CS2030 Programming Methodology
Semester 1 2020/2021

14 October 2020
Problem Set #7 Functional Programming

1. Explain why the following code does not follow Functional Programming principles:

```
var still_alive = true;

while (still_alive) {
    wear_mask();
    wash_hands();
    keep_1m_apart();
    test_covid();
}
```

Solution: The return values (if any) of the 4 function calls are not used. So how does `still_alive` change its value? Answer: by a side effect in one of the 4 functions. This goes against the principle of FP.

2. We revisit Q4 of Recitation #5: study the classes `QA`, `MCQ` and `TFQ` in the Appendix. Since the checking of answers may throw exceptions, let's use the `Sandbox` functor to deal with them. In `main` method of `qaTest`, 4 questions have been created and added to a list. Complete the code to use the `Sandbox` functor to display the question, check the answer, and add the result to the `results` list.

```
public class qaTest {
    static void displayResults(List<Boolean> rList) {
        int marks = 0;
        for (boolean a : rList)
            if (a)
                marks++;
        System.out.println(String.format("You got %d questions correct.",
                                         marks));
    }

    public static void main(String[] args) {
        List<Boolean> results = new ArrayList<>();
        List<QA> questions = new ArrayList<>();

        questions.add(new MCQ("What is 1+1?", 'A'));
        questions.add(new TFQ("The sky is blue (T/F)", 'T'));
        questions.add(new MCQ("Which animal is an elephant?", 'C'));
        questions.add(new TFQ("A square is a circle (T/F)", 'F'));

        for (QA q : questions) {
            //Insert code here to use Sandbox to wrap each question,
            //show it, get user input, and add the result to
            //the results list above
        }
        displayResults(results);
    }
}
```

Solution: All the code is given in the zip file. We just highlight the necessary code here.

```
        questions.add(new MCQ("What is 1+1?", 'A'));
        questions.add(new TFQ("The sky is blue (T/F)", 'T'));
        questions.add(new MCQ("Which animal is an elephant?", 'C'));
        questions.add(new TFQ("A square is a circle (T/F)", 'F'));

        for (QA q : questions) {
            //Insert code here to use Sandbox to wrap each question,
            //show it, get user input, and add the result to
            //the results list above
            Sandbox.make(q)
                .map(QA::displayQuestion)
                .map(QA::getAnswer)
                .consume(results::add);
        }
```

3. Lists are also functors! Unlike `Optional` or `Sandbox`, they wrap many values, not just one. And although Java does not provide a `map` function, it is easy to do so.

```
<T,U> List<U> listMap(Function<T,U> f, List<T> list) {
    List<U> newList = new ArrayList<>();
    for (T item : list)
        newList.add(f.apply(item));
    return newList;
}
```

It is also useful to define a `filter` function. This takes in a predicate, and returns a new list whose elements (from the original list) make the predicate true. Note that both `map` and `filter` create *new* lists, and leave their arguments unchanged, in true FP style.

```
<T> List<T> listFilter(Predicate<T> p, List<T> list) {
    List<T> newList = new ArrayList<>();
    for (T item : list)
        if (p.test(item))
            newList.add(item);
    return newList;
}
```

With these, we may work on an entire collection of objects easily. For example, to convert all the words in a sentence to upper case, we may do the following:

```
List<String> intoWords(String sentence) {
    List<String> result = new ArrayList<>();

    //Add each word into the result list
    new Scanner(sentence).forEachRemaining(result::add);
    return result;
}

listMap(String::toUpperCase,
        intoWords("The rain in Spain falls mainly in the plain"));

=> [THE, RAIN, IN, SPAIN, FALLS, MAINLY, IN, THE, PLAIN]
```

To upper-case only those words which have an even length:

```
boolean isEven(int n) { return n % 2 == 0; }

listMap(String::toUpperCase,
        listFilter(s -> isEven(s.length()),
                   intoWords("The rain in Spain falls mainly in the plain")));

=> [RAIN, IN, MAINLY, IN]
```

(a) Define a function, `stringReverse`, that takes in a string `s` and returns a new string that is `s` reversed; eg. `stringReverse("hello")` returns `"olleh"`.

Solution: The easiest way is to convert the string to a `StringBuffer`, which has a `reverse` method, and then convert it back:

```
String stringReverse(String s) {
    return new String(new StringBuffer(s).reverse());
}

// Alternatively:

String stringReverse(String s) {
    char[] cArr = new char[s.length()];
    for (int i = 0; i < s.length(); i++) {
        int j = s.length() - 1 - i;
        cArr[i] = s.charAt(j);
        }
    return new String(cArr);
}
```

(b) Use this to reverse all the words in a sentence that contain the letter `i`. Try any sentence you like.

Solution:
```
listMap(word -> stringReverse(word),
    listFilter(word -> word.contains("i"),
        intoWords("The rain in Spain falls mainly in the plain")));

=> [niar, ni, niapS, ylniam, ni, nialp]
```

(c) Define a function, `stringToList`, that takes in a string `s` and returns a new `ArrayList` whose elements are strings containing each character of `s`. Example: `stringToList("hello")` returns `[h, e, l, l, o]`.

Solution:
```
List<String> stringToList(String s) {
    List<String> result = new ArrayList<>();
    for (char c : s.toCharArray()) {
        result.add(new String(new char[]{c}));
    }
    return result;
}
```

(d) What happens when you do the following?

```
listMap(s -> stringToList(s),
        intoWords("CS2030 is great fun"));
```

Solution:
```
listMap(s -> stringToList(s),
    intoWords("CS2030 is great fun"));

=> [[C, S, 2, 0, 3, 0], [i, s], [g, r, e, a, t], [f, u, n]]
```

(e) To avoid nested lists, write a `listFlatmap` function that, like `listMap`, takes in a function `f` and a list, and returns a new list whose elements are mapped by `f`, but flattens any nested list. Thus

```
listFlatMap(s -> stringToList(s),
        intoWords("CS2030 is great fun"));

=> [C, S, 2, 0, 3, 0, i, s, g, r, e, a, t, f, u, n]
```

With `listFlatmap`, lists are also monads!

Solution:

```
<T,U> List<U> listFlatmap(Function<T, List<U>> f, List<T> list) {
    List<U> newList = new ArrayList<>();

    for (T items : list)
        newList.addAll(f.apply(items));
    return newList;
}
```