MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2021 Quiz 2

You have 50 minutes to complete this quiz. There are **5** problems. The quiz is open-book: you may access any 6.031 or other resources, but you may not communicate with anyone except the course staff.

This page automatically saves your answers as you work. Saved answers are marked with a green cloud-with-up-arrow icon. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz. There is no 'save' or 'submit' button.

If you want to ask a clarification question, visit whoosh.mit.edu/6.031 and click "raise hand" to talk to a staff member.

Good luck!

When the quiz starts, **before you begin**, please sign this honor statement.

> I affirm that I will not communicate with classmates or anyone else (other than 6.031 staff members) about anything related to this quiz until the solutions are officially released.

By entering your full name below (first and last name), you agree to this honor statement.

Problem	Points
1	21
2	21
3	22
4	16
5	20

In this quiz you will work with abstract datatypes representing rubber ducks that squeak when they are squeezed.

A squeak can be either loud or quiet:

```
public enum Squeak { LOUD, QUIET }
```

The mutable `MutDuck` is an interface declaring a single instance method:

```
// A mutable rubber duck that can squeak loudly a certain limited number of times.
public interface MutDuck {
    // If this duck has loud squeaks remaining,
    //   returns LOUD and modifies this duck to reduce that amount by one.
    // Otherwise, returns QUIET.
    public Squeak squeeze();
}
```

The immutable `ImDuck` is an interface declaring several methods:

```
// A threadsafe immutable rubber duck that can squeak loudly a certain limited number of times.
public interface ImDuck {
    // Returns the squeak this duck would make when squeezed:
    // LOUD if this duck has loud squeaks remaining, otherwise QUIET.
    public Squeak voice();

    // Returns the duck that results after squeezing this duck.
    // The returned duck has one fewer loud squeaks than this duck,
    // or zero loud squeaks, whichever is larger.
    public ImDuck squeezed();

    // Make an ImDuck with loudSqueaks remaining.
    // Requires loudSqueaks >= 0.
    // This method is threadsafe.
    public static ImDuck make(int loudSqueaks);
}
```

For brevity, this quiz uses `// comments like this` instead of `/** JavaDoc comments */` for specifications.

Problem ⤮1. (21 points)

Fill in the blanks of the class `ThreadsafeDuck` below, which should be a threadsafe implementation of `MutDuck` that uses `ImDuck` as its rep. Your rep for `ThreadsafeDuck` may not use any types other than `ImDuck`. You must use the monitor pattern to achieve thread safety. Do not change any of the provided code.

```
public class ThreadsafeDuck implements MutDuck {
```

Rep for `ThreadsafeDuck` (write only field declarations here):

```

```

```
    // Make a ThreadsafeDuck with loudSqueaks remaining.
    // Requires loudSqueaks >= 0.
    public ThreadsafeDuck(int loudSqueaks) {
```

Body of `ThreadsafeDuck` constructor (write only the code in the body, you cannot change the constructor signature or spec):

```

```

```
    }

    @Override
    public Squeak squeeze() {
```

Body of `squeeze` method (write only the code in the body, you cannot change the method signature or spec):

```

```

```
    }
}
```

Problem ⤮2. (21 points)

Implement `ImDuck` as a recursive data type with two concrete variants. The reps for the variants may not use any types other than `ImDuck`.

Datatype definition:

```

```

Define the `squeezed` operation for each of your variants, either in mathematical notation (example) or Java notation. If you choose Java notation, omit all other parts of the class, like rep fields, constructors, comments, etc – just provide the body of `squeezed()` for each variant.

Implement `ImDuck.make()` using your variants. This method must be recursive.

```
// Make an ImDuck with loudSqueaks remaining.
// Requires loudSqueaks >= 0.
// This method is threadsafe.
public static ImDuck make(int loudSqueaks) {
```

Body of `make` method (must be recursive):

```
}
```

Problem 3. (22 points)

Recall the creator operation for `ImDuck`:

```
  // Make an ImDuck with loudSqueaks remaining.
  // Requires loudSqueaks >= 0.
  // This method is threadsafe.
  public static ImDuck make(int loudSqueaks);
```

Ben Bitdiddle recognizes an opportunity to improve the program: "we can reuse ImDucks we've made before!" He defines a new creator `quickDuck`, with the same spec apart from its name:

```
  // Make an ImDuck with loudSqueaks remaining.
  // Requires loudSqueaks >= 0.
  // This method is threadsafe.
  public static ImDuck quickDuck(int loudSqueaks);
```

And implements it as follows:

use either this implementation:

```
 1    private static ImDuck cachedDuck = null;
 2    private static int loudSqueaksInCachedDuck = -1;

 3    public static ImDuck quickDuck(int loudSqueaks) {
 4        if (loudSqueaks > loudSqueaksInCachedDuck) {
              // we're gonna need a bigger duck
 5            cachedDuck = ImDuck.make(loudSqueaks);
 6            loudSqueaksInCachedDuck = loudSqueaks;
 7            return cachedDuck;
          }

 8        ImDuck duck = cachedDuck;
 9        int squeaksToSqueezeOut = loudSqueaksInCachedDuck - loudSqueaks;
10        while (squeaksToSqueezeOut > 0) {
11            duck = duck.squeezed();
12            --squeaksToSqueezeOut;
          }
13        return duck;
    }
```

or this implementation:

```
 1    private static ImDuck cachedDuck = null;
 2    private static int loudSqueaksInCachedDuck = -1;

 3    public static ImDuck quickDuck(int loudSqueaks) {
 4        if (loudSqueaks <= loudSqueaksInCachedDuck) {
 5            ImDuck duck = cachedDuck;
 6            int squeaksToSqueezeOut = loudSqueaksInCachedDuck - loudSqueaks;
 7            while (squeaksToSqueezeOut > 0) {
 8                duck = duck.squeezed();
 9                --squeaksToSqueezeOut;
              }
10            return duck;
          }

          // we're gonna need a bigger duck
11        cachedDuck = ImDuck.make(loudSqueaks);
12        loudSqueaksInCachedDuck = loudSqueaks;
13        return cachedDuck;
      }
```

Suppose we change all client code to call `quickDuck()` instead of `ImDuck.make()`, so that the entire execution of a particular single-threaded program sees this sequence of calls to `quickDuck()`:

use either this sequence:

```
quickDuck(3) ... quickDuck(5) ... quickDuck(1)
```

or this sequence:

```
quickDuck(5) ... quickDuck(3) ... quickDuck(1)
```

or this sequence:

```
quickDuck(1) ... quickDuck(5) ... quickDuck(3)
```

How many total times was `ImDuck.make()` called for the execution of the `quickDuck` calls above? Include recursion, and assume the program uses your implementation of `ImDuck`.

Louis Reasoner writes in his code review comment: "Wait, we can't just substitute `quickDuck()` everywhere we're using `ImDuck.make()`. I can get `quickDuck()` to return `null`, and `ImDuck.make()` never does that."

Louis is partly right and partly wrong.

First explain how Louis is right: give an example of a Java `int` value n such that `quickDuck(`n`)` returns `null`, and give the line numbers of the body of `quickDuck()` (lines 4-13) that are executed by this call.

n =

line numbers executed:

Now explain how Louis is wrong, using one sentence with at least two highly-relevant technical terms.

In another code review comment, Alyssa Hacker points out that `quickDuck` does not follow its spec, because it is not threadsafe. Give an example of a bad interleaving for `quickDuck`, using specific values for `n` and referring to line numbers in the code above. Each line of your interleaving should have the form **quickDuck(*n*) runs line *i*** (or **runs lines *i-j***) for some constant integers n, i, j.

Explain in one sentence why your interleaving is bad.

Problem ⤭4. (16 points)

Consider this function:

```
// Returns uppercase `letter` if `squeak` is LOUD, or
// lowercase `letter` if `squeak` is QUIET.
public static String encode(final String letter, final Squeak squeak) {
    if (squeak == LOUD) {
        letter.toUpperCase();
    } else {
        letter.toLowerCase();
    }
    return letter;
}
```

In one sentence, explain the mutability bug in the code above.

Now assume the bug in `encode` is fixed, and it compiles and satisfies its spec. Given a function `record` that produces a recording of a pair of ducks squeaking at each other:

use one of the following implementations:

```
public static String record(MutDuck huey, MutDuck dewey) {
    String result = "";
    while (true) {
        Squeak d = dewey.squeeze();
        Squeak h = huey.squeeze();
        if (h == QUIET && d == QUIET) {
            break;
        }
        result += encode("h", h) + encode("d", d);
    }
    return result;
}
```

```
public static String record(MutDuck huey, MutDuck dewey) {
    String result = "";
    while (true) {
        Squeak d = dewey.squeeze();
        Squeak h = huey.squeeze();
        if (d == QUIET && h == QUIET) {
            break;
        }
        result += encode("d", d) + encode("h", h);
    }
    return result;
}
```

```
public static String record(MutDuck huey, MutDuck dewey) {
    String result = "";
    while (true) {
        Squeak d = dewey.squeeze();
        Squeak h = huey.squeeze();
        result += encode("h", h) + encode("d", d);
        if (h == QUIET && d == QUIET) {
            break;
        }
    }
    return result;
}
```

```
public static String record(MutDuck huey, MutDuck dewey) {
    String result = "";
    while (true) {
        Squeak d = dewey.squeeze();
        Squeak h = huey.squeeze();
        result += encode("d", d) + encode("h", h);
        if (d == QUIET && h == QUIET) {
            break;
        }
    }
    return result;
}
```

Write a regular expression that matches exactly the set of strings that can be returned by `record`.

Problem ⤭5. (20 points)

Assume the variable `ducks` has type `List<ImDuck>`. Here is an expression that uses `map` and `filter` on `ducks`:

use either this expression:

```
ducks.stream()
     .map((x) -> x.squeezed())
     .map((y) -> y.voice())
     .filter((z) -> z == LOUD);
```

or this expression:

```
ducks.stream()
     .map((x) -> x.squeezed())
     .filter((y) -> y.voice() == LOUD)
     .map((z) -> z.voice());
```

The next few questions ask about static types. Where the type is a function, you may use either conventional mathematical notation (*domain* → *range*), or an appropriate Java interface type. All type parameters in a generic type must be replaced by a specific type.

For example, in this expression, the static type of the return value of `ducks.stream()` is `Stream<ImDuck>`.

What is the static type of the return value of `filter()` in this expression?

What is the static type of the entire subexpression `(z) -> ...`?

Now write an expression, using `map` and/or `filter`, that returns the elements of `ducks` that have exactly one loud squeak remaining.