MIT
6.031TS Software Construction — TypeScript Pilot
Prof. Rob Miller and Max Goldman

rcm

# 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.

*solutions*

| 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:

```
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.
interface MutDuck {
    // If this duck has loud squeaks remaining,
    //   returns LOUD and modifies this duck to reduce that amount by one.
    // Otherwise, returns QUIET.
    squeeze(): Squeak;
}
```

The immutable `ImDuck` is an interface declaring several methods, along with a factory function:

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

    // 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.
    squeezed(): ImDuck;
}

// Make an ImDuck with loudSqueaks remaining.
// Requires integer loudSqueaks >= 0.
function makeImDuck(loudSqueaks: number): ImDuck { ... }
```

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

**Problem ⤮1.** (21 points)

Fill in the blanks of the class `MyDuck` below, which implements the `MutDuck` interface and uses `ImDuck` as its rep. Your rep for `MyDuck` may not use any types other than `ImDuck`. Do not change any of the provided code.

```
class MyDuck implements MutDuck {
```

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

```
private duck: ImDuck;
```

```
    // Make a MyDuck with loudSqueaks remaining.
    // Requires integer loudSqueaks >= 0.
    public constructor(loudSqueaks: number) {
```

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

```
this.duck = makeImDuck(loudSqueaks);
```

```
    }

    // same spec as MutDuck.squeeze()
    public squeeze(): Squeak {
```

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

```
const squeak: Squeak = this.duck.voice();
this.duck = this.duck.squeezed();
return squeak;
```

```
    }
}
```

**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:

```
ImDuck = LoudDuck(nextDuck:ImDuck) + QuietDuck
```

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

```
squeezed(QuietDuck) = QuietDuck
squeezed(LoudDuck(next:ImDuck)) = next

or

class QuietDuck { public squeezed():ImDuck { return this; } }
class LoudDuck { public squeezed():ImDuck { return this.nextDuck; } }
```

Implement `makeImDuck()` using your variants. This method must be recursive, not iterative.

```
// Make an ImDuck with loudSqueaks remaining.
// Requires integer loudSqueaks >= 0.
function makeImDuck(loudSqueaks: number): ImDuck {
```

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

```
if (loudSqueaks === 0) return new QuietDuck();
else return new LoudDuck(make(loudSqueaks-1));
```

}

**Problem ⤮3.** (22 points)

Recall the creator operation for `ImDuck`:

```
// Make an ImDuck with loudSqueaks remaining.
// Requires integer loudSqueaks >= 0.
function makeImDuck(loudSqueaks: number): ImDuck
```

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 integer loudSqueaks >= 0.
function quickDuck(loudSqueaks: number): ImDuck
```

And implements it as follows:

```
let cachedDuck: ImDuck = makeImDuck(0);
let loudSqueaksInCachedDuck: number = 0;

function quickDuck(loudSqueaks: number): ImDuck {
    if (loudSqueaks > loudSqueaksInCachedDuck) {
        // we're gonna need a bigger duck
        cachedDuck = makeImDuck(loudSqueaks);
        loudSqueaksInCachedDuck = loudSqueaks;
    }

    let duck: ImDuck = cachedDuck;
    let squeaksToSqueezeOut: number = loudSqueaksInCachedDuck - loudSqueaks;
    while (squeaksToSqueezeOut > 0) {
        duck = duck.squeezed();
        --squeaksToSqueezeOut;
    }
    return duck;
}
```

Suppose we change all client code to call `quickDuck()` instead of `ImDuck.make()`, so that the entire execution of a particular 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 `makeImDuck()` called for the execution of the `quickDuck` calls above? Include recursion, and assume the program uses your implementations of `ImDuck` and `makeImDuck`.

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

Alyssa Hacker likes `quickDuck()`, but says: “We can do better.” She observes that client programs tend to create a bunch of ducks before calling any operations on them, for example:

```
1  function main(): void {
2      // let's get our ducks in a row
3      const aDuck: ImDuck = quickDuck(102);
4      const bDuck: ImDuck = quickDuck(55);
5      const cDuck: ImDuck = quickDuck(189);

6      // now start the chorus
7      solo(aDuck);
8      duet(bDuck, cDuck);
9  }
```

So she proposes `quickerDuck()`:

```
// Make an ImDuck with loudSqueaks remaining,
// but defer constructing it until actually needed.
// Requires integer loudSqueaks >= 0.
function quickerDuck(loudSqueaks: number): Promise<ImDuck>
```

Alyssa explains: “quickerDuck() can wait as long as it can, to see how big the ducks might get, before actually constructing any ducks.” But Alyssa admits that `quickerDuck` is not a drop-in replacement for `quickDuck`, because it requires changes to client code.

For the `main()` code above, after replacing `quickDuck` with `quickerDuck`, what other changes would be required to make `main()` work again? Make sure to follow Alyssa’s plan to wait to see how big the ducks are before actually constructing any.

Put each kind of change in one box below, starting the box with **line *i*:** (or **lines *i-j*:** or **lines *i,j,k*:**), followed by a brief description of the change. Group similar changes for clarity and simplicity. You don’t need to use all the boxes.

line 1: add async

line 1: change return type from void to Promise<void>

lines 3-5: change declared type from ImDuck to Promise<ImDuck>

lines 7-8: put await in front of each variable

Alyssa and Ben divide up the implementation of `quickerDuck`. Alyssa will do the tricky bit involving promises, with this spec:

```
// Returns a promise such that as soon as its value is requested by a client
// (e.g. by calling then()), `callback` is called to provide the promise's value.
function makePromise(callback: () => ImDuck):Promise<ImDuck>;
```

Ben now writes `quickerDuck` using `makePromise`. Here is a skeleton of his code:

```
let biggest: number = 0;

function quickerDuck(loudSqueaks: number): Promise<ImDuck> {
    biggest = Math.max(biggest, loudSqueaks);

    return makePromise(() => {
        TODO
    });
}
```

Fill in the code that should replace `TODO` in the body of the callback. **Use `quickDuck` in your solution**, not `makeImDuck`. Keep in mind that `quickerDuck` should be better than `quickDuck`, i.e., the example `main()` above should call `makeImDuck` fewer times after substituting `quickerDuck` for `quickDuck`.

```
quickDuck(biggest);
return quickDuck(loudSqueaks);
```

**Problem 4.** (16 points)

Consider this function:

```
// Returns uppercase `letter` if `squeak` is LOUD, or
// lowercase `letter` if `squeak` is QUIET.
function encode(letter: string, squeak: Squeak): string {
    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:*

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

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

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

```
function record(huey: MutDuck, dewey: MutDuck): string {
    let result: string = "";
    while (true) {
        let d: Squeak = dewey.squeeze();
        let h: Squeak = 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`.

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 `Array<ImDuck>`. Here is an expression that uses `map` and `filter` on `ducks`:

*use either this expression:*

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

*or this expression:*

```
ducks
  .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 TypeScript syntax. All type parameters in a generic type must be replaced by a specific type.

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

top expression: Array<Squeak>; bottom expression: Array<ImDuck>

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

top expression: Squeak -> boolean;  bottom expression: ImDuck -> Squeak

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

```
one possible answer:
ducks
  .filter((duck) => duck.voice() === LOUD)
  .filter((duck) => duck.squeezed().voice() === QUIET)
```