Solutions to Quiz 2 (May 8, 2015)

Problem 1 (Multiple choice) (20 points).
Circle all correct answers for the following questions.

(a) Which of the following are true?

A. Using only thread-safe data types in your rep ensures that an ADT will be thread-safe.
B. Using only immutable data types in your rep ensures that an ADT will be thread-safe.
C. Java Swing objects cannot safely be observed from multiple threads.
D. Java Swing objects cannot safely be mutated from multiple threads.
E. You can ensure that your code is thread-safe with a thorough test suite.

Solution. C, D.
Just using thread-safe types or immutable types is not enough; for example, you might have mutable references. We cannot rely on testing to uncover concurrency bugs.

(b) Given the following grammar, which is not written in ANTLR syntax:

\[ Q ::= (T? R | S) \]
\[ R ::= [aeiou]+ \]
\[ S ::= ('x' | 'y' | 'z')* T \]
\[ T ::= [0-9] \]

Select all strings below where the entire string can be parsed by this grammar if Q is the root nonterminal:

A. 7eieio
B. y0y0
C. zzz
D. eieio
E. 10iou

Solution. A, D.
B is wrong because y cannot appear after a number. C is wrong because a number must appear after x, y, or z. E is wrong because the number in front of iou should only be a single digit.

Circle one best answer for the following questions.
We would like to implement some Tweet-processing functions, similar to those in Problem Set 1, using map/filter/reduce.

(c) Select an option to implement this method:

\[ \text{public static Stream<Tweet> writtenBy(Stream<Tweet> tweets, String username)} \]

A. return tweets.map(p -> p.getAuthor().equals(username));
B. return tweets.map(q -> q.getAuthor()).filter(r -> r.equals(username));
C. return tweets.filter(s -> s.getAuthor().equals(username));
D. return tweets.filter(t -> t.equals(username)).map(u -> u.getAuthor());
E. return tweets.filter(v -> v.equals(username));

Solution. C.

(d) Select an option to implement this method:

\[
\text{public static Stream}\langle\text{String}\rangle \ \text{getAuthors(Stream}\langle\text{Tweet}\rangle \ \text{tweets)}
\]

A. return tweets.map(a -> a.getAuthor());
B. return tweets.map(b -> b.getAuthor()).reduce((c, d) -> c + d);
C. return tweets.reduce((e, f) -> e.getAuthor() + f.getAuthor());
D. return tweets.reduce("", (g, h) -> g + h.getAuthor(), (i, j) -> i + j);
E. return tweets.filter(k -> k.getAuthor());

Solution. A.

Problem 2 (Defining data types) (20 points).
Here's the definition of the type Optional\langle T\rangle from the Java API:

A container object which may or may not contain a non-null value.
If a value is present, isPresent() will return true and get() will return the value.

Optional provides operations including:

\[
\begin{align*}
&\text{// return an empty Optional instance} \\
&\text{empty: void --> Optional}\langle T\rangle \\

&\text{// return an Optional with the specified present non-null value} \\
&\text{of: T --> Optional}\langle T\rangle \\

&\text{// return true if there is a value present, otherwise false} \\
&\text{isPresent: Optional}\langle T\rangle \ --> \ boolean \\

&\text{// if a value is present in this Optional, return the value,} \\
&\text{otherwise throw NoSuchElementException} \\
&\text{get: Optional}\langle T\rangle \ --> \ T \\
\end{align*}
\]

(a) Write a datatype definition for an implementation of Optional with two concrete variants:

\[
= () + ()
\]

Solution. Optional\langle T\rangle = EmptyOptional\langle T\rangle() + PresentOptional\langle T\rangle(value:T)

(b) Define the isPresent operation in function notation:
isPresent: $\text{Optional}\langle T \rangle \rightarrow \text{boolean}$

isPresent() =

isPresent() =

Solution.

isPresent(EmptyOptional) = false
isPresent(PresentOptional(value)) = true

(c) Write a datatype definition for immutable lists ImList\<$E>$ containing element type $E$, implemented with two concrete variants:

= () + ()

Solution. ImList\<$E>$ = Empty\<$E>() + Cons\$(T)(first:E, rest:ImList\<$E>$)

(d) We’d like to define an operation first on ImList that returns an Optional containing the first element of the list, or an empty Optional if the list is empty.

Define the first operation in function notation. You may reference the operations of Optional defined above by writing e.g. Optional.empty(), Optional.of(y), etc.

first: ImList\<$T>$ \rightarrow Optional\<$T$

first() =

first() =

Solution.

first(Empty) = Optional.empty()
first(Cons(first, rest)) = Optional.of(first)

Problem 3 (ADTs) (30 points).
Consider the following ADT:

1) /** Represents a mutable ordered sequence of distinct objects of another type $T$
2) * (where the ordering is determined by $T$’s compareTo method).

*/
3) **public interface** OrderedList\<$T$ extends Comparable\<$T$>> {

4) /** @return the largest element in this list.
5) * Requires this list to be nonempty.

*/

6) $T$ max();
7)    /** @param value must be greater than this.max(), if this list is nonempty
8)    * Adds value to the end of this list.
  */
9)    void add(T value);

a)    /** @param value
b)    * Removes value from this list.
c)    * @throw IllegalArgumentException if value not found in this list
  */
d)    void remove(T value);

(a) Write clearly the statement numbers and/or letters of every precondition in the code above. If there are
no preconditions, write “none.”

Solution. 5, 6, 7, 9, a, d.

(b) Write clearly the statement numbers and/or letters of every postcondition in the code above. If there
are no postconditions, write “none.”

Solution. 4, 6, 8, 9, b, c, d.

(c) Write clearly the statement numbers and/or letters of every rep invariant in the code above. If there are
no rep invariants, write “none.”

Solution. None.

(d) Which kinds of operations are found in the ADT above? Circle all that apply.

A. creator
B. mutator
C. observer
D. producer

Solution. B, C.

(e) Fill in the blanks using the pieces provided to produce a correct and consistent implementation of the
ADT.

    public class CS implements OrderedList<Character> {

        private String rep;
        // Rep invariant:
        //   (circle one) A, B, C, D, E
        // Abstraction function:
        //   (circle one) A, B, C, D, E

        public CS() {
            rep = "";
        }
    }
public Character max() {
    (circle one) F, G, H, I, J
}

public void add(Character value) {
    (circle one) F, G, H, I, J
}

public void remove(Character value) {
    int index = rep.indexOf(value);
    if (index < 0) {
        throw new IllegalArgumentException();
    }
    rep = rep.substring(0, index) + rep.substring(index+1);
}

A. rep.length() > 0
B. rep.charAt(i) < rep.charAt(j) for all 0 <= i < j < rep.length()
C. rep.charAt(i) > rep.charAt(j) for all 0 <= i < j < rep.length()
D. represents the sequence of characters rep.charAt(n-1),...,rep.charAt(0) where n=rep.length()
E. represents the sequence of characters rep.charAt(0),...,rep.charAt(n-1) where n=rep.length()
F. return rep.charAt(0);
G. rep = value + rep;
H. rep = rep + value;
I. rep = rep - value;
J. rep.concat(value);

Solution. There is only one available implementation for max, which forces us to choose a compatible rep. Remember that the AF describes abstract sequences and the RI is on concrete sequences, which here are stored end-first.
C, D, F, G.

Problem 4 (Concurrency) (30 points).
Consider the following code:

public class FishFish {
    private String oneFish = "two fish";

    public String getFish() {
        return oneFish;
    }
}
public synchronized boolean setFish(String newFish) {
    boolean same = oneFish.equals(newFish);
    oneFish = newFish;
    return ! same;
}

public synchronized String whatFish() {
    if (oneFish.contains(" ")) {
        return oneFish.substring(0, oneFish.indexOf(" "));
    }
    return oneFish;
}
}

(a) Suppose we call methods of the same FishFish instance on two different threads.
Circle all cases where the execution of code in the two methods might be interleaved:

A. One thread calls getFish, another thread calls getFish
B. One thread calls getFish, another thread calls setFish
C. One thread calls getFish, another thread calls whatFish
D. One thread calls setFish, another thread calls setFish
E. One thread calls setFish, another thread calls whatFish
F. One thread calls whatFish, another thread calls whatFish

Solution. A, B, C. Calling setFish or whatFish acquires the lock on this. ■

(b) In the above code, suppose we take each synchronized method, remove the synchronized keyword, and instead wrap the body of that method in a synchronized(oneFish) { ... } block. Again we call methods of the same FishFish instance on two different threads. Circle all cases where the execution of code in the two methods might be interleaved:

A. One thread calls getFish, another thread calls getFish
B. One thread calls getFish, another thread calls setFish
C. One thread calls getFish, another thread calls whatFish
D. One thread calls setFish, another thread calls setFish
E. One thread calls setFish, another thread calls whatFish
F. One thread calls whatFish, another thread calls whatFish

Solution. A, B, C, D, E. Calling setFish or whatFish now acquires the lock on oneFish, which setFish reassigns during its execution. ■

Again in the same example, suppose we remove all use of synchronized.

We start two threads running concurrently with access to the same newly-created FishFish instance f:
// in thread 1
f.setFish("red");
f.setFish("blue fish");

// in thread 2
System.out.println(f.whatFish());

(e) Could we encounter a runtime error in setFish? **Circle one:** yes  no
If yes, **in one sentence**, say why:

**Solution.** No.

(d) Could we encounter a runtime error in whatFish? **Circle one:** yes  no
If yes, **in one sentence**, say why:

**Solution.** Yes: if oneFish is changed to remove spaces between execution of the first and second lines, indexOf will return -1, and substring will throw 'IndexOutOfBoundsException'.

(e) In the boxes below, write clearly all the possible outputs when the code runs to completion. There are more boxes than possible outputs.

**Solution.** We might run whatFish before, between, or after the setFishes:

two
red
blue

But we also might interleave their executions:

blu
blue fish