Problem Set 4: Memory Scramble

The deadlines for this problem set are shown on the **course calendar**.

The purpose of this problem set is to explore programming with concurrent use of a shared mutable data type.

Design Freedom and Restrictions

On this problem set, you have substantial design freedom.

Pay attention to the **PS4 instructions** in the provided code. You must satisfy the provided specifications for certain functions in `board.ts` and `commands.ts`. You may rewrite or delete any of the other provided code, add new classes and methods, etc. On this problem set, unlike previous problem sets, we will not be running your tests against any other implementations.

Take care that the coordinate system of your game board matches the specifications: *(row, column)* coordinates start at *(0, 0)* in the top left corner, increasing vertically downwards and horizontally to the right.

It is your responsibility to examine Didit feedback and make sure your code compiles and runs for grading, but you must do your own testing to ensure correctness.

Get the code

To get started,

1. Ask Didit to create a remote `psets/ps4` repository for you on github.mit.edu.
2. Clone the repo. Find the `git clone` command at the top of your Didit assignment page, copy it entirely, paste it into your terminal, and run it.
3. Run `npm install`, then open in Visual Studio Code. See Problem Set 0 for if you need a refresher on how to create, clone, or set up your repository.

Overview

On this problem set, you will build a networked multiplayer version of **Memory**, a.k.a. Concentration, the game in which you turn over face-down cards and try to find matching pairs. Our version will have players turning over cards simultaneously, rather than taking turns. We'll call it Memory Scramble.

Problem 1: you will design and implement one or more ADTs for this game.

Problem 2: you will connect your ADTs to a game server that handles clients using HTTP.

Scramble in your browser.

ultaneous players using an asynchronous game board.

Problem 4: you will design and implement a map-like function (as in map/filter/reduce) that replaces cards on the board.

Problem 5: you will design and implement watching for changes to the board, which makes the Memory Scramble user interface more responsive.

Iterative Development Recommendation

Both automatic and manual grading for the **alpha** will focus more on your board and server without concurrency.

First, do problems 1 & 2 **without considering concurrency**: make your `Board` or other ADTs use synchronous methods, and don't worry about multiple concurrent players in your server. Implement the gameplay rules specified below by writing tests, choosing data structures, and writing code. Consider the abstraction functions and rep invariants of all your data types.

Then, revise your work in problem 3 to **make your system ready for concurrent players**. Revise your choice of ADT operations based on the need for asynchronous operations.

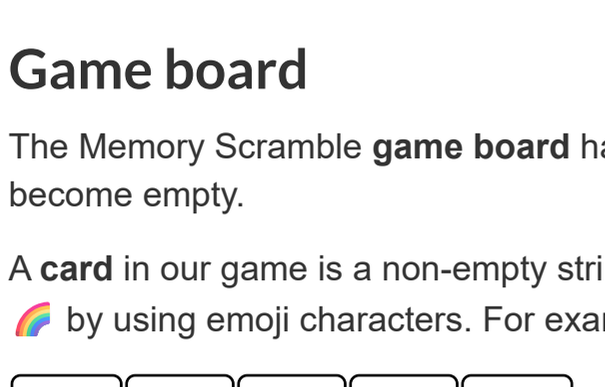
Finally, go on to problems 4 and 5.

Game board · **Gameplay rules** · **Example game transcript** · **Playing on the web**

Game board

The Memory Scramble **game board** has a grid of spaces. Each space starts with a card. As cards are matched and removed, spaces become empty.

A **card** in our game is a non-empty string of non-whitespace non-newline characters. This allows "pictures" like `Hello` and pictures like 🌈 by using emoji characters. For example:



Two cards **match** if they have the same string of characters.

All cards start face down. Players will turn them face up, and the cards will either turn face down again (if the player turned over cards that don't match) or be removed from the board (if they do match).

Game boards are **loaded from a file**. Here is the formal grammar for board files:

```
BOARD_FILE ::= ROW "x" COLUMN NEWLINE (CARD NEWLINE)+

CARD ::= [^\s\n\r]+
ROW ::= INT
COLUMN ::= INT
INT ::= [0-9]+
NEWLINE ::= "\r"? "\n"
```

In `BOARD_FILE`, `ROW` is the number of rows, `COLUMN` is the number of columns, and the cards are listed reading across each row, starting with the top row.

For example, this board of rainbows and unicorns:

… would be specified in a file as:

```
3x3↵
🦄↵
🦄↵
🌈↵
🌈↵
🌈↵
🦄↵
🌈↵
🦄↵
🌈↵
```

In this example, ↵ is used to show newlines. This board is provided in the `boards` folder of the problem set as `perfect.txt`.

A valid board file must have exactly `ROW` × `COLUMN` newline-terminated `CARD` lines.

Gameplay rules

Multiple **players** manipulate the cards on the board concurrently. They play the game by trying to turn over pairs of identical cards.

Here's an informal summary of the rules:

1. When a player turns over a *first card* in a pair, they *control* that card; but if someone else already controls it, the player *waits* until they can take control.
2. When the player turns over a *second card*, if the cards match, the player keeps control of them; otherwise, the player gives up control. The cards stay face up for now.
3. When the player makes their next move, if their previous cards matched, those cards are removed from the board; otherwise, if no one controls them, they turn face down.

Notice at least two differences from usual Memory game rules: you can "turn over" a card that is already face up, *waiting* until that card is available; and turned-over cards stay face up on the board until the player who turned them over makes another move.

Complete rules

First card: a player tries to turn over a first card by identifying a space on the board…

- **1-A** If there is no card there (the player identified an empty space, perhaps because the card was just removed by another player), the operation fails.
- **1-B** If the card is face down, it turns face up (all players can now see it) and the player **controls** that card.
- **1-C** If the card is already face up, but not controlled by another player, then it remains face up, and the player **controls** the card.
- **1-D** And if the card is face up and controlled by another player, the operation *waits*. The player will contend with other players to take control of the card at the next opportunity.

Second card: once a player controls their first card, they can try to turn over a second card…

- **2-A** If there is no card there, the operation fails. The player also relinquishes control of their *first card* (but it remains face up for now).
- **2-B** If the card is face up and controlled by a player (another player or themselves), the operation fails. To avoid deadlocks, the operation does *not* wait. The player also relinquishes control of their *first card* (but it remains face up for now).
- **—** If the card is face down, or if the card is face up but not controlled by a player, then:
  - **2-C** If it is face down, it turns face up.
  - **2-D** If the two cards are the same, that's a successful match! The player *keeps control* of both cards (and they remain face up on the board for now).
  - **2-E** If they are not the same, the player relinquishes control of both cards (again, they remain face up for now).

While one player is *waiting* to turn over a first card, other players continue to play normally. They do not wait, unless they also try to turn over a first card controlled by another player.

Failure in 1-A, 2-A, and 2-B means that the player fails to control a card (and in 2-A/B, also relinquishes control). The player can still continue to play the game.

After trying to turn over a *second card*, successfully or not, the player will try again to turn over a *first card*. When they do that, before following the rules above, they finish their previous play:

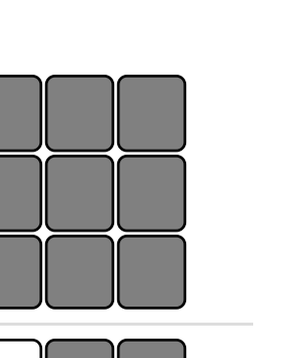
- **3-A** If they had turned over a matching pair, they control both cards. Now, those cards are removed from the board, and they relinquish control of them. Score-keeping is not specified as part of the game.
- **3-B** Otherwise, they had turned over one or two non-matching cards, and relinquished control but left them face up on the board. Now, for each of those card(s), if the card is still on the board, currently face up, and currently not controlled by another player, the card is turned face down.

If the player never tries to turn over a new first card, then the steps of 3-A/B never occur.

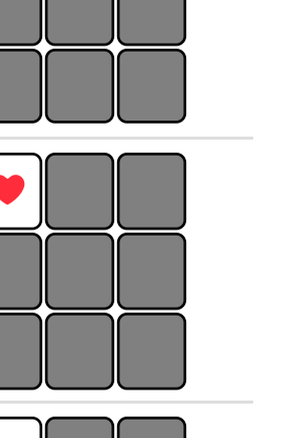
Example game transcript

We start with a 3×3 grid of face-down cards and 3 players: Alice, Bob, and Charlie.

Hover or tap on the board to show tooltips. →

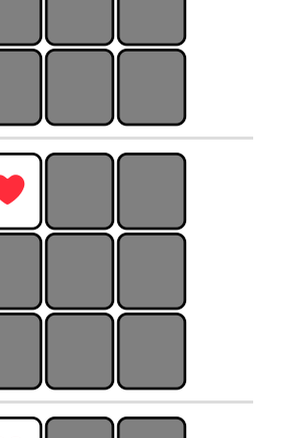


Alice turns over the top left card. She *controls* that card for the moment (rule 1-B).



Bob and Charlie also try to turn over that card at the same time.

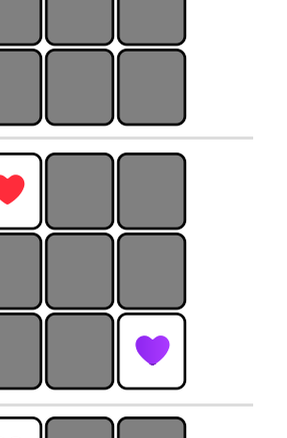
Both are now waiting for the chance to control it (1-D).



Alice turns over the bottom right card (2-C). It doesn't match.

Alice no longer controls any cards, but they stay face up for now (2-E).

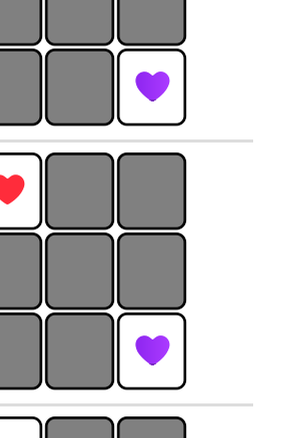
Either Bob or Charlie will now get to control the top left card.



Bob becomes the controller of that red heart card (1-C).

Alice hasn't made another move, so the purple heart is still face up.

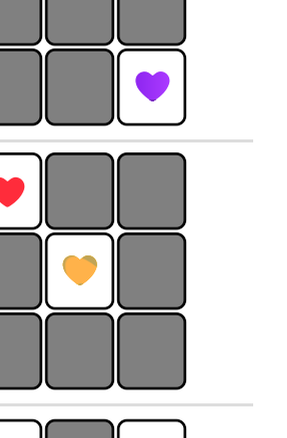
Charlie is still waiting.



Alice goes ahead and turns over a new first card, a yellow heart at the center of the board.

The red heart is controlled by Bob, so it stays face up (3-B).

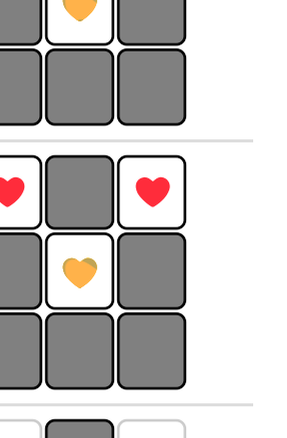
But the purple heart isn't controlled, so it turns face down (same 3-B).



While Alice is thinking about her second card, Bob turns over the top right card.

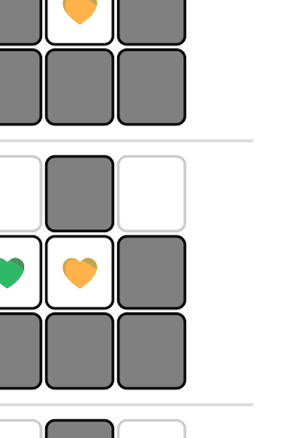
His first and second cards are a match!

Bob keeps control of the cards for now (2-D). Charlie is still waiting.



Bob turns over a new first card, a green heart on the left side.

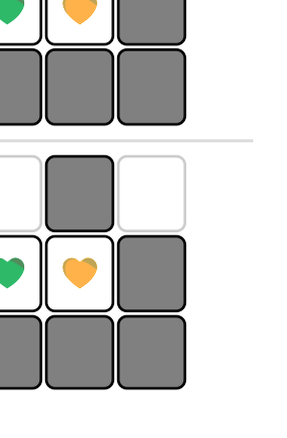
His matched red hearts are removed from the board, and he relinquishes control of them (3-A).



Charlie finally gets a chance at the top left card — but it's gone (1-A).

Charlie is now ready to turn over a new first card.

Alice and Bob each control one card and will try to turn over a matching second.



Playing on the web

To support multiplayer Memory Scramble games over the network, we define a HTTP protocol so the game can be played in a web browser. Code for the web server is already provided (in `server.ts`), but it relies on these provided function specifications in the code for this problem set, which you will have to implement:

- `parseFromFile()` from `board.ts`
- `look()`, `flip()`, `map()`, and `watch()` from `commands.ts`

In the protocol, each player identifies themselves by an ID, a nonempty string of alphanumeric or underscore characters of their choice. All requests with the same player ID are the actions of a single player.

For all requests, the server responds with a *board state*, showing the current state of the board from the player's perspective. A board state is described by the following grammar:

```
BOARD_STATE ::= ROW "x" COLUMN NEWLINE (SPOT NEWLINE)+

SPOT ::= "none" | "down" | "up " CARD | "my " CARD
CARD ::= [^\s\n\r]+
ROW ::= INT
COLUMN ::= INT
INT ::= [0-9]+
NEWLINE ::= "\r"? "\n"
```

In the board state, `ROW` is the number of rows, `COLUMN` is the number of columns, and the cards are listed reading across each row, starting with the top row.

`none` indicates no card in that location, `down` is a face-down card, `up` is a face-up card controlled by another player (or by no one), and `my` is a face-up card controlled by the player who sent the request.

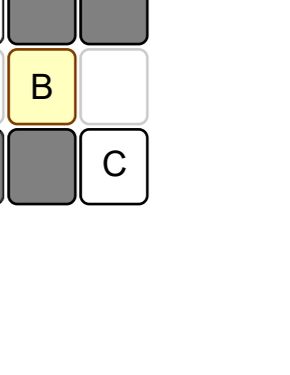
For example:

```
3x3↵
up␣A↵
down↵
down↵
none↵
my␣B↵
none↵
down↵
down↵
up␣C↵
```

In this example ␣ is used to show blank spaces and ↵ to show newlines.

There are no cards at (1,0) or (1,2).

The cards at (0,0), (1,1) and (2,2) are face up, and the player receiving this message controls the `B` card at (1,1).



Problem 1: Game board

Specify, test, and implement a **mutable** `Board` ADT to represent the Memory Scramble game board. It may be useful to define other ADTs as well.

Remember to include: a specification for every class and every method you write; abstraction functions, rep invariants, and safety from rep exposure arguments; `checkRep()` and `toString()`.

`Board` defines a static factory method for creating new boards that you must support:

- `parseFromFile(filename)`: creates a board by parsing a file in the format described above

For **parsing** board files, a parser generator (like ParserLib) is more complexity than you need.

- `String.split()` and `String.match()` probably can handle all the parsing and extracting.
- `fs.promises.readFile()` reads a file asynchronously. Be sure to use the promise-based version of this function (called `fsPromises.readFile` in the documentation, and referenced as `fs.promises.readFile` if you import `fs`), rather than the older callback-function-based version (also called `readFile` but found in `fs` itself).

To keep track of **players** and their state:

- Each player might simply be identified by their ID used in the command functions. All of the game state would be stored in the `Board`.
- Each player might be represented using an ADT, *e.g.* `Player`. Some of the player's state might be stored in the `Player` object, or the `Board` might store all the state.

You will need to track **who controls which cards** and the **status of each card** according to the gameplay rules.

Iterative development recommendation: implement a synchronous board first (*i.e.* with no `async` methods other than `parseFromFile`), and connect it to a server in Problem 2. The **alpha grading** will focus more on playing the game without concurrency. Then revise for correct behavior with concurrent players in Problem 3.

To implement **waiting** according to the gameplay rules, use asynchronous methods and `await`, or callback functions. The board should **not** busy-wait.

You decide **which actions are atomic**, since the rules do not specify. In rule **3-B**, for example, turning the two cards back over might be a single action — where no other player can see or manipulate the intermediate board — or two separate actions. But the board must never reach an inconsistent state, such as two players controlling the same card, or having a face-down card that a player still controls.

Note that a flip operation can *fail* at several places in the rules. In the spec of the `flip()` command (which you will need to implement to connect your `Board` to the provided web server), failure is indicated by a rejected promise, which can be done by throwing an exception from an `async` function. Keep this in mind when you are designing your own `Board` operations. For testing for a rejected promise, `assert.rejects` is useful.

Simulating a game

In `simulation.ts` we have provided skeleton code for a program that simulates one player interacting with a board, randomly flipping over several cards.

You should be able to fill in the blanks in `simulationMain()` function and use `npm run simulation` to run simulated single-player Memory Scramble games.

Problem 2: Connect to the web server

In order to connect your `Board` type to the provided web server, and also to the Didit autograder, you need to implement the functions in `commands.ts` so that they use operations of `Board`. Start with these two functions, which are sufficient to play the game:

- `look()` from `commands.ts`
- `flip()` from `commands.ts`

At this point, your `look()` and `flip()` may not yet implement correct waiting behavior according to the rules or be able to handle concurrent players, but they should be able to handle **multiple clients that make moves in sequence**.

In general, keep all the functions in `commands.ts` extremely simple. They should be no more than *glue code*, calling at most a couple of your `Board` operations, without any additional logic. Specifically:

- The body of each function should be at most 3 lines long, consisting of at most a couple simple function calls, possibly using local variables, plus a `return` statement.
- All of the functions that they call should be already well-tested, either by your `Board` test suite or by TypeScript itself.

If that's the case, you don't need to write a separate test suite for `commands.ts`. Your existing tests for `Board` should suffice.

If any function in `commands.ts` is more complex than that — for example, with control statements like `if` or `while`, or string-processing operations — then you will need to create `test/commands.test.ts` and design a test suite for it. Before you do that, **stop!** First discuss your design with course staff, because this is a sign that the operations you chose for your `Board` type are not powerful enough.

- **Spec**. The specs for `look()` and `flip()` are given, but you may strengthen them if you want.
- **Test**. Keep these operations extremely simple so you don't have to test them separately.
- **Code**. Implement `look()` and `flip()` as simply as possible.

Running the web server

To start the server, use `npm start`, which runs `main()` in `server.ts`. This code parses command-line arguments, calls the `Board` parsing function based on those arguments, and creates and starts a server. You should not need to change `server.ts`.

To run `main()`, first open a terminal, either inside Visual Studio Code or in a separate terminal window. Then start a web server on port 8080 with our 3×3 board of rainbows and unicorns:

```
npm start 8080 boards/perfect.txt
```

Remember that the server keeps running until you terminate its process, and if you try to run a second server on the same port, it will fail with "address already in use". If you run the server on the command line, use `Ctrl`-`C` in that terminal to stop it.

Once your server is complete, you can play the game in your web browser!

In your web browser, go to the URL that the server displays when it starts, and it will return a web page that speaks the game protocol with your server.

You can open the URL in multiple tabs to pretend to be different players. Each time you open the page, it randomly generates a new player ID to use with your server.

You can also connect to a friend's server by IP address, but you may not be able to do it directly as a URL in your browser (because your browser may require connecting by HTTPS, which this server does not support). Instead, find the web page as a file in your own ps4 code (its name is `public/index.html`), open that HTML file in your web browser, and specify the other server by IP address.

Problem 3: Revise for concurrent players

Revise `Board` to make it asynchronous, updating other ADTs as needed.

Consider using `await` on a promise returned by the board to implement **waiting for a card** according to the rules. Refer to "making and keeping a promise" in *Mutual Exclusion* for discussion of:

- the `Promise.withResolvers()` factory function, which is most useful for transforming callbacks into promises
- the specification and implementation of `Deferred`, which is most useful for storing promises to be resolved later

In `simulation.ts`, increase the value of `players` to simulate multi-player Memory Scramble games. By instrumenting your code, for example with `console.log()` statements or by keeping track of cards that different players flip over, you should observe correct *control* of cards, correct *waiting* when players try to flip over a first card, and correct *removal* of matching cards when players make a next move.

Notice that your simulation should not terminate if one player turns over matching cards in their last move (leaving two cards face up under that player's control) and another player tries to flip one of them as their first card.

Consider revising your testing strategy to test different interleavings of concurrent player actions. You may find the `timeout()` function useful for forcing short waits. Very short timeouts (1 or even 0 milliseconds) give up control long enough for concurrent actions to run to completion. Avoid waiting for hundreds of milliseconds, because this may cause your tests to timeout on Didit. **Do not put `timeout()` calls in your implementation code**; this is a sign that your design has a race condition.

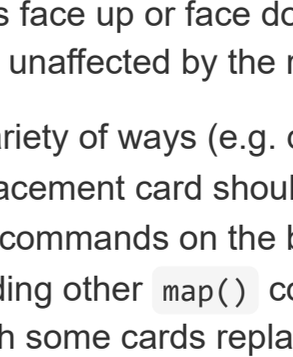
Review your `commands.ts` file to make sure it is using the asynchronous `Board` properly, according to the rules. Your commands should not busy-wait or hang. While one request is waiting (e.g. for a card to flip), other requests must be handled normally.

Problem 4: Changing cards

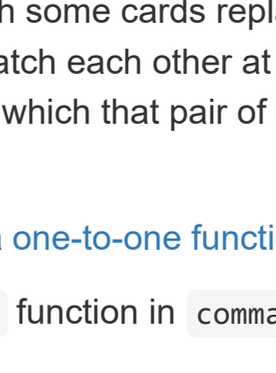
This problem adds one more command to the protocol, `map()`.

The `map()` command applies a transformer function `f` to every card on the board, consistently replacing it with `f(card)`.

For example, this board of rainbows and unicorns (with two cards already matched and removed):



…might be converted by `map()` into this board of sunshine and lollipops:



The `map()` command is independent of other state of the game. So it doesn't matter whether a card is face up or face down, or whether it is controlled by another player, it can still be replaced, and its face-up/face-down and control state is unaffected by the replacement.

The transformer function `f` is asynchronous (returning a promise), so it might be implemented in a variety of ways (e.g. doing a database lookup, accessing a language translation web API, even asking a human user what the replacement card should be). This means that `map()` may have to wait for its calls to `f` to complete. But `map()` must not prevent other commands on the board from interleaving with it while it is running (including other commands issued by the same player, and including other `map()` commands). Depending on how you implement `map()`, other operations may see partially-transformed boards (with some cards replaced but not others), but the board must remain consistent to players in the sense that if two cards on the board match each other at the start of a `map()`, then while that `map()` is in progress, it must not cause any player to observe a board state in which that pair of cards do not match.

The function `f` must be a mathematical function as described in the `map()` spec, but it need not be a one-to-one function.

Specify, test, and implement support in your board ADT that will enable you to implement the `map()` function in `commands.ts` as simple glue code that requires no testing of its own.

The `map()` command is not used by the Memory Scramble web server or web client, but it can be invoked by Didit autograding.

Problem 5: Board events

This problem adds one more command to the protocol, `watch()`.

This command does not immediately return. Instead, it *waits* until the next time the board changes. At that point, it responds with the current state of the board.

A **change** is defined as any cards turning face up or face down, being removed from the board, or changing from one string to a different string.

A `watch()` call must wait until such a change occurs. When there is no change in the state of the cards — for example, if a player tries to turn over an empty space, or if a player takes control or relinquishes control of a card but it does not turn face up or down — pending watch requests must continue to wait.

While a `watch(board, player)` request is waiting, the player may use other commands on the same board, and they will be processed normally. For example, a concurrent `look(board, player)` request must not wait.

Specify, test, and implement support in your board ADT that will enable you to implement watching the board for changes. Your board ADT might implement notification of only a single change, or notification of all future changes:

- If the client receives only a single change, it might be an asynchronous method whose promise fulfills when the board changes.
- Listeners might provide a callback function to `Board`. The board will call them back on each change.

First try using your board's change notifications in `simulation.ts` to watch the board concurrently during a simulated game.

Then use your change notifications to **specify, test, and implement** the `watch()` function in `commands.ts` as simple glue code that requires no testing of its own.

Note that you decide **which changes are atomic**. For example, the `watch()` specification does not say whether, when a pair of matched (or mismatched) cards is removed (or turned face down), that is reported as a single change or two separate changes.

Once you have added `watch()` to your server, you can flip the switch in the Memory Scramble web user interface from "update by polling" to "update by watching", and you should find that the UI is quicker to display board updates made by other clients.

Before you're done

- Make sure you have **documented specifications**, in the form of properly-formatted TypeDoc comments, for all your types and operations.
- Make sure you have **documented abstraction functions and representation invariants**, in the form of a comment near the field declarations, for all your ADTs.

  In addition, **document how the type prevents rep exposure**. See *Documenting the AF, RI, & SRE*.
- When you implement waiting behavior, make sure you are using a promise, not busy-waiting.
- Make sure you specify, test, and implement `equalValue()` for all immutable types.
- Make sure to implement `toString()`. (And if you want to be sure you implemented it right, then *strengthen its spec* and *test* it as well.)

  Also use `@inheritdoc` when a class implements an interface method, to remind readers where they can find the spec.
- Make sure you have a thorough, principled **test suite** for every type.

This is the last problem set in 6.102: your work should exemplify **SFB**, **ETU**, **RFC** software construction!

Submitting

Make sure you commit AND push your work by 10:00pm on the deadline date.

This problem set can take much longer for Didit to process than previous psets, so push early.

Remember that Didit feedback is provided on a best-effort basis:

- There is no guarantee that Didit tests will run within any particular timeframe, or at all. If you push code close to the deadline, the large number of submissions will slow the turnaround time before your code is examined.
- If you commit and push right before the deadline, it's okay if the Didit build finishes after the deadline, because the commit-and-push was on time.
- Passing the public test on Didit is no guarantee that you will pass the full battery of autograding tests, but failing it is sure to mean many lost points on the problem set.

Grading

Your overall ps4 grade will be computed as approximately:
~40% alpha autograde (including online exercises) + ~10% alpha manual grade + ~35% beta autograde + ~15% beta manual grade

The autograder test cases will not change from the alpha to the beta, but their point values will. In order to encourage incremental development, alpha autograding will focus primarily on situations where clients make moves in series (separated by time), not concurrently. On the beta, autograding will look at a full range of concurrent situations.

Manual grading of the alpha will only examine internal documentation (AF, RI, testing strategy, etc.) and implementation that does not relate to concurrency. Manual grading of the beta may examine any part, including re-examining those docs or code, and how you addressed code review feedback.