Solutions to Quiz 1 (October 25, 2017)

For this quiz, a **shopping list** is an **ordered** list of **unique** items together with a **positive integer** quantity for each item (we will not consider units for quantities, *e.g.* 1 oz. vs. 1 lb. vs. 1 box, just a number).

For example, someone who plans to buy a gallon of milk, a small bunch of bananas, and a box of Klondike bars, might have this shopping list:

- Milk (1)
- Banana (3)
- Klondike bars (1)

Shopping lists are ordered, so the following is a **different** shopping list. It has the same items and quantities, but in a different order:

- Banana (3)
- Klondike bars (1)
- Milk (1)

This is **not** a shopping list:

- Banana (1)
- Banana (1)
- Banana (1)
- Klondike bars (1)
- Milk (1)

And neither is this:

- Milk (0)

Problems 1–3 refer to the code for `FixedShoppingList` on page 5, which you may detach.
A client uses `FixedShoppingList` like this:

```
FixedShoppingList s = new FixedShoppingList(Arrays.asList(
        "Banana",
        "Milk",
        "Banana",
        "Klondike bars",
        "Banana"));
System.out.println(s);
```

which produces:

```
Banana (3)
Milk (1)
Klondike bars (1)
```

Notice that the bananas have ended up first on this list, and the Klondike bars have ended up last.

Problem 1 (Code review) (**20 points**).
For each of these code review comments on **`FixedShoppingList`**, circle AGREE or DISAGREE and explain why in one sentence.

(a) Line 21: in `toString()`, reduce the scope of `seen`.

Solution. DISAGREE, `seen` needs to be outside the loop. ■

(b) Lines 21–22: in `toString()`, make `seen` and `result` **`final`**.

Solution. AGREE, defend against reassignment, both still mutable. ■

(c) Line 3: declare `replist` as **`private`**.

Solution. AGREE, prevent rep exposure. ■

(d) Line 3: declare `replist` as **`final`**.

Solution. AGREE, defend against reassignment. ■

(e) Line 6: in the constructor, just use `Collections.unmodifiableList(items)` instead of `new ArrayList<>(items)`.

Solution. DISAGREE, good to use `unmodifiableList`, but still need a defensive copy to prevent rep exposure. ■

Problem 2 (Equality) (**16 points**).
Alyssa looks at **`FixedShoppingList`** and realizes that it should implement `equals()`.

She also sees that that just checking whether two `FixedShoppingList` instances have identical rep values will not be a correct implementation.

(a) Write two different rep values for `FixedShoppingList` that represent the same abstract value according to the code and specs given (*e.g.*, `toString`):

Solution. For example, `[ "Apple", "Apple", "Banana"]` and `[ "Apple", "Banana", "Apple"]` ■

(b) And write clearly and completely their abstract value:

Solution. For those rep values, the shopping list:

- Apple (2)
- Banana (1)

■

Suppose we implement `equals()` correctly, and now wish to implement `hashCode()`. Remember that if two objects are equal, then they must have the same hashcode. For each option below for the body of `hashCode()`, circle CORRECT or INCORRECT and explain why in one sentence.

(c)
```
return replist.hashCode();
```

Solution. INCORRECT: if two `FixedShoppingList` instances with different `replist` values are equal, those different `List` objects may have different hashcodes, and we violate the spec. ■

(d)
```
if (replist.isEmpty()) { return 0; }
else { return replist.get(0).hashCode(); }
```

Solution. CORRECT: the first item on a `FixedShoppingList` always appears first in `replist`. Equal `FixedShoppingList` instances must either both be empty or both have the same first item, and this implementation will return the same hashcodes for them. ■

Problem 3 (Immutable ADT) (**22 points**).

Reminder: please write neatly. Make sure each answer satisfies *all* the requirements of the question.

(a) Write an abstraction function for **`FixedShoppingList`** that works with the code and specs given.

Solution. Represents the shopping list whose items are the unique elements in `replist`, in the order that they first appear in `replist`, and whose quantities are the number of times they appear in `replist`. ■

(b) Write *requires* and *effects* to create a **stronger** spec for `withAnother()` (compared to the one in the code, lines 7–8) that is still satisfied by the given implementation.

In addition, your spec must allow this as a valid test case:

`[ Milk (1) ].withAnother("Cookies") = [ Milk (1), Cookies (1) ]`

where we check the items, their order, and their quantities in the result.

Solution. The spec must be strong enough to admit the test. For example,

requires: true

effects: returns a new shopping list identical to this shopping list except that the quantity of `itemToAdd` is `this.howMany(itemToAdd)+1`, and if `itemToAdd` is not on this list, it is at the end of the returned list ■

(c) Write *requires* and *effects* to create a **weaker** spec for `howMany()` (compared to the one in the code, lines 12–13) that is still satisfied by the given implementation.

In addition, your spec must allow this as a valid test case:

`[ Milk (1) ].howMany("Milk") = 1`

Solution. The spec must be weaker than in the code, but strong enough to admit the test. For example,

requires: `item` is on this list

effects: returns the quantity of `item` on this list ■

Problem 4 (Mutable ADT) (**18 points**).

Problems 4–6 refer to the code for `MutableShoppingList` on page 6.
A client uses `MutableShoppingList` like this:

```
MutableShoppingList m = new MutableShoppingList();
m.another("Banana");
m.another("Milk");
m.another("Banana");
m.another("Klondike bars");
m.another("Banana");
System.out.println(m);
```

which produces:

```
Banana (3)
Klondike bars (1)
Milk (1)
```

Ben looks at `MutableShoppingList`. "Alyssa, how does this `TreeMap` work?"
"It keeps its keys sorted in alphabetical order."
"OK," says Ben.

(a) Write *requires* and *effects* to create a **stronger** spec for the **`MutableShoppingList()`** constructor (compared to the one in the code, lines 4–5) that is still satisfied by the given implementation.

Solution. For example,

requires: true

effects: returns a new empty shopping list ■

(b) Write one statement to include in the rep invariant for `MutableShoppingList` that works with the code and specs given, and that 6.031 does **not** already assume implicitly.

Solution. All values in `itemCounts` are positive ■

(c) And write one piece of the rep invariant that 6.031 **does** already assume implicitly.

Solution. `itemCounts`, its keys, and its values are all non-null ■

Problem 5 (Testing and Operations) (**24 points**)**.**
Alyssa looks at **`MutableShoppingList`**. "This `howMany` method has a bug! Did anyone test this thing?"
Devise a testing strategy for the `howMany()` operation:

(a) Write one good three-part partition on the method's implicit input only, ignoring `item`.

Solution. This list has 0, 1, or more than 1 item ■

(b) Write one good three-part partition on the relationship between the method's two inputs.

Solution. This list does not contain `item`, has 1 of `item`, or more than 1 of `item` ■

`MutableShoppingList` uses `SortedMap` and `TreeMap`.

(c) In one sentence, what is the relationship between the **types** `SortedMap` and `TreeMap`?

Solution. `TreeMap` is a subtype of `SortedMap` ■

(d) In one sentence, what is the relationship between the **specs** of `SortedMap` and `TreeMap`?

Solution. `TreeMap` is stronger or equal to `SortedMap` ■

One **creator** ADT operation used in the `MutableShoppingList` code is the `TreeMap` constructor (line 6).

For each other kind of ADT operation in our taxonomy, give an example that is either **defined** or **used** in the `MutableShoppingList` code. Write the operation's type signature in function notation with its name, inputs, and outputs.

Creator: `TreeMap` : *void* $\rightarrow$ `TreeMap`

(e) Producer:

: $\rightarrow$

Solution. *e.g.* `+` : `String` $\times$ `String` $\rightarrow$ `String` ■

(f) Observer:

: $\rightarrow$

Solution. *e.g.* `get` : `Map<K,V>` $\times$ `K` $\rightarrow$ `V`, or

`howMany` : `MutableShoppingList` $\times$ `String` $\rightarrow$ **`int`** ■

(g) Mutator:

: $\rightarrow$

Solution. *e.g.* `append` : `StringBuilder` $\times$ `String` $\rightarrow$ `StringBuilder`, or

`another` : `MutableShoppingList` $\times$ `String` $\rightarrow$ *void* ■

You may detach this page. Write your username at the top, and hand in all pages when you leave.

Immutable `FixedShoppingList`

```
1  /** Immutable shopping list. */
2  public class FixedShoppingList {

3      List<String> replist;

4      /** Make a new shopping list. */
5      public FixedShoppingList(List<String> items) {
6          replist = new ArrayList<>(items);
       }

7      /** Make a new shopping list that includes itemToAdd. */
8      public FixedShoppingList withAnother(String itemToAdd) {
9          List<String> items = new ArrayList<>(replist);
10         items.add(itemToAdd);
11         return new FixedShoppingList(items);
```

```
    }

    /** Return the quantity of item on this list. */
    public int howMany(String item) {
        int count = 0;
        for (String listItem : replist) {
            if (item.equals(listItem)) { count++; }
        }
        return count;
    }

    /** Return a string where the items on this list appear in order,
     *  each on a separate line along with its quantity. */
    @Override public String toString() {
        Set<String> seen = new HashSet<String>();
        StringBuilder result = new StringBuilder();
        for (String item : replist) {
            if ( ! seen.contains(item)) {
                seen.add(item);
                result.append(item + " (" + howMany(item) + ")\n");
            }
        }
        return result.toString();
    }
}
```

You may detach this page. Write your username at the top, and hand in all pages when you leave.

Mutable `MutableShoppingList`

```
/** Mutable shopping list. */
public class MutableShoppingList {

    SortedMap<String, Integer> itemCounts;

    /** Make a new shopping list. */
    public MutableShoppingList() {
        itemCounts = new TreeMap<>();
    }

    /** Add 1 of the given item to this list. */
    public void another(String item) {
        itemCounts.put(item, itemCounts.getOrDefault(item, 0) + 1);
    }

    /** Return the quantity of item on this list. */
    public int howMany(String item) {
        return itemCounts.get(item); // BUG!
    }
```

```
13      /** Return a string where the items on this list appear in order,
14       *  each on a separate line along with its quantity. */
15      @Override public String toString() {
16          StringBuilder result = new StringBuilder();
17          for (String item : itemCounts.keySet()) {
18              result.append(item + " (" + howMany(item) + ")\n");
            }
19          return result.toString();
        }
    }
```