put(2, "H");
    m1.put(4, "John");
    m1.put(1, "");
    LinkedHashMap<String, String> m2 = new LinkedHashMap<>();
    m2.put("hi", "999");
    m2.put("hello", "HHHH");
    m2.put("369", "000000000000000");
    assertAll(
      () -> assertEquals("John", FindMaxString.findMaxString(m1)),
      () -> assertEquals("HHHH", FindMaxString.findMaxString(m2))
    );
  }
}

import java.util.LinkedHashMap;

class FindMaxString {

  /**
   * Returns the max string mapped to by arbitrary values.
   */
  static <K extends Comparable<K>> String findMaxString(LinkedHashMap<K, String> map) {
    String max = "";
    for (K t : map.keySet()) {
      if (map.get(t).compareTo(max) > 0) {
        max = map.get(t);
      }
    }
    return max;
  }
}
```