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();
}
```

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);
    }
}
```

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"`.

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

(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]`.

(d) What happens when you do the following?

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

(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!

```
//*************** QA ****************
import java.util.*;

abstract class QA {

    String question;
    char correctAnswer;

    public QA(String question, char ans) {
        this.question = question;
        this.correctAnswer = ans;
    }

    abstract boolean getAnswer();

    public QA displayQuestion() {
        System.out.println(this.question);
        return this;
    }

    public char getInput() {
        Scanner sc = new Scanner(System.in);
        System.out.printf("=> ");
        char result = sc.next().charAt(0);
        System.out.println("");
        return result;
    }
}

//*************** MCQ ****************
import java.util.*;

class MCQ extends QA {
    public MCQ(String question, char ans) {
        super(question, ans);
    }

    @Override
    boolean getAnswer() {
        char answer = this.getInput();
        if (answer < 'A' || answer > 'E') {
            throw new InvalidMCQException("Invalid MCQ answer");
        }
        return this.correctAnswer == answer;
    }
}
//*************** TFQ ****************
import java.util.*;

class TFQ extends QA {
    public TFQ(String question, char ans) {
        super(question, ans);
    }

    @Override
    boolean getAnswer() {
        char answer = this.getInput();
        if (answer != 'T' && answer != 'F') {
            throw new InvalidTFQException("Invalid TFQ answer");
        }
        return this.correctAnswer == answer;
    }
}
//*************** Sandbox.java **********
/**
   Functor that separates exception handling from main processing.
 */
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.Objects;
```

```
/** Class Sandbox is a Functor, and is an immutable class.
    It's purpose is to separate exception handling from the main
    sequence of mapping.
 */
public final class Sandbox<T> {
    private final T thing;
    private final Exception exception;

    private Sandbox(T thing, Exception ex) {
        this.thing = thing;
        this.exception = ex;
    }

    public static <T> Sandbox<T> make(T thing) {
        return new Sandbox<T>(Objects.requireNonNull(thing), null);
    }

    public static <T> Sandbox<T> makeEmpty() {
        return new Sandbox<T>(null, null);
    }

    public boolean isEmpty() {
        return this.thing==null;
    }

    public boolean hasNoException() {
        return this.exception == null;
    }

    public <U> Sandbox<U> map(Function<T,U> f) {
        return map(f, "");
    }

    public <U> Sandbox<U> map(Function<T,U> f, String errorMessage) {
        if (this.isEmpty())
            return new Sandbox<U>(null, this.exception);

        try {
            return new Sandbox<U>(f.apply(this.thing), null);
        }
        catch (Exception ex) {
            return new Sandbox<U>(null,
                                  new Exception(errorMessage, ex));
        }
    }

    /** consume calls the Consumer argument with the thing.
        Exceptions, if any, are handled here.
    */
    public void consume(Consumer<T> eat) {
        if (this.isEmpty() && !this.hasNoException())
            handleException();

        if (!this.isEmpty() && this.hasNoException())
            eat.accept(this.thing);
    }

    private void handleException() {
        System.err.printf(this.exception.getMessage());
        Throwable t = this.exception.getCause();
        if (t != null) {
            String msg = t.getMessage();
            if (msg != null)
                System.err.printf(": %s", msg);
        }
        System.err.println("");
    }
}
```