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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>
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:

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

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

```java
// 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:

```java
// 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.

```java
public class ThreadsafeDuck implements MutDuck {

Rep for ThreadsafeDuck (write only field declarations here):

```java
private ImDuck duck;
```n

// 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):

```java
this.duck = ImDuck.make(loudSqueaks);
```n

}@Override

public Squeak squeeze() {

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

```java
synchronized (this) {
    Squeak squeak = this.duck.voice();
    this.duck = this.duck.squeezed();
    return squeak;
}
```n

}

}

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:

```java
ImDuck = LoudDuck(nextDuck:ImDuck) + QuietDuck
```n
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.

\[
\text{squeezed(QuietDuck)} = \text{QuietDuck} \\
\text{squeezed(LoudDuck(next:ImDuck))} = \text{next}
\]

or

\[
\text{class QuietDuck} \{ \ \text{public ImDuck squeezed()} \{ \ \text{return this;} \} \} \\
\text{class LoudDuck} \{ \ \text{public ImDuck squeezed()} \{ \ \text{return this.nextDuck;} \} \}
\]

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

```java
// Make an ImDuck with loudSqueaks remaining.
// Requires loudSqueaks >= 0.
// This method is threadsafe.
public static ImDuck make(int loudSqueaks) {
    if (loudSqueaks == 0) return new QuietDuck();
    else return new LoudDuck(make(loudSqueaks-1));
}
```

Problem ×3. (22 points)

Recall the creator operation for ImDuck:

```java
// 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:

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

And implements it as follows:
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()`:

`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.

10 for sequence 3-5-1; 6 for sequence 5-3-1; 8 for sequence 1-5-3

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 = -1

line numbers executed:
4,8-10,13 for top implementation; 4-7,10 for bottom implementation

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

-1 violates the precondition that loudSqueaks >= 0, so the implementer doesn't have to satisfy the postcondition anymore, and can behave arbitrarily.

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.

One possible bad interleaving is a race to cache the first duck:
quickDuck(1) runs 4-5 (for top implementation) or 4,11 (for bottom implementation);
quickDuck(2) runs the same lines
both calls then finish and return.
Incorrect answers: race on local variables (duck or squeaksToSqueezeOut), or assuming cachedDuck points to a mutable object that can affect the local variable duck.
Explain in one sentence why your interleaving is bad.

quickDuck(1) ends up returning the cachedDuck created by quickDuck(2), which has 2 squeaks in it.
**Problem ×4.** (16 points)

Consider this function:

```java
// 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.

*letter is a String, which is immutable, so toUpperCase and toLowerCase are producers, not mutators, and this code discards their return values and returns the original letter.*

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:*

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

```
impl #1: (HD)*((hD)*|(Hd)*); #2: (DH)*((dH)*|(Dh)*); #3: (HD)*((hD)*|(Hd)*)hd; #4: (DH)*((dH)*|(Dh)*)dh
```

**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:*

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

*or this expression:*

```java
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 (\textit{domain} \rightarrow \textit{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 \texttt{ducks.stream()} is \texttt{Stream<ImDuck>}.

What is the static type of the return value of \texttt{filter()} in this expression?

\begin{tabular}{|l|}
\hline
\textbf{top expression:} Stream<Squeak>; \textbf{bottom expression:} Stream<ImDuck> \\
\hline
\end{tabular}

What is the static type of the entire subexpression \((z) \rightarrow \ldots\)?

\begin{tabular}{|l|}
\hline
\textbf{top expression:} Squeak \rightarrow \text{boolean}; \textbf{bottom expression:} ImDuck \rightarrow \text{Squeak} \\
\hline
\end{tabular}

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

\begin{tabular}{|l|}
\hline
\textbf{one possible answer:} \\
\texttt{ducks.stream()} \\
\hspace{1em} .filter((duck) -> duck.voice() == \texttt{LOUD}) \\
\hspace{1em} .filter((duck) -> duck.squeezed().voice() == \texttt{QUIET}) \\
\hline
\end{tabular}