

CSCI-C 212 Exam 1 Spring 2026 (80 points)
Feb 11, 2026

C212 Exam 1 Rubric

1. (20 points) Design the double `computeFlightCost(String cabinType, String fare, int bags, int snacks, int drinks, boolean isElite)` method, which returns the cost of a flight ticket.

The `CabinType` defines the base cost of the ticket. There are three possible `CabinTypes`:

A `CabinType` is one of:

- "Economy"
- "Business"
- "First"

An "Economy" class cabin costs \$160; a "Business" class cabin costs \$280; a "First" class cabin costs \$500.

The `Fare` defines an additional percentage on top of the base cost.

A `Fare` is one of:

- "Basic"
- "Standard"
- "Flex"

The "Basic" fare adds an additional 20% to the base cost (i.e., the base cost is multiplied by 1.2); "Standard" adds an additional 50% to the base cost; "Flex" adds an additional 70% to the base cost.

The first and second bag each costs \$30. Any additional bags cost \$15. For example, a person carrying 3 bags would pay \$75 in bag fees.

Each snack costs \$3, each drink costs \$4. For a person in the "First" class cabin, however, all snacks and drinks cost \$0.

Finally, an "elite" passenger's ticket can never be more than \$1,000, even if the subtotal exceeds that amount.

In designing this method, follow the design recipe from class: write the signature, purpose statement, testing, and *then* do the implementation. You may assume that all inputs are well-formed.

For the tests, write THREE different tests that test the three `CabinTypes`. Delta values are not necessary.

The skeleton code is on the next page. You must fill in the Java documentation comment, the tests, and the method to receive full credit.

Hint: when writing test cases, we are not requiring you to simplify the expected value. So, you don't have to compute, for example, 70% of 280. If you write out the arithmetic expression that reduces to the correct value, we will mark it as correct.

Rubric:

- (6 pts) +2 for each test. The tests must be mathematically correct (or very close). If they duplicate a **CabinType**, it is not considered a correct test and should not earn the points.
- (2 pts) The Java documentation comment is sensible.
- (2 pts) +0.33 point for each parameter annotation. These need to be sensibly described.
- (-1 pt) If they don't have the return annotation, then deduct a point from the total score.
- (3 pts) +1 for each **CabinType** cost. The prices must be correct.
- (1.5 pts) +0.5 for each **Fare** type. The percentage must be applied to the base cost and be correct.
- (2 pts) +1 for handling two or fewer bags. +1 for handling more than 2 bags.
- (2.5 pts) +0.75 points for the snack cost. +0.75 for the drink costs. +1 for handling the case where the **CabinType** is "First".
- (1 pt) "Elite" passengers are clamped to being no more than \$1000.

```
class FlightCostTester {

    @Test
    void testComputeFlightCost() {
        assertEquals(160+32+75+3+4,
            Problem1.computeFlightCost("Economy", "Basic", 3, 1, 1, false));
        assertEquals(280+56+30+3+4,
            Problem1.computeFlightCost("Business", "Basic", 1, 1, 1, false));
        assertEquals(500+100+0,
            Problem1.computeFlightCost("First", "Basic", 0, 1, 1, false));
    }
}

class FlightCost {

    /**
     * Javadocs omitted for space.
     */
    static double computeFlightCost(String cabinType, String fare, int bags,
                                    int snacks, int drinks, boolean isElite) {

        double cost = 0;
        // Cabin
        if (cabinType.equals("Economy")) { cost += 160; }
        else if (cabinType.equals("Business")) { cost += 280; }
        else { cost += 500; }

        // Fare
        if (fare.equals("Basic")) { cost *= 1.2; }
        else if (fare.equals("Standard")) { cost *= 1.5; }
        else { cost *= 1.7; }

        // Bags
        if (bags > 2) { cost += 60 + (bags - 2) * 15; }
        else { cost += bags * 30; }

        if (!cabinType.equals("First")) { cost += drinks * 4 + snacks * 3; }

        return isElite ? Math.min(cost, 1000) : cost;
    }
}
```

2. (30 points) This question has three parts, each of which is weighted equally.

- (a) (10 points) Design the `String removeRunsOfLength3(String s)` method that, when given a string `s`, removes all runs of length 3 of the same character.

For example, `removeRunsOfLength3("aaabcddddeef")` returns `"bcdeef"`.

Your method must be standard recursive or you will receive no credit.

Rubric:

- (3 pts) Correct base case.
- (3.5 pts) Correct removals of runs of length 3.
- (3.5 pts) Correctly keeps non-runs.

```
static String removeRunsOfLength3(String s) {
    if (s.length() < 3) {
        return s;
    } else {
        char c1 = s.charAt(0);
        char c2 = s.charAt(1);
        char c3 = s.charAt(2);
        if (c1 == c2 && c2 == c3) {
            return removeRunsOfLength3(s.substring(3));
        } else {
            return c1 + removeRunsOfLength3(s.substring(1));
        }
    }
}
```

- (b) (10 points) Design the *tail recursive* `String removeRunsOfLength3TR(String s)` method that solves the same problem as (a), but uses tail recursion. You will need to design a helper method. Remember the relevant access modifiers!

Your helper method must be tail recursive or you will receive no credit.

Rubric:

- (1 pt) Driver method correctly calls helper method.
- (1 pt) Helper method is private.
- (2 pts) Correct base case.
- (3 pts) Correct removals of runs of length 3.
- (3 pts) Correctly keeps non-runs.

```
static String removeRunsOfLength3TR(String s) {
    return removeRunsOfLength3TRHelper(s, "");
}
private static String removeRunsOfLength3TRHelper(String s, String acc) {
    if (s.length() < 3) {
        return acc + s;
    } else {
        char c1 = s.charAt(0);
        char c2 = s.charAt(1);
        char c3 = s.charAt(2);
        if (c1 == c2 && c2 == c3) {
            return removeRunsOfLength3(s.substring(3), acc);
        } else {
            return removeRunsOfLength3(s.substring(1), acc + c1);
        }
    }
}
```

- (c) (10 points) Design the *iterative* `String removeRunsOfLength3Loop(String s)` method that solves the same problem as (a) and (b), but uses a loop.

Your method must use a loop and not be recursive or you will receive no credit.

Rubric:

- (1 pt) Correctly initializes an accumulator.
- (2 pts) Correct loop condition.
- (3 pts) Does not alter the accumulator in the runs case.
- (3 pts) Correctly updates the accumulator in the non-runs case.
- (1 pt) Correct return value.

```
static String removeRunsOfLength3Loop(String s) {
    String acc = "";
    while (!(s.length() < 3)) {
        char c1 = s.charAt(0);
        char c2 = s.charAt(1);
        char c3 = s.charAt(2);
        if (c1 == c2 && c2 == c3) {
            s = s.substring(3);
        } else {
            s = s.substring(1);
            acc = acc + c1;
        }
    }
    return acc;
}
```

3. (30 points) Design the `String removeOccurrences(String s, String m)` method that, when given two strings s and m , removes all occurrences of m in s .

You must account for overlapping occurrences. That is, suppose you call the method with the arguments `removeOccurrences("abcturkturkeyeycba", "turkey")`. When removing the instance "turkey", we get a new string "abcturkeycba" because we combine the strings "abcturk" and "eycba". The instance of "turkey" created by that concatenation should also be removed, producing "abccba".

In designing this method, follow the design recipe from class: write the signature, purpose statement, testing, and *then* do the implementation. You may assume that all inputs are well-formed.

For the tests, you must write at least THREE tests that comprehensively cover the inputs. It is up to you to figure out what that means!

Note: Your algorithm must use either recursion or a loop; you cannot use built-in methods that do the “heavy lifting” such as `replace` or `replaceAll`.

The skeleton code is on the next page. You must fill in the Java documentation comment, the tests, and the method to receive full credit.

Rubric:

- (5 points) Javadocs are correct.
- (5 points) Test cases are correct.
- (5 points) Handles the empty string.
- (5 points) Handles strings that do not contain *m*.
- (10 points) Handles all other cases. I'm not quite sure how to award partial credit, so just come talk to me if you find a solution that doesn't make sense.

```
/* Example recursive solution. */
static String removeOccurrences(String s, String m) {
    if (!s.contains(m)) {
        return s;
    } else {
        int occ = s.indexOf(m);
        String next = s.substring(0, occ) + s.substring(occ + m.length());
        return removeOccurrences(next, m);
    }
}

/* Example iterative solution. */
static String removeOccurrences(String s, String m) {
    while (s.contains(m)) {
        int occ = s.indexOf(m);
        s = s.substring(0, occ) + s.substring(occ + m.length());
    }
    return s;
}
```