Solutions to Quiz 1 (March 22, 2017)

Problem 1 (Specifications) (21 points).
Given the code, answer the questions below.

```java
/** Schedule represents a schedule of non-overlapping events */
public class Schedule {
    private final List<Event> events;
    // events is sorted in increasing order of start time
    // AND for all i<j, events[i].end <= events[j].start

    /**
     * Make a new Schedule
     * @param events a list of non-overlapping events sorted by increasing start time
     */
    public Schedule(List<Event> events) {
        this.events = events;
        checkRep();
    }

    private void checkRep() {
        ...
    }
}

/** Event represents an immutable event with a name, a start time, and an end time. */
interface Event {
    public String name();
    public Date start();
    public Date end();
}
```

(a) Write the line number(s) that state the preconditions of a creator operation for the Schedule ADT.

Solution. 9, 11. Line 11 specifies the types of the parameters, which are part of the precondition.

(b) Write the line number(s) that state the rep invariant of the Schedule ADT.

Solution. 4-5. (checkRep() on lines 16-18 could be considered a statement of the rep invariant as well.)

(c) Write the line number(s) that state the rep of the Schedule ADT.

Solution. 3.
Consider the following independent changes to this code. What effect do they have on the spec of the Schedule ADT?

(d) If “sorted by increasing start time” is removed from line 9, the spec for Schedule is:

A. stronger
B. weaker
C. same
D. incomparable

Solution.  A.

(e) If the call to checkRep() is removed from line 13, the spec for Schedule is:

A. stronger
B. weaker
C. same
D. incomparable

Solution.  C.

(f) If we add code to the Schedule constructor that satisfies the rep invariant regardless of whether the client satisfied the precondition, the spec for Schedule is:

A. stronger
B. weaker
C. same
D. incomparable

Solution.  C.

Problem 2 (Testing) (16 points).
Given this specification:

```plaintext
/**
 * Split a string on a delimiting character.
 *
 * @param text a string
 * @param delim a delimiter by which to split the string
 * @param limit an upper bound on the number of elements to return;
 * if limit < 0, there is no upper bound; limit != 0
 * @return a list of strings [s1, s2, ..., sN] such that:
 * - text = s1 + delim + s2 + delim + ... + delim + sN
 * - N <= limit if limit > 0
 * - none of s1, s2, ..., sN contain delim
 * @throws IllegalArgumentException if limit > 0 and
 * there are more than limit-1 occurrences of delim in text.
 */
public static List<String> split(String text, char delim, int limit);
```
(a) Start implementing a systematic testing strategy for this function by writing one good partitioning of the input space on input limit alone, i.e., the partition should not mention either text or delim.

Solution.
E.g., limit < 0, limit = 1, limit > 1

(b) Now, write one good partitioning of the input space on the relationship between limit and the occurrences of delim in text. Your partition should mention all three inputs.

Solution. Let N be the number of times delim appears in text. One reasonable partition is:

N < limit-1, N = limit-1, N >= limit

Another good one uses the boundary points at 0 and one as well:

N = 0, N = 1, 1 < N < limit-1, N = limit-1, N >= limit

Problem 3 (ADTs) (25 points).
Given this code:

1 /** An immutable class representing a dog show. */
2 public class DogShow {
3     private final Map<String, Integer> dogs;
4
5     public DogShow(List<String> dogsInShow) {... }
6     public DogShow copy() {... }
7     public List<String> getDogs() {... }
8 }
9
(a) Classify each operation according to its type, using the letters C, M, O, P.

DogShow() is a _______

copy() is a _______

getDogs() is a _______

Solution. DogShow is a creator (C), copy is a producer (P), and getDogs is an observer (O).

(b) Which of the following are possible abstraction functions for this ADT? (circle all that apply)

A. AF: dogs in a dog show are stored in dogs, with their weights as values
B. AF(dogs) = a dog show where dogs.get(breed) is the number of dogs in the show with the given breed
C. AF(dogs) = a map from a dog’s name to its weight in grams
D. AF(dogs) = a dog show where the nth dog to appear onstage is the dog whose name maps to n in dogs

Solution. B, D.
(e) Which of the following are possible rep invariants for this ADT? (circle all that apply)

A. dogs contains no negative integers as values
B. dogs.size() is 0, 1, >1
C. dogsInShow has no repeats
D. each dog in dogs represents a dog
E. dogs.size() is <= 50

Solution. A, E.

(d) If this ADT had good rep independence, which of the following would be true? (circle all that apply)

A. the implementer could change the precondition of the constructor without telling the client
B. the implementer could change the rep invariant without telling the client
C. the implementer could change the return type List<String> in the signature of getDogs() without telling the client
D. the implementer could make the abstraction function one-to-one without telling the client
E. the implementer could change dogs to a Map<String,String> without telling the client

Solution. B, D, E.
(e) For each implementation below, is the rep safe or exposed? Briefly explain.

```java
/** An immutable class representing a dog show. */
public class DogShow {
    private final List<String> dogs;

    public DogShow(List<String> dogsInShow) {
        dogs = Collections.unmodifiableList(dogsInShow);
    }

    public DogShow copy() {
        return new DogShow(dogs);
    }

    public List<String> getDogs() {
        return dogs;
    }
}
```

Safe? (Y/N)
Reason:

Solution. No, not safe. The constructor has rep exposure because the Collection.unmodifiableList() merely wraps the List object that was passed in, instead of making a copy of it. So the client has an alias to a mutable object in the rep.

copy() is safe and getDogs() is safe, because they are using unmodifiable lists.

(f) /* An immutable class representing a dog show. */

```java
/** An immutable class representing a dog show. */
public class DogShow {
    private final List<String> dogs;

    public DogShow(List<String> dogsInShow) {
        this.dogs = new ArrayList<>();
        dogs.addAll(dogsInShow);
    }

    public DogShow copy() {
        return new DogShow(dogs);
    }

    public List<String> getDogs() {
        return Collections.unmodifiableList(dogs);
    }
}
```

Safe? (Y/N)
Reason:

Solution. Yes, safe. The rep is private, so clients can’t reassign any of its fields. There is defensive copying in the constructor, which the producer also takes advantage of, and the observer returns an unmodifiable view of the rep.

Problem 4 (Code Review) (20 points).
Alyssa P. Hacker wrote the following method and asked Ben Bitdiddle to review her code:
public static int pathLength(final String start, final String target, final Map<String, String> edges) {
    String current = start;
    String next;
    int edgesTraversed = 0;
    while (edges.containsKey(current)) {
        next = edges.get(current); // traverse edge
        edgesTraversed++;
        if (next.equals(target)) return edgesTraversed;
        edges.remove(current); // avoid entering an infinite cycle
        current = next;
    }
    return -1;
}

Ben wrote a series of code review comments. For each comment below,

1. indicate whether you AGREE or DISAGREE with the comment;

2. provide one sentence explaining why.

(a) Ben says: “on line 18, removing entries from edges will be disallowed by Java at runtime because edges is declared as final.”

AGREE or DISAGREE?

Explanation:

Solution. DISAGREE. Though edges cannot be reassigned, its value can be mutated.
(b) Ben says: “on line 18, the implementation doesn’t satisfy the spec.”
AGREE or DISAGREE?
Explanation:
Solution.  AGREE. Mutation of parameters has to be explicitly allowed in the spec.
<!–if (next.equals(start)) return -1; // avoid entering an infinite cycle–> ■

(c) Ben says: “on line 17, next and target are Strings, so they should be compared using == instead of .equals().”
AGREE or DISAGREE?
Explanation:
Solution.  DISAGREE. Strings are not primitives, so you must use .equals() to check for equality. ■

(d) Ben says: “On line 12, the scope of the variable next should be minimized.”
AGREE or DISAGREE?
Explanation:
Solution.  AGREE. ‘next’ is only used within the body of the while loop, so it should be declared there. ■

(e) Ben says: “The implementation doesn’t fail fast on inputs that violate the precondition.”
AGREE or DISAGREE?
Explanation:
Solution.  AGREE. It could fail faster by asserting the precondition at the start of the method.

assert edges.containsKey(start);
assert edges.containsValue(target);

Problem 5 (Multiple Choice) (18 points). (a) Suppose you have the following class:

```java
public class Square {
    private int sideLength; // 1
    // Rep invariant: ... // 2
    // Abstraction function: ... // 3

    /**
     * @return area of this shape // 4
     */
    public int getArea() {
        ... int
    }
}
```
Now suppose that you’d like to separate your class out and have it implement an interface, like so:

```java
public interface Shape {
    ...
}

public class Square implements Shape {
    ...
}
```

For each of the following commented lines (1, 2, 3, 4) in the old class, denote whether you think it belongs in the new interface or the new implementation class:

(circle one best answer)

A. 1, 2, and 4 belong in the implementation; 3 belongs in the interface.
B. 1, 2, and 3 belong in the implementation; 4 belongs in the interface.
C. 1 and 2 belong in the implementation; 3 and 4 belong in both.
D. 1, 2, 3, and 4 belong in the implementation.
E. 1, 2, 3, and 4 belong in both.

**Solution.** B. Everything related to the rep should go into the implementation, while specs and method signatures belong in the interface.

(b) Given the following snippet of code:

```java
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.length; i++) {
    numbers[i] /= 2;
}
System.out.println(Arrays.toString(numbers));
```

Which of the following happens when you try to run it?

(circle one best answer)

A. The code has a static type error because it tries to assign `double` values to `int` variables.
B. The code will have a dynamic `ArrayIndexOutOfBoundsException`.
C. The code runs successfully and prints out `[0, 1, 1, 2, 2]`.
D. The code runs successfully and prints out `[0.5, 1, 1.5, 2, 2.5]`.

**Solution.** C. The code runs without error. Since you’re performing integer division and updating the results, the output of the program will be integer values.
(e) In a version control system, what would a cycle in the commit history graph indicate? (circle one best answer)
A. One commit undid the changes of a previous commit.
B. Commits were made in multiple clones of the same repository but not yet merged.
C. One commit is the result of merging multiple diverging changes.
D. This occurs when you cannot automatically merge two changes.
E. This is impossible.

Solution. E. If a cycle occurred, that would mean that a commit was its own ancestor—that would be very bad!

(d) Consider the following function:

```java
public static double SquarePyramidVolume(int h, int s) {
    double pyramidDenominator = 3.0;
    return s*s*h/pyramidDenominator;
}
```

What are some constructive code review comments you could make? (circle all that apply)
A. A better method name would be a verb phrase and written in camelCase.
B. The extra variable pyramidDenominator is unnecessary, and you could replace the return statement in the function with \( s^2 \times h / 3 \).
C. The variable pyramidDenominator should be a final constant.
D. The parameters to the function should be renamed to something more descriptive like height and baseWidth.
E. The variable pyramidDenominator should be named p to match the style of the method parameters.

Solution. A, C, and D. B would result in the addition of a magic number, and the suggested variable name in E is not descriptive. A, C, and D are good suggestions mentioned in the Code Review reading.

(e) Suppose you have the following class:

```java
public class Name {
    private final String name;
    ...
    public boolean equals(Object obj) {
        if (!(obj instanceof Name)) return false;
        Name that = (Name)obj;
        return this.name.toLowerCase().equals(that.name);
    }
}
```

Which of the following expressions return true? (circle all that apply)
A. `new Name("Mary").equals(new Name(""))`
B. `new Name("Mary").equals(new Name("Mary"))`
C. `new Name("Mary").equals(new Name("MARY"))`
D. `new Name("Mary").equals(new Name("mary"))`
E. `new Name("mary").equals("mary")`

Solution. D, because the equals method first converts "Mary" to lowercase "mary", and then compares it with that.name. Note that E returns false, because the ‘obj’ being passed is a ‘String’, not a ‘Name’.
(f) Which of the following are properties of an equivalence relation that this `Name.equals()` method violates? Ignore null references. (circle all that apply)

A. antisymmetry  
B. invariance  
C. reflexivity  
D. rep independence  
E. symmetry  

**Solution.** C, E. Any name with capital letters will never compare equal to itself, and `new Name("Mary") . equals(new Name("mary"))` is true but `new Name("mary") . equals(new Name("Mary"))` is false.