# 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

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

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

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
```

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

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
```

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

# 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: number): Array<n
        if (day === Weekday.Tue || day === Weekday.Thu
            return NO_TIME; // wrong day, doesn't meet
        }
        if (hour !== this.hour) {
            return NO_TIME; // wrong hour, doesn't mee
        }
        return FULL_HOUR; // meets for the full hour
    }

}
```

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.

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

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.

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
```

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

```
        }
        return ______________; // meets for the 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 or 24-hour time
     */
    public constructor(
        public readonly hour: number
    ) {
        checkRep();
    }

    // ... parts omitted
}
```

Give the rep space of TR.

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

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

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.

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 {
```

```
}
```

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.

# 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}> {
```

```

```

```
    }

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

```

```

```
    }
}
```

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 {
```

```
}
```

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