MIT
6.005: Software Construction
Prof. Rob Miller & Max Goldman
revised Monday 23rd November, 2015, 21:54

# Solutions to Quiz 2 (November 20, 2015)

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

**(a)** Which of the following are true of Java concurrency?

A. A `synchronized` method declaration ensures that that method cannot interleave with other methods.
B. Two nested `synchronized` blocks within a method will always deadlock.
C. Blocking read operations are a reason to use multiple threads in a network server.
D. Lock ordering is a strategy for preventing deadlock.
E. The fields of an object are thread-confined to that object.

**Solution.** C, D.

A is false because `synchronized` only guarantees that other methods that synchronize on the same lock cannot interleave. B is false because Java locks are reentrant. E is false, it doesn't make sense. ■

**(b)** Which of the following are true of grammars and regular expressions?

A. For every regular expression, it is possible to write a grammar that matches the same strings.
B. If a grammar is not recursive, it will not match any strings.
C. The regular expression `(a+)b` matches the two-letter string `"ab"`.
D. The regular expression `(a+)+` matches the single-letter string `"a"`.
E. The regular expression `(a+)*` matches the empty string `""`.

**Solution.** A, C, D, E. ■

**(c)** Suppose we have a Python program where `restaurants` is a list of objects that have a `stars` field, and suppose we define:

```
f1 = lambda c,x: c+1
f2 = lambda s: s == 4
f3 = lambda r: r.stars
```

Which lines of Python code will compute the number of restaurants with 4 stars?

```
A. filter(f1, reduce(f2,    map(f3, restaurants)), 0)
B. reduce(f1, filter(f2,    map(f3, restaurants)), 0)
C. filter(f1,    map(f2, reduce(f3, restaurants)), 0)
D. reduce(f1,    map(f2,    map(f3, restaurants)), 0)
E.    map(f1, filter(f2, reduce(f3, restaurants)), 0)
```

**Solution.** B.

D almost works, but will count all restaurants, not only 4-star restaurants. ■

**(d)** Consider the following two specifications for a method `trim`:

Spec 1:

```
/**
 * Removes all spaces at the beginning and end of the input string.
 * @param toTrim string may contain upper- and lower-case letters and spaces
 * @return input string with leading and trailing spaces removed
 */
public String trim(String toTrim);
```

Spec 2:

```
/**
 * Removes all spaces and newlines at the beginning and end of the input string.
 * @param toTrim string may contain upper- and lower-case letters, spaces, and newlines
 * @return input string with leading and trailing spaces and newlines removed
 */
public String trim(String toTrim);
```

Which of the following are true of these specifications?

A. Spec 1 has a stronger precondition than spec 2
B. Spec 1 has a stronger postcondition than spec 2
C. Spec 1 is stronger than spec 2
D. Spec 1 is weaker than spec 2
E. Spec 1 is incomparable to spec 2

**Solution.** A, D.

Every input that satisfies 1's precondition also satisfies 2's precondition but not the reverse, so spec 1 has a stronger precondition.

Every implementation that satisfies 2 also satisfies 1: spec 1 has a stronger precondition, and for any input that satisfies 1's precondition, 2's postcondition is the same as 1's postcondition. ■

**Problem 2** (ADTs and Equality) (**20 points**).
Ben Bitdiddle is a talented developer and is being hired as a contractor at a number of firms. He needed a way to keep track of when he was free and to make sure that he only committed to one job at any time. In addition, he decided there were certain times he was not willing to work. To help keep track of his time, he designed the ADT below.

```
/**
 * Calendar is a mutable ADT representing busy time and free time.
 * Some free time is dedicated free time that cannot be busy.
 */
public class Calendar {
    private final List<Interval> dedicatedFreeTime = new ArrayList<>();
    private final List<Interval> jobTimes = new ArrayList<>();

    /**
     * Create a new calendar with no busy time and the given dedicated free time.
     * @param freeTime dedicated free time intervals that cannot be busy
     */
    public Calendar(List<Interval> freeTime) {
        for (Interval i : freeTime) {
            dedicatedFreeTime.add(i);
        }
    }

    // ... observer and mutator methods ...
}

/**
 * Interval is an immutable ADT representing the half-closed interval [start,end)
 * that starts at time start and ends just before time end.
 */
public class Interval {

    // java.time.Instant is immutable, represents an instant in time
    private final Instant start;
    private final Instant end;

    /**
     * Create a new time interval [start, end).
     */
    public Interval(Instant start, Instant end) {
        this.start = start;
        this.end = end;
    }

    // ... observer and producer methods ...
}
```

**(a)** For each of the statements below, say whether it should be included in the internal documentation of `Calendar` by writing:

**AF** if the statement belongs in the abstraction function

**RI** ... the rep invariant

**EXP** ... the argument that type has no rep exposure

**NONE** if it should not be included in any of those

You should include in the AF, RI, or EXP **all good statements** that are compatible with the code and specs on the previous page.

Do not include statements that are not compatible with the code and specs.

```
intervals in jobTimes do not overlap
```

```
fields are private and final
Interval objects are immutable
time intervals that do not overlap any interval in jobTimes are free time
time intervals that do not overlap any interval in dedicatedFreeTime are busy
intervals in dedicatedFreeTime do not overlap
no interval in jobTimes overlaps with any interval in dedicatedFreeTime
initial dedicated free time intervals are copied to a new list
```

**Solution.** RI, EXP, EXP, AF, NONE, NONE, RI, EXP.

Time intervals that don't overlap `dedicatedFreeTime` might be busy, or they might be non-dedicated free time.

Since the constructor assigns `dedicatedFreeTime` from a client input without the non-overlapping precondition, we cannot require it as a rep invariant. ■

**(b)** The `equals` method of `Interval` should: (**circle one best answer**)

A. Use the reference equality implementation provided by Java
B. Return true when called with an `Interval` argument
C. Return true when `start` and `end` are equal according to reference equality
D. Return true when `start` and `end` are equal according to `equals`
E. Return true when the amount of time between `start` and `end` is the same

**Solution.** D. ■

**Problem 3** (Recursive Data Types) (**26 points**).
We want to represent a simplified version of HTML (the language of web pages) with normal text, **bold text**, and *italic text*, where there is arbitrary nesting. Here is an example piece of HTML:

```
Some text <b>that is bold <i>and italic <b>(still bold)</b></i></b>
```

which would be rendered in a web browser as:

Some text **that is bold *and italic (still bold)***

Here is a partial grammar for simplified HTML:

```
html ::=  ( normal | bold | italic ) *
normal ::= text
text ::= [^<>]*
```

**(a)** Which of the following options for completing the grammar would accept the example piece of HTML above? **Circle all correct answers.**

A.
```
bold ::= '<b>' normal italic '</b>'
italic ::= '<i>' normal bold '</i>'
```

B.
```
bold ::= '<b>' html '</b>'
italic ::= '<i>' html '</i>'
```

C. 
```
bold ::= '<b>' (normal | bold | italic) '</b>'
italic ::= '<i>' (normal | bold | italic) '</i>'
```

**Solution.** B.

A is broken, nonterminals `bold` and `italic` will infinitely recursive, they have no base case.

C cannot represent bold or italic tags whose contents are not all normal text or a single bold or italic tag. ■

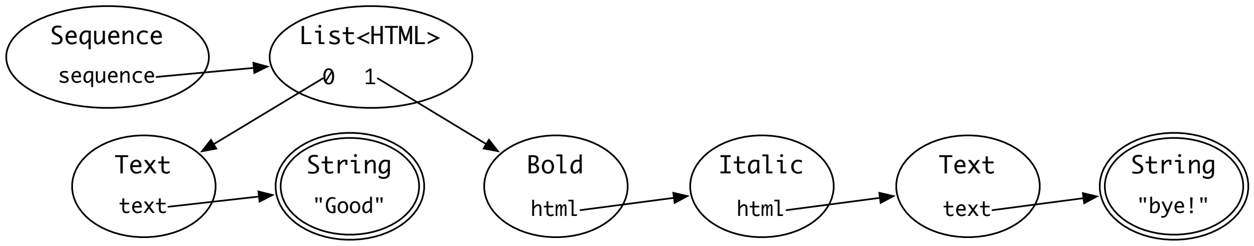
Below is a recursive datatype definition for a `HTML` ADT:

```
HTML = Sequence(sequence: List<HTML>) + Text(text: String)
       + Bold(html: HTML) + Italic(html: HTML)
```

**(b)** Given that recursive type, draw a snapshot diagram that correctly represents the abstract syntax tree for this example piece of HTML:

```
Good<b><i>bye!</i></b>
```

**Solution.** ■



Below is an interface for the `HTML` type:

```
public interface HTML {
    /** @return the number of characters in this HTML */
    public int length();

    /** @return the number of bolded characters */
    public int bolded();

    /** @return the number of italicized characters */
    public int italicized();

    /** @return the number of characters that are both bolded and italicized */
    public int overemphasized();
}
```

For the concrete variants below, implement `bolded()`, `italicized()`, and `overemphasized()`.

**Implement each method with a single line of code.**

**(c)**
```
public class Text implements HTML {
    private final String text;
    // ...
    @Override public int length() { return text.length(); }
    @Override public int bolded() {                                          }

    @Override public int italicized() {                                      }

    @Override public int overemphasized() {                                  }
}
```

(d)
```
public class Bold implements HTML {
    private final HTML html;
    // ...
    @Override public int length() { return html.length(); }

    @Override public int bolded() {                                          }

    @Override public int italicized() {                                      }

    @Override public int overemphasized() {                                  }
}
```

(e)
```
public class Italic implements HTML {
    private final HTML html;
    // ...
    @Override public int length() { return html.length(); }

    @Override public int bolded() {                                          }

    @Override public int italicized() {                                      }

    @Override public int overemphasized() {                                  }
}
```

**Solution.**

```
public class Text implements HTML {
    private final String text;
    // ...
    @Override public int length() { return text.length(); }
    @Override public int bolded() { return 0; }
    @Override public int italicized() { return 0; }
    @Override public int overemphasized() { return 0; }
}

public class Bold implements HTML {
    private final HTML html;
    // ...
    @Override public int length() { return html.length(); }
    @Override public int bolded() { return length(); }
    @Override public int italicized() { return html.italicized(); }
    @Override public int overemphasized() { return italicized(); }
}
```

```
public class Italic implements HTML {
    private final HTML html;
    // ...
    @Override public int length() { return html.length(); }
    @Override public int bolded() { return html.bolded(); }
    @Override public int italicized() { return length(); }
    @Override public int overemphasized() { return bolded(); }
}
```

■

**Problem 4** (Thread Safety) (**18 points**).
Louis Reasoner is trying to create a study group system for his friends.

```
/** Immutable student. */
public interface Student {
    // ...
}

/** Threadsafe mutable study group. */
public class StudyGroup {
    // Thread safety: implements the monitor pattern

    private final Set<Student> members;

    /** Create a new empty study group. */
    public StudyGroup() {
        this.members = new HashSet<>();
    }

    /** Add student s to this group. */
    public synchronized void join(Student s) {
        members.add(s);
    }

    /** Remove student s from this group. */
    public synchronized void leave(Student s) {
        members.remove(s);
    }

    /** Move student s from this group to destination. */
    public synchronized void migrate(StudyGroup destination, Student s) {
        if (members.contains(s)) {
            this.leave(s);
            destination.join(s);
        }
    }
}
```

Louis has asked you to help him debug in his code.

**(a)** Given two threads that can access the objects defined below, write minimal code that will run in each thread and could trigger deadlock.

```
final Student alyssa = ... // a student
final Student ben = ...    // another student
final Student louis = ...  // another student

final StudyGroup east = new StudyGroup();
final StudyGroup west = new StudyGroup();
final StudyGroup dome = new StudyGroup();

Thread thread1 = new Thread(new Runnable() {
    public void run() {


    }
});

Thread thread2 = new Thread(new Runnable() {
    public void run() {


    }
});

thread1.start();
thread2.start();
```

**(b)** Describe briefly and precisely when deadlock would occur:

**Solution.**

```
// thread 1
east.join(ben);
east.migrate(west, ben);

// thread 2
west.join(alyssa);
west.migrate(east, alyssa);
```

Both threads enter `migrate` and obtain the lock on `this`. They deadlock trying to call `destination.join()`.

■

**Problem 5** (Message Passing) (**16 points**).
Joe has created a collaborative to-do list application with a central server managing the data. Clients update the to-do list by sending messages to the server over a standard network socket:

- *add* <*item*>: adds an item to the end of the list, where *item* is a string
- *remove* <*n*>: complete and remove the $n^{th}$ item from the list, where $n$ is an integer indexed from $0$

Here is how the protocol is defined:

- When receiving *add*, the server will add the item to the end of the list
- When receiving *remove*, if the index is valid, the server will remove the item at that index from the list

Suppose two clients are connected to the server, and the list currently contains one item: "buy milk".

Client 1 *add*s "walk dog" and "wash car" to the list and then *remove*s "walk dog" by sending the following messages in order:

(A) *add* walk dog

(B) *add* wash car

(C) *remove* 1

Concurrently, client 2 *remove*s "buy milk" by sending the following message:

(D) *remove* 0

**(a)** Write a sequence in which the server might receive messages A, B, C, and D that will yield the correct result:

**(b)** Write a second, different sequence in which the server might receive the messages that will yield the correct result, or write NONE if there is none:

**Solution.** A, B, C, D. ■

**(c)** Write a sequence in which the server might receive the messages that will yield an incorrect result:

**(d)** Write a second, different sequence in which the server might receive the messages that will yield an incorrect result, or write NONE if there is none:

**Solution.** D, A, B, C -or- A, D, B, C -or- A, B, D, C. ■