Solutions to Quiz 2 (April 22, 2016)

Problem 1 (Multiple Choice) (**20 points**).
For each question, *choose all correct answers*.

(a) Which of the following are true of thread confinement?

A. all `private final` fields are confined
B. all immutable values are confined
C. the values of all local variables are confined
D. a `BlockingQueue` used to implement message passing should be confined
E. the view tree in a Swing GUI should be confined

Solution. E. ■

(b) Select all the strings below where the *entire string* is matched by the regular expression:
`([a-z]+[1-9]+)*[1-9]`

A. `bb8b8b`
B. `r2d222`
C. `c3po000`
D. `8`
E. `r5d4`

Solution. B, D. ■

(c) Consider an immutable class `ComplexNum` for representing complex numbers:

```java
private final double realPart;
private final double imaginaryPart;

// AF: represents (realPart + i * imaginaryPart)
// RI: true

// ... other creators, producers, observers ...

public double absoluteValue() {
    return Math.sqrt(realPart*realPart + imaginaryPart*imaginaryPart)
}

@Override public boolean equals(Object other) {
    if (!(other instanceof ComplexNum)) { return false; }
    ComplexNum that = (ComplexNum)other;
    return this.absoluteValue() == that.absoluteValue();
}
```

This implementation of `equals`...

A. is reflexive
B. is symmetric
C. is transitive
D. returns true for `ComplexNum` pairs that should not be equal according to the AF
E. returns false for `ComplexNum` pairs that should be equal according to the AF

Solution. A, B, C, D. ■

(d) Since `ServerSocket.accept()` is a blocking method, we can conclude that...

A. `accept()` will not return until a client makes a connection
B. `accept()` will throw an exception if called when no clients are connecting
C. calling `accept()` could prevent the program from terminating
D. it is safe to call `accept()` simultaneously from multiple threads
E. while `accept()` is blocking, calling other methods on the same `ServerSocket` instance from other threads will block

Solution. A, C. ■

Problem 2 (Map/Filter/Reduce) (**18 points**).
Ben Bitdiddle is trying to select a new password for his social media accounts. Given the following interface for representing passwords:

```
public interface Password {
    public String plainText();
    public int strength();
}
```

For each method below, fill in the blanks to implement the specification using map, filter, and reduce. You must use exactly two operations, but you may use the same operation twice. On each line, fill in the first blank with `map`, `filter`, or `reduce`, and fill in the second blank with a lambda expression. Solutions will be graded for correctness and clarity.

(a)
```
/**
 * @param passwords the passwords to consider
 * @param minStrength the minimum strength a password should have
 * @return the plaintext strings of passwords that have strength at least minStrength
 */
public static List<String> strongEnough(List<Password> passwords, int minStrength) {
    return passwords.stream()

                    .___________(_________________________________)

                    .___________(_________________________________)
                    .collect(toList());
}
```

(b)
```
/**
 * @param passwords the passwords to consider
 * @param substring required substring
 * @return the plaintext strings of passwords that contain substring
 */
public static List<String> containing(List<Password> passwords, String substring) {
    return passwords.stream()

                    .___________(_________________________________)

                    .___________(_________________________________)
                    .collect(toList());
}
```
(c)
```
/**
 * @param passwords the passwords to consider
 * @return the average strength of the passwords
 */
public static double averageStrength(List<Password> passwords) {
    return passwords.stream()

                    .___________(_________________________________)

                    .___________(0, _________________________________)
                    / (double)passwords.size();
}
```

Solution.

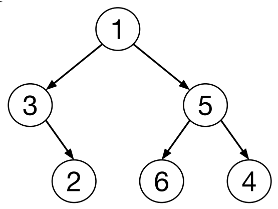
(a)
```
.filter(p -> p.strength() >= minStrength)
.map(p -> p.plainText())
```

(b)
```
.map(p -> p.plainText())
.filter(p -> p.contains(substring))
```

(c)
```
.map(p -> p.length())
.reduce(0, (a,b) -> a+b)
```

■

Problem 3 (Abstract Data Types) (**25 points**).
A tree is a data structure that consists of a root node and zero or more children which themselves are also trees. A binary tree is a tree in which each node has at most two children, designated *left* and *right*. For example:



Simon Straightforward is an enthusiastic 6.005 student who just learned about ADTs and wants to build a class to represent binary trees of positive integers. He comes up with this internal representation:

```
/** A binary tree of positive integers. */
public class PosIntBinaryTree {
    private final List<List<Integer>> nodes;
}
```

Simon wants each sub-list in `nodes` to represent a level in the binary tree, using `0` in place of missing elements. For example, the binary tree shown above is represented by the following list of lists:

```
[ [ 1 ]
  [ 3, 5 ]
  [ 0, 2, 6, 4 ] ]
```

(a) Silly Simon forgot to document his abstraction function and rep invariant!

Select statements to put into the AF and RI of `PosIntBinaryTree` by circling the letters of all statements to include from the list below. *Include all good statements* that are compatible with Simon's design.

```
AF:  A  B  C  D  E  F  G
RI:  A  B  C  D  E  F  G
```

A. `for all 0 <= i < nodes.size(), nodes.get(i).get(0) > 0`

B. `if nodes.size() == 0, represents the empty tree`

C. `if nodes.size() > 0, nodes.get(0).get(0) is the root node`

D. `nodes.get(i).size() == 2^i`

E. `nodes.size() == 2^k for some nonnegative integer k`

F. `for node nodes.get(i).get(j),`
`its left child (if any) is nodes.get(i+1).get(j*2)`

G. `for node nodes.get(i).get(j),`
`its right child (if any) is nodes.get(i+1).get(j*2+1)`

Solution. AF: B, C, F, and G. RI: D. ■

Rob Recursive, an even more enthusiastic student, wants to represent binary trees of *any* integers.

(b) What bad practice in Simon's rep prevents Rob from easily extending it?

Solution. Magic number 0. ■

(c) Rob decides to implement the binary trees recursively, using an interface `IntBinaryTree` with two concrete variants, one of which represents the empty tree. He also decides the type will be immutable.

```
/** An immutable binary tree of integers. */
public interface IntBinaryTree { ... }
```

Write a recursive datatype definition for `IntBinaryTree`:

Solution. `IntBinaryTree = Empty() + Tree(value: int, left,right: IntBinaryTree)` ■

(d) Start implementing the concrete variant that represents the empty tree by writing its field declarations, abstraction function, and rep invariant. If parts of the AF or RI would normally be assumed in 6.005, write them explicitly.

Fields:

AF:

RI:

Solution.

No fields

AF: represents the empty immutable binary tree of integers

RI: true ■

(e) Start implementing the other concrete variant by writing its field declarations, abstraction function, and rep invariant. If parts of the AF or RI would normally be assumed in 6.005, write them explicitly.

Fields:

AF:

RI:

Solution.

```
private final int value;
private final IntBinaryTree left, right;
```

AF: represents the immutable binary tree of integers with root node `value`, left subtree `left`, and right subtree `right`

RI: `left,right != null` ■

Problem 4 (Recursive Data Types) (**20 points**).
A Boolean formula is a propositional expression consisting of variables, $\wedge$ (and) and $\vee$ (or) binary operators, and $\neg$ (not) unary operators.

For example, the following expression means "either P or Q is true, and either P is false or R is true":

$$(P \vee Q) \wedge (\neg P \vee R)$$

A formula is in **negation normal form** if the negation operator ($\neg$) is *only* applied directly to variables. For example, $P \wedge (\neg Q \vee R)$ is in negation normal form, but $P \wedge \neg(Q \vee R)$ and $\neg(P \wedge (\neg Q \vee R))$ are not.

Here is a datatype definition for an ADT to represent negation normal form Boolean formulas:

```
NegNormFormula = Variable(name: String, isNegated: boolean) +
                 And(left: NegNormFormula, right: NegNormFormula) +
                 Or(left: NegNormFormula, right: NegNormFormula)
```

Now consider this specification:

```
// returns a negation of the input formula
negate : NegNormFormula -> NegNormFormula
```

(a) Classify this operation according to our types of ADT operations:

Solution. Producer. ∎

(b) Implement the operation. Use De Morgan's laws and the rule for double negation:

$$\neg(P \wedge Q) = \neg P \vee \neg Q \qquad \text{and} \qquad \neg(P \vee Q) = \neg P \wedge \neg Q \qquad \text{and} \qquad \neg\neg P = P$$

```
public class Variable implements NegNormFormula {
    private final String name;
    private final boolean isNegated;
    public Variable(String name, boolean isNegated) { ... }
    @Override public NegNormFormula negate() {
```

Solution.

```
        return new Variable(name, ! isNegated);
```

∎

```
    }
}
public class And implements NegNormFormula {
    private final NegNormFormula left;
    private final NegNormFormula right;
    public And(NegNormFormula left, NegNormFormula right) { ... }
    @Override public NegNormFormula negate() {
```

Solution.

```
        return new Or(left.negated(), right.negated());
```

∎

```
    }
}
public class Or implements NegNormFormula {
    private final NegNormFormula left;
    private final NegNormFormula right;
    public Or(NegNormFormula left, NegNormFormula right) { ... }
    @Override public NegNormFormula negate() {
```

Solution.

```
        return new And(left.negated(), right.negated());
```

∎

```
    }
}
```

Problem 5 (Thread Safety) (**17 points**).
It's election season! With the tight race for president of Fictional Dystopia, you've been asked to develop a secure online voting system.

In Fictional Dystopian elections, each voter has **exactly two** votes. Each voter can vote twice for the same candidate, or split their vote between two different candidates.

```
public class TwoVotesEachElection {

    private final AtomicInteger[] voteCounts;
    private final Set<String> voters;

    // ... constructor initializes voteCounts to an array of AtomicIntegers,
    //     each with value zero, and initializes voters to a threadsafe Set ...

    public int getNumberOfCandidates() { return voteCounts.length; }

    // requires: 0 <= can1, can2 < getNumberOfCandidates()
    // effects: if and only if the voter hasn’t already voted, casts both votes
    public void vote(final String voterID, final int can1, final int can2) {
        if (voters.contains(voterID)) { return; }
        voters.add(voterID);
        // use the atomic incrementAndGet operation
        // (we don’t need the "get" part, but there is no plain "increment")
        voteCounts[can1].incrementAndGet();
        voteCounts[can2].incrementAndGet();
    }

    // ...
}
```

Suppose `election` is a `TwoVotesEachElection` with 4 candidates (so `voteCounts.length = 4`).

(a) Unfortunately, even though `TwoVotesEachElection` uses threadsafe datatypes, the `vote()` operation is not threadsafe. Explain clearly an interleaving that violates thread safety for `TwoVotesEachElection`.

Solution. Two threads enter `vote` with the same `voterID` not yet in `voters`.

Both threads call `voters.contains` and obtain `false`, then both call `voters.add` successfully.

At this point, both threads will increment the vote counts for their `can1` and `can2`, giving this voter 4 votes instead of 2, in violation of the postcondition. ■

Louis Reasoner is convinced that using `AtomicInteger` is unnecessary, so he changes the code to use primitive integers instead:

```
public class LouisTwoVotesEachElection {

    private final int[] voteCounts;
    private final Set<String> voters;

    // ... constructor initializes voteCounts to an array of zeros,
    //     and initializes voters to a threadsafe Set ...

    public int getNumberOfCandidates() { return voteCounts.length; }

    // requires: 0 <= can1, can2 < getNumberOfCandidates()
    // effects: if and only if the voter hasn’t already voted, casts both votes
    public void vote(final String voterID, final int can1, final int can2) {
        if (voters.contains(voterID)) { return; }
        voters.add(voterID);
        voteCounts[can1]++;
        voteCounts[can2]++;
    }

    // ...
}
```

Of course, this just makes the problem worse.

(b) Suppose 100 voters, each using a different thread, participate in our 4-candidate election by concurrently calling Louis' `vote()`. Every voter uses a unique ID, and no one casts both of their votes for the same candidate. The value of the sum

`voteCounts[0] + voteCounts[1] + voteCounts[2] + voteCounts[3]`

could be which of the following? *Choose all that apply.*

A. 0
B. 1
C. 2
D. 4
E. 100
F. 200
G. 400

Solution. C, D, E, F.

At least one thread will successfully increment the count for its first candidate; same for some thread and its second candidate. With no races at all, 200 votes will be counted. ■

(c) Louis wants to fix the thread safety problems by putting all the code inside `vote()` in a `synchronized` block. Of the following objects in Louis' version of the code, which are suitable for us to synchronize on in order to ensure thread safety for concurrent calls to `vote()`? *Choose all that apply.*

A. `this`
B. `voterID`
C. `voters`
D. `voteCounts`
E. `voteCounts[0]`

Solution. A, C, D.

Locking on different `voterID` strings doesn't help. `voteCounts[0]` is a primitive in Louis' code. ■