6.102 Quiz 1

Spring 2025

- You have **80 minutes** to complete this exam. There are **4 problems**.
- The exam is **closed-book** and closed-notes, but you are allowed to bring a single 8.5×11" double-sided page of notes, handwritten directly on the paper (not computer-printed or photocopied), readable without a magnifying glass, created by you.
- You may also use blank scratch paper.
- You may use **nothing else on your computer** or other devices: no 6.102 website; no TypeScript interpreter or programming tools; no web search or discussion with other people.
- Before you begin: you must **check in** by having the course staff scan the QR code at the top of the page.
- This page **automatically saves your answers** as you work. 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 exam.
- If you feel the need to **write a note to the grader**, you can click the gray pencil icon to the right of the answer.
- If you have a question, or need to use the restroom, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- To **leave early**: enter *done* at the very bottom of the page, and show your screen with the check-out code to a staff member.
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

You can **open this introduction in a separate tab**.

`Schedule` is an **immutable** abstract data type representing the days and times that a course meets every week.

A course may only meet during business hours (Monday through Friday, 9am to 5pm).

Every time period in a schedule starts and ends on an hour or half-hour boundary.

`Schedule` is an interface with at least one operation, sketched below (more detailed specs are found in relevant questions):

- `lookup(day: Weekday, hour: number): Array<number>` tests whether the course meets during a given hour of a given day, and returns the part of the hour that it meets, expressed as a half-open interval of minutes. For example, if `lookup(Weekday.Mon, 3)` returns the 2-element array `[30,60]`, then the course is meeting Monday 3:30-4pm (but not 3-3:30pm, and not necessarily stopping at 4pm).

As a reminder, some people say “3:30pm” (using AM/PM notation), and other people say “15:30” (using 24-hour notation). Both are the same time of day. But nobody calls this time “3.5pm” or “15.5”.

The `lookup` operation also uses this abstract data type:

```
enum Weekday { Mon, Tue, Wed, Thu, Fri };
```

The `Schedule` interface has several implementation classes that are public and visible to clients:

- class `MWF` represents a course schedule that meets on Mondays, Wednesdays, and Fridays for 1 hour at the same time each day, starting on an hour boundary
- class `TR` represents a course schedule that meets on Tuesdays and Thursdays for 1.5 hours at the same time each day
- class `Irregular` represents a course schedule that meets on any set of days, for any periods of time

Problem 1

Here is a spec for one of the operations of `Schedule`:

```
/**
 * Schedule is an immutable type representing the days and times
 * that a course meets every week.
 */
interface Schedule {
    //... parts omitted

    /**
     * @param day   day to look up
     * @param hour  hour to look up; an integer 1-12 in AM/PM time
     * @returns if the class meets during a time matching `day` and `hour`,
     *          then returns a 2-element array [starting minute, ending minute]
     *          representing the half-open interval of the time period it meets
     *          during that hour; otherwise returns an empty array
     */
    lookup(day: Weekday, hour: number): Array<number>;

}
```

Critique the following proposed partitions for `lookup()`. Your answers should fit in the boxes without scrolling.

`day` is a weekday, or `day` is on the weekend

> Bad – includes illegal subdomains ("weekend" can't even be expressed with the `Weekday` type)

`hour` is morning (9-11am), noon, or afternoon (1-4pm)

> Bad – subdomains do not cover the space. The precondition allows calling with hours outside of business hours (e.g. 6am/6pm).

the returned array contains no numbers, or 0, or 30, or 60

> Bad – subdomains overlap in, e.g. `[0,60]`

Now provide one additional good partition for `lookup()`, using an input that none of the previous partitions used explicitly. Your answer should fit in the box without scrolling.

Partition on `this`: meets on `day`, doesn't meet on `day`
OR
Partition on `this`: created by `MWF`, created by `TR`, created by `Irregular`
OR
(other good partitions on `this` are also possible)

Start with the original specification, restated here:

```
// ... overview omitted
interface Schedule {
    //... parts omitted

    /**
     * @param day   day to look up
     * @param hour  hour to look up; an integer 1-12 in AM/PM time
     * @returns if the class meets during a time matching `day` and `hour`,
     *          then returns a 2-element array [starting minute, ending minute]
     *          representing the half-open interval of the time period it meets
     *          during that hour; otherwise returns an empty array
     */
    lookup(day: Weekday, hour: number): Array<number>;
}
```

For each of the following changes (considered separately), say whether it would make the spec STRONGER, WEAKER, SAME, or INCOMPARABLE to the original spec, and explain why in one sentence.

```
@param hour   hour to look up; an integer 0-23 in either AM/PM time or 24-hour time
```

STRONGER spec, because precondition is weaker.

```
@throws error if hour is outside of business hours
```

INCOMPARABLE, because original spec returned empty array in those cases, and this spec now requires throwing an error.

Assume you already have a good test suite for the original specification. Say how each of the changes would affect that test suite. Consider both partitions and test cases; whether the original test suite still works; and what changes should be made to update it. Your answers should fit in the box without scrolling.

```
@param hour   hour to look up; an integer 0-23 in either AM/PM time or 24-hour time
```

Existing test cases still work; needs new partition (or new subdomains in existing partition) and test cases that cover 24-hour times (such as 13-23).

```
@throws error if hour is outside of business hours
```

Original tests won't all pass; test cases for non-business hours need to be changed to check for an error; partitions based on returned result may need to add a subdomain for throwing an error instead of returning.

Problem ×2

Louis Reasoner is working on the implementation of `MWF`. Starting with the original code for `lookup()`, shown on the left, Louis refactors it to the code on the right:

```
// ... overview omitted
class MWF implements Schedule {
    private hour: number;

    //... parts omitted

    /**
     * @inheritdoc
     */
    public lookup(day: Weekday, hour: number): Array<number> {
        if (day === Weekday.Tue || day === Weekday.Thu) {
            return []; // wrong day, doesn't meet
        }
        if (hour !== this.hour) {
            return []; // wrong hour, doesn't meet
        }
        return [0, 60]; // meets for the full hour
    }

}
```

```
const NO_TIME = [];
const FULL_HOUR = [0, 60];

// ... overview omitted
class MWF implements Schedule {
    private hour: number;

    //... parts omitted

    /**
     * @inheritdoc
     */
    public lookup(day: Weekday, hour: nu
        if (day === Weekday.Tue || day =
            return NO_TIME; // wrong day
        }
        if (hour !== this.hour) {
            return NO_TIME; // wrong hou
        }
        return FULL_HOUR; // meets for
    }

}
```

You are code-reviewing Louis's changes (the change from the left code to the right code, above). First make a positive comment about what he did, referring to specific design principle(s). Your answer should fit in the box without scrolling.

> Louis replaced some magic numbers with good names, and also DRYed out his code that way. The changes made his code more ETU and more RFC.

Now criticize Louis's change, focusing on whether it is safe from bugs. Your answer should fit in the box without scrolling.

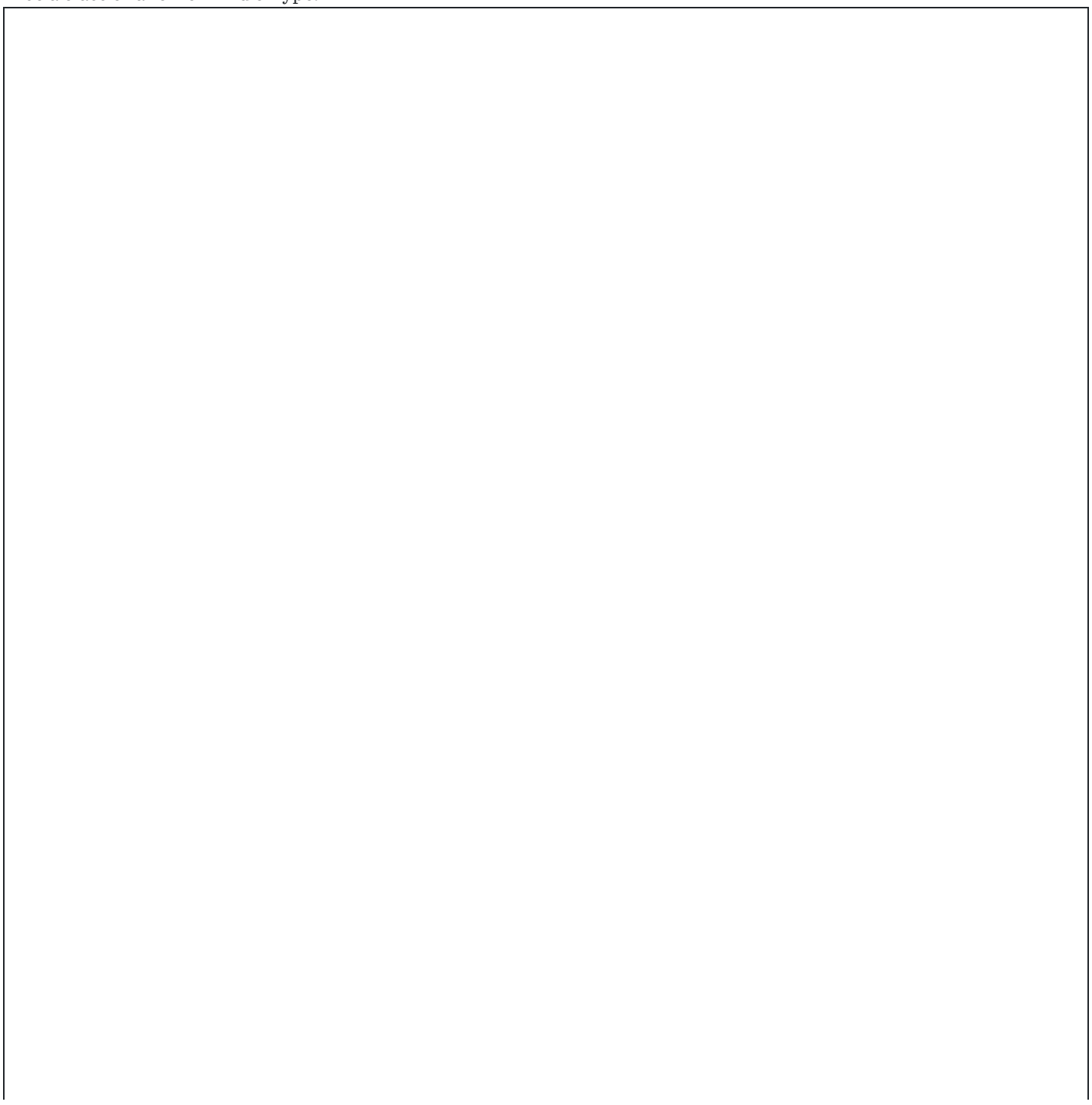
> Louis's solution means that many clients may end up with aliases to the mutable arrays `NO_TIME` and `FULL_HOUR`, and one of them may mutate the array, causing an aliasing bug.

Alyssa Hacker steps in and proposes changing the return type of the `lookup` operation:

```
// ... overview omitted
interface Schedule {
    // ... parts omitted

    lookup(day: Weekday, hour: number): Minutes;
}
```

Design and implement a `Minutes` abstract data type. Your ADT should:

- be well-designed and well-documented;
- preserve the ability of implementers of `lookup()` to return the values they need to return;
- preserve the ability of clients of `lookup()` to get information out of those values (such as the starting time);
- use static checking where possible and dynamic checking where appropriate;
- adopt the good aspects of Louis's idea above, but without the risky aspects;
- be a class or another kind of type.



one possible solution:

```
/**
 * Minutes is an immutable type representing an interval in an hour
 * (limited to half-hour boundaries), or no time at all.
 */
class Minutes {
    // constant values of Minutes, public to clients
    public static readonly NO_TIME: Minutes = new Minutes();
    public static readonly FIRST_HALF_HOUR: Minutes = new Minutes(0, 30);
    public static readonly SECOND_HALF_HOUR: Minutes = new Minutes(30, 60);
    public static readonly FULL_HOUR: Minutes = new Minutes(0, 60);

    // RI: startMinute and endMinute are both undefined
    //     OR (startMinute is 0 or 30
    //         and endMinute is 30 or 60)
    // AF(startMinute, endMinute) =
    //         no time interval if startMinute or endMinute is undefined
    //      or an interval
    // Safety from rep exposure:
    //      rep fields are immutable and unreassignable
    private constructor (
        public readonly startMinute?: number,
        public readonly endMinute?: number
    ) {
        this.checkRep();
    }

    private checkRep(): void {
        if (this.startMinute !== undefined && this.endMinute !== undefined) {
            assert(this.startMinute === 0 || this.startMinute === 30);
            assert(this.endMinute === 30 || this.endMinute === 60);
        }
    }

    /** @returns true if and only if this represents a time interval */
    public exists(): boolean {
        return this.startMinute !== undefined;
    }

}
```

another solution:

```
enum Minutes { NO_TIME, FIRST_HALF, SECOND_HALF, FULL_HOUR };

/** @param minutes
 *  @returns starting minute of minutes interval
 *  @throws error if minutes is NO_TIME
 */
function getStartingMinute(minutes: Minutes): number {
    if (minutes === NO_TIME) { throw new Error("can't get starting time of no-time"); }
```

```
        return minutes == SECOND_HALF ? 30 : 0;
    }

    /** @param minutes
     *  @returns ending minute of minutes interval
     *  @throws error if minutes is NO_TIME
     */
    function getEndingMinute(minutes: Minutes): number {
        if (minutes === NO_TIME) { throw new Error("can't get ending time of no-time"); }
        return minutes == FIRST_HALF ? 30 : 60;
    }
```

Now use your new ADT to fill in the blanks in the implementation of `lookup` for `MWF`:

```
// ... overview omitted
class MWF implements Schedule {
    private hour: number;

    //... parts omitted

    /**
     * @inheritdoc
     */
    public lookup(day: Weekday, hour: number): Minutes {
        if (day === Weekday.Tue || day === Weekday.Thu) {
            return _____________; // wrong day, doesn't meet
```

[ ]

```
        Minutes.NO_TIME

        }
        if (hour !== this.hour) {
            return _____________; // wrong hour, doesn't meet
```

[ ]

```
        Minutes.NO_TIME

        }
        return _____________; // meets for the full hour
```

[ ]

```
    }
}
```

```
        Minutes.FULL_HOUR
```

Problem 3

Here is an implementation of TR:

```
/**
 * TR is an immutable type representing a course schedule that meets
 * Tuesdays and Thursdays for 1.5 hours, at the same time each day.
 */
class TR implements Schedule {

    /**
     * Make a TR schedule.
     * @param hour a legal course start time measured in hours (e.g. 9.5 is 9:30), using either AM/PM
     */
    public constructor(
        public readonly hour: number
    ) {
        checkRep();
    }

    // ... parts omitted
}
```

Give the rep space of TR.

number

Write the rep invariant of TR. Your answer should fit in the box without scrolling.

hour*2 is an integer hour is in [1, 3.5] or hour is in [9, 15.5]

Write the abstraction function for TR. Your answer should fit in the box without scrolling.

AF(hour) = the schedule that meets on Tuesdays and Thursdays for 1.5 hours
starting at hh:mm in 24-hour time, where
hh = (floor(hour) + 12) mod 24
mm = 00 if hour is an integer else 30

or

AF(hour) = the schedule that meets on Tuesdays and Thursdays for 1.5 hours
starting (`hour` + 12) hours after midnight if 1 <= hour <= 3.5 or `hour` hours after midnight otherwise

Does `===` provide behavioral equality for the `TR` data type? If so, explain why. If not, give a specific counterexample. Your answer should fit in the box without scrolling.

No. `new TR(1)` and `new TR(1)` are two values that are unequal according to `===`, but are equal by behavioral equality, since the only operation `lookup()` returns the same results for both values.

Write `equalValue()` for `TR`. For this part, you should assume `that` is also a `TR` object (as the type signature shows).

```
/** Return true if and only if this and that represent the same abstract value. */
public equalValue(that: TR): boolean {
```

```
}
```

```
    return (this.hour % 12) === (that.hour % 12);
```

OR

```
    const to24HourTime = (hour) => hour < 4 ? hour + 12 : hour;
    return to24HourTime(this.hour) === to24HourTime(that.hour);
```

(other solutions also possible)

Ben Bitdiddle says that we really need an `equalValue()` operation at the level of the `Schedule` data type (in which `that` is any `Schedule`), so Louis Reasoner proposes implementing it this way for `TR`:

```
/** Return true if and only if this and that represent the same abstract value. */
public equalValue(that: Schedule): boolean {
    return isDeepStrictEqual(this, that);
}
```

Explain the problem with Louis's implementation, including a brief example. Your answer should fit in the box without scrolling.

`isDeepStrictEqual()` compares objects field-by-field, so it will say a `TR` object using AM/PM time is different from a `TR` object using 24-hour time even if they represent the same time of day. It will also always say that a `TR` object is different from an `Irregular` object, even if they represent the same Tuesday/Thursday schedule, because their reps will have different fields.

Problem ×4

Ben Bitdiddle finds the `lookup()` operation unhelpful to clients, so he is considering adding one of these alternative ways to get the hourly time slots out of a `Schedule`:

- `timeSlots()` is a generator that produces a sequence of time slots
- `forEach()` calls a client-provided function on each time slot (similar to `map()` for arrays)

Here are sketches of their specs:

```
// ... overview omitted
interface Schedule {
    // ... parts omitted

    /** ... spec omitted */
    timeSlots(): Generator<{day: Weekday, hour: number, startMinute: number, endMinute: number}>;

    /** ... spec omitted */
    forEach(f: (day: Weekday, hour: number, startMinute: number, endMinute: number)=>void): void;
}
```

Fill in the blanks below for how `timeSlots()` and `forEach()` would be implemented on the `MWF` class. (Ben hasn't decided yet which operation he likes better.) Your answers should fit in the boxes without scrolling.

```
/**
 * MWF is an immutable course schedule that meets on Mondays, Wednesdays, and Fridays
 * for 1 hour at the same time each day, starting on an hour boundary.
 */
class MWF implements Schedule {
    private hour: number;

    // ... parts omitted

    /** @inheritdoc */
    *timeSlots(): Generator<{day: Weekday, hour: number, startMinute: number, endMinute: number}> {
```

```

```

```
    for (const day of [Weekday.Mon, Weekday.Wed, Weekday.Fri]) {
        yield { day: day, hour: this.hour, startMinute: 0, endMinute: 60 };
    }

    OR

    yield { day: Weekday.Mon, hour: this.hour, startMinute: 0, endMinute: 60 };
```

```
        yield { day: Weekday.Wed, hour: this.hour, startMinute: 0, endMinute: 60 };
        yield { day: Weekday.Fri, hour: this.hour, startMinute: 0, endMinute: 60 };
    }

    /** @inheritdoc */
    forEach(f: (day: Weekday, hour: number, startMinute: number, endMinute: number)=>void): void {
```

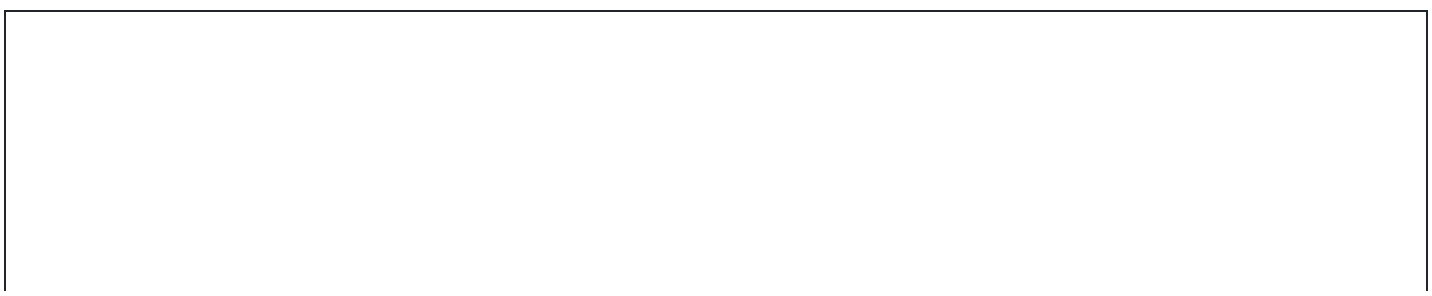
```
        for (const day of [Weekday.Mon, Weekday.Wed, Weekday.Fri]) {
            f(day, this.hour, 0, 60);
        }

        OR

        f(Weekday.Mon, this.hour, 0, 60);
        f(Weekday.Wed, this.hour, 0, 60);
        f(Weekday.Fri, this.hour, 0, 60);

    }
}
```

To show how a client might use these operations, Ben uses a function `totalTime()` that simply adds up the the time spent in class each week. First he writes the function using `timeSlots()`:

```
/** @returns total classtime in minutes */
function totalTime(sched: Schedule): number {
    let total = 0;
    for (const { day, hour, startMinute, endMinute } of sched.timeSlots()) {
        total += (endMinute - startMinute);
    }
    return total;
}
```

Rewrite `totalTime()` using `forEach()` instead. Remember that there are several ways to define functions, so if you don't know the syntax for one way, try another way that is more familiar to you. Your answer should fit in the box without scrolling.

```
/** @returns total classtime in minutes */
function totalTime(sched: Schedule): number {
```



}

```
    let total = 0;
    sched.forEach( (day, hour, startMinute, endMinute) => {
        total += (endMinute - startMinute);
    });
    return total;
```

Classify `forEach()`, as an operation of the `Schedule` data type.

observer