CS 314 Final Review — Sum Tabs — **Solution**

Maps

Implement a method which, given a list of transactions at a certain kind of establishment on Sixth Street, returns the total tabs for each customer. The transactions will be passed into this method as a `List<String>` where each string will have the following format: `<Name> <Cost>` (Name will be a single word, Cost will be a double). Your method will return a `Map` which maps customers names to their total tab.

Complete the following method.

```
// Calculates the total tabs for each customer given a list of transactions
// pre: transactions != null
// post: A map which maps customer names to their total tab.
public static Map<String, Double> sumTabs(List<String> transactions) {
```

Here are some example calls to `sumTabs()`. (The resulting map does not have to be sorted like shown):

```
transactionList1 = ["A 5.50", "B 7.75", "A 7.90", "A 15.30", "B 2.25"]
sumTabs(transactionList1).toString() → {A=10.0, B=28.7}

transactionList2 = ["A 1.25", "B 10.15", "C 8.25", "D 7.00"]
sumTabs(transactionList1).toString() → {A=1.25, B=10.15, C=8.25, D=7.0}
```

You may create a `TreeMap` or `HashMap`.
You may also use the Java `Scanner` class's constructor, `next()`, and `nextDouble()` methods, or any `String` methods and the Double class's `parseDouble()` method.

Do not use any other Java classes or methods.

```
// Calculates the total tabs for each customer given a list of transactions
// pre: transactions != null
// post: A map which maps customer names to their total tab.
public static Map<String, Double> sumTabs(List<String> transactions) {
  HashMap<String, Double> result = new HashMap<>();
  for(String s : transactions){
      Scanner scan = new Scanner(s);
      String name = scan.next();
      double cost = scan.nextDouble();
      if(!result.containsKey(name))
          result.put(name, cost);
      else
          result.put(name, result.get(name) + cost);
  }
  return result;
}
```

This is a very straightforward map problem. Basically, we want to go transaction by transaction, parse it, and update the map. If it's the first time we've seen this customer's name, then we put a new entry in the map with the cost from the string. If we've seen the customer before, then we update their entry in the map.

Many students in the past have made the following mistake when updating values in maps.

```
if(result.containsKey(name))
  result.get(name) += cost;
```

The above code does not actually change the value associated with `name` in the map. Instead, it "gets" the value from the map associated with `name` which is of type Integer. Then, seeing that the `+=` operator is being applied to this Integer object, the object is automatically un-boxed into a primitive `int` and then `cost` is added to its value. However, the primitive `int` is never re-boxed into an Integer nor is it re-inserted into the map. Also, Integer objects are immutable so the only way to change the Integer value associated with `name` is to change the reference.