Homework #1

CR🤪ZY 8's

Overview

Description

Create a demo of an interactive card game, Crazy Eights (https://en.wikipedia.org/wiki/Crazy_Eights), that allows two turns to be played (one "player" turn and one computer turn).

- the game will use a standard 52-card deck of French suited playing cards (https://en.wikipedia.org/wiki/Standard_52-card_deck)
- a configuration file will be used to set the cards remaining the in the draw pile, the player's hand and the computer's hand
- alternatively, without a configuration, the game will generate a deck of cards, shuffle, and deal five cards to each player

- a single card is drawn from the draw pile - it is used as the **starter**
- the **starter** card dictates the suit or rank that should be played next
    - for example, if the **starter** is 2♦
    - …then the next card to played must have either a rank of 2 or a suit of ♦
- the player and computer alternate turns discarding a single card from their hand that matches either the suit or rank of the **starter**
- a card with a rank of 8 can be placed at any time regardless of the rank and suit of the **starter**
- if an 8 is played, then the suit of the **starter** can be set
    - for example, if the **starter** is 2♦
    - …and an 8 is played
    - the suit can be set to any of the four suits: ♠ ❤️ ♣ ♦
- if a card cannot be played, cards must be drawn from the draw pile until a card *can* be played (otherwise pass)
- the first to discard all of the cards in their hand wins!

👀 **Note however that** the demo will only have two turns: the player first, and the computer second. The computer's turn logic will be limited.

You'll create this game in 2 parts:

1. create functions for managing a standard 52-card deck (you may use these functions in the next part, but you are not required to do so)
2. create an interactive demo of the first two turns of a game of Crazy Eights

See the example animated gif at the top of these instructions to see how the demo may work.

Objectives

1. Write some JavaScript!
    - control structures
    - functions
    - Object, Array, and string manipulation
2. Learn how to run node programs
3. Learn about node built-ins:
    - `process`
    - the `fs` module
    - `import` and `export`
4. Install and use ES Modules; create your own
5. Run unit tests to check your work
6. Use a static analysis tool (eslint) to help prevent bugs / errors

Submission Process

The final version of your assignment should be in GitHub.

- **push** your changes to the homework repository on GitHub
- all repositories will be cloned on the due date
- after the due date, no further commits will be seen by the graders (even if you continue to commit your work)

Preparation

Ensure that node and npm are installed (this should have been done for *homework #0*). You should be able to open up your terminal or DOS Shell and run `node -v` and `npm -v`. Both commands should output a version number (⚠️ at least `16.9.x` for `node` is required for tests to run).

1. use git / clone the repository
2. install development modules
    - mocha and chai for running the supplied unit tests
    - eslint for cleaning up your JavaScript / spotting common sources of bugs and errors in your code
3. install modules required by your game

Use Git / Clone the repository

Make sure you have git / a git client (**do not use the web interface to edit / upload files**)!

- the easiest way to work with a git repository hosted on GitHub is by using their official graphical client (https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/getting-started-with-github-desktop)
- commandline git (https://git-scm.com/book/en/v1/Getting-Started-Installing-Git)
    - note: **this is dfferent from the GitHub CLI tool**, which is GitHub's official commandline client
    - for a quick demo of using commandline git with a personal access token as your password, see this video (https://nyu.zoom.us/rec/share/K_8lawLxmB-Byc4cC7FADshZBd60e8IHWmF33RHe-xX9RcRT2dIOeYq_MDt3kOTb.wF0T_NwFFKDJVIg5?startTime=1694787035000)
- a git plugin / extension for your editor (for example, VS Code (https://code.visualstudio.com/docs/editor/versioncontrol#_git-support))
- SourceTree (https://www.sourcetreeapp.com/)

👀 Click on this link: https://classroom.github.com/a/hKerDL_X (https://classroom.github.com/a/hKerDL_X) to accept this assignment in GitHub classroom. This will:

- add you as a member to the course GitHub organization if you are not already a member
- create a repository containing some tests and blank starter files

You can then go through the following steps to clone your repository and commit your first changes:

1. once you've accepted the assignment, you may need to refresh the page before it gives you a link to your repository
2. click on that link to see your assignment repo on GitHub (it should be called `homeworkNN-yourusername` where `NN` is the homework number and `yourusername` is your GitHub username
3. on the repository's page, use the green "Clone or download" button on the right side of the screen to copy the HTTPS clone URL to **clone** the homework. To use the commandline client (with GITHUB_REPOSITORY_URL being the url you copied from the green button):
    ```
    git clone GITHUB_REPOSITORY_URL
    ```
4. create a file called `.gitignore` in the same directory
5. add the following line to the file so that git ignores any locally installed node modules: `node_modules`
6. in the same project directory, create a file called README.md, and edit it so that it includes:
    - your github username
    - the title of your project: Homework #01 - Crazy Eights
7. again, in the same project directory, run `git add README.md` to let git know that we're ready to "save"
8. save your work locally by running `git commit -m "add homework meta information"` ... everything within the quotes after `-m` is any commit message you'd like

    - please make your commit messages descriptive
    - (what features have been added, what bug has been fixed, etc.)
9. finally, send your work to github by running `git push` (or `git push origin master`)

Install Development Modules

You'll have to install a couple of node modules to help you run tests and use static analysis tools on your code. These tools won't be required for your program to run, but they will be useful while you're writing your programs.

You'll be installing the following modules globally:

- `mocha` - for running unit tests

You'll also install the following modules locally in your project directory:

- `chai` - supplies assertions for unit tests
- `eslint` - for catching potential errors in your code
- `eslint-plugin-mocha` - to support linting test code

Go into the directory of your cloned repository (`cd username-homework01`), and create `package.json`

```
npm init
```

Follow the prompts to create a `package.json` ... you can use the default answers to the questions that are asked (for our purposes, most of the prompts are not relevant)

You can add dependencies to `package.json` using the following commands:

```
npm install --save-dev mocha
npm install --save-dev eslint
npm install --save-dev chai
npm install --save-dev eslint-plugin-mocha
# can also be installed in a single line
npm install --save-dev eslint mocha chai eslint-plugin-mocha
```

Note that these will all be installed locally to your project directory:

- It will create a `package-lock.json` file that stores exact versions
- It will make a modification to an existing file, `package.json` , within your project folder
- It will create a `node_modules` folder where your downloaded modules are stored (this folder is included in your `.gitignore` file because these external libraries are not meant to be in your project's version control

Troubleshooting errors

- On some systems (for example, Ubuntu 18.04), you may have to use `sudo` to run `npm` as the super user
- If any part of your code complains about a missing package, try installing it with `npm` either locally or globally (do your best to keep everything local at first, of course)

Install Required Modules

You'll also need a few modules to help you ask the user for input, clear the screen and determine the space that string takes up on on screen. Note that in recent versions of node you can omit `--save` before the package / modules name, as it's the default behavior.

- `readline-sync`

    - in your repository directory, install the node module, `readline-sync`, by running this command in your project's directory: <pre> `npm install --save readline-sync` </pre>
    - note that the `readline-sync` module allows you to prompt for user input **synchronously**
        - this is very different from how node.js apps usually operate
        - however, for our purposes, using sync prompt is fine (for now), and it mimics the browser's prompt functionality well
    - check out the example usage on readline-sync's npm page (https://www.npmjs.com/package/readline-sync) * essentially: `const readlineSync = require('readline-sync');` * which imports the function `question` from the `readline-sync` module into your program
    - note that installing `readline-sync` will make a modification to `package.json` as well. This modification to `package.json` should be committed and pushed as well!
- `clear`
    - in your repository directory, install the node module, `clear`, by running this command in your project's directory: <pre> `npm install --save clear` </pre>
    - the `clear` module allows you to "clear" the screen for a command line program (across different platforms)
    - note that in Windows 11, cmd.exe … there may be artifacts / leftover output (this will be okay)
        - an optional alternative is not use `clear`, but use `console.log('\x1Bc');` instead
    - check out the example usage on clear's npm page (https://www.npmjs.com/package/clear)
        - essentially: `import clear from 'clear';
        - which imports the function `clear` which can be used by simply calling it: `clear()`
    - installing `clear` will make a modification to `package.json`; again, it should be committed and pushed along with your other files

Minimum Number of Commits

As you write your code, make sure that you make at least four commits total (more commits are better; if you can, try to commit per feature added).

- the commits should be meaningful (that is, do not just add a newline, commit and push to make up the requirements for commits).
- make sure your commit messages describe the changes in the commit; for example:
    - `add config file reader and set board based on config file`
    - `fix bug that prevented vertical lines of tiles from being flipped`

```
git add --all
git commit -m 'your commit message'
```

- push your code frequently

```
git push
```

Running Your Programs, Linting, and Testing

To run your programs, use the commandline (Terminal.app, DOS, etc.).

```
# in your project directory
# assuming the file to run is myfile.js located in a folder called bin
node bin/myfile.mjs
```

Use a utility called `npx` to run any executable tools that you've installed, like your linting tool (`eslint`) and your testing tool (`mocha`).

`npx` is included with `npm`, which, on some platforms is automatically installed with `node`. It's used to find the binary of the argument passed to it within your project's `node_modules` so that you don't have to execute it from `node_modules/package_name/bin/executable`.

After installing `eslint` and `mocha`:

```
npx eslint bin/*.mjs
npx eslint lib/*.mjs
```

Part 1 - `cards` Module

Background

Start by creating several functions that may be helpful in implementing a card game. These functions are described below. Unit tests have been included in your repository in the file, `test/test-cards.mjs`.

Creating a Module / Exporting Functions

You'll be creating a module that contains a several functions. The file that you'll be writing your module in is already included in your repository in `lib/cards.mjs`. Both your *actual* interactive game (in Part 2) and the supplied unit tests will use this module. Note that we **are using ES6 modules** (that is, you will use `import` and `export`).

To make the functions you write available when your module is brought into another program, you can either:

1. prefix your function with `export`
   ```
   export const foo = (x, y) => {
       // implementation
   };
   ```
2. export an object containing your functionscreate all of your functions in an object and assign that object to `module.exports`:
   ```
   const foo = (x, y) => {
       // implementation
   };
   const bar = (a, b) => {
       // implementation
   };
   export {
       foo
       bar
   };
   ```
3. When you `import` your module, you must use a path relative to the file that you're writing. Also note that only the "exported" functions will be available. In the example below:
   - `bin/app.mjs` brings in exports from `lib/my-modules.mjs`
   - only the functions that are exported in `my-module.mjs` are available (in this case, `foo` should have been exported)

     ```
     // in bin/app.mjs
     import * as myModule from '../lib/my-module.mjs';
     myModule.foo();
     ```

You should make sure your exports are up to date as you implement your functions so that you can run your unit tests as you complete your function implementations.

Unit Tests

You can use the supplied unit tests (in `test/test-cards.mjs`) to check that your functions are:

- are named correctly
  - have the required parameters
- return the appropriate value(s)
- meet the minimum requirements according to the specifications

The given unit tests use Mocha as a testing framework and Chai for assertions. While you don't have to know how to write these tests, you should read through them (the api is very *human readable*) to get a feel for how your functions are being tested. If you're curious about writing unit tests, check out this article on codementor (https://www.codementor.io/nodejs/tutorial/unit-testing-nodejs-tdd-mocha-sinon).

You can **run the included unit tests by using this command** in your project directory:

```
npx mocha test/test-cards.mjs
```

If you run these tests before starting, you'll get a bunch of reference errors. This is because you have no functions implemented yet. Additionally, you'll have to export the functions you create so that the tests have access to them.

Please try continually running the unit tests as you develop your program.

⚠⚠⚠Only Running a Subset of Unit Tests

To clear out the noise of dealing with many failing tests due to unimplemented functions, use `.only()`:

Any time that you see the functions `describe()` or `it()` in `test/test-cards.mjs`, you can follow it with `.only()` to limit the tests being run to those contained in the call to `describe` or `it`. For example:

```
# describe('generateBoard', function() { });
describe.only('generateBoard', function() { });
# or
# it('generates a board with specified number of rows and columns', function() { });
it.only('generates a board with specified number of rows and columns', function() { });
```

Alternatively, you can simply comment out unused tests until you're ready to implement them.

Assumptions

The functions make some assumptions about how you'll be representing a deck of cards.

1. each card should be represented by an object containing two properties, a `rank` and a `suit`
   - the **suit** can be one of the following `String` values:
     - `['♠', '❤', '♣', '♦']`
     - (note that these emoji will be in the source code for `cards.mjs`)
   - the **rank** can be any of the following `String` values:
     - `2` through `10` (as strings)
     - `['J', 'Q', 'K', 'A']`
   - an example of a card:

     ```
     {suit: '❤', rank: '3'}
     ```

2. a deck of cards, the player's hand or the computer's hand… can all be represented by an `Array` of card objects:

```
[
 {suit: '❤', rank: '3'}
 {suit: '❤', rank: 'A'}
 {suit: '♣', rank: '7'}
]
```

3. all of the functions treat a deck of cards as an `Array`
    - the end of an `Array` of cards is the "top" of the deck
    - when a card is "drawn" from a deck, it's taken from the end of the `Array`
4. ⚠️ many of the functions will opt to return a new `Array` of cards rather than modify an existing `Array`
    - a quick way to make a copy is to use the spread operator:

      ```
      const oldCards = [
       {suit: '❤', rank: '3'}
       {suit: '❤', rank: 'A'}
       {suit: '♣', rank: '7'}
      ];
      const newCards = [...oldCards];
      ```

    - alternatively, `Array.prototype.slice` works as well

      ```
      const newCards = oldCards.slice();
      ```

5. you can assume that all of the code examples below use `cards` as the name of the imported module of your `cards.mjs` functions

Functions to Implement

`range`

Parameters:

- `start` (optional) - the start of the range of numbers, inclusive
- `end` - the end of the range of numbers, exclusive
- `inc` (optional) - the amount to increment by

Returns:

An `Array` containing the range of `Number` values specified

Description:

Create an `Array` of numbers that starts at `start`, ends before `end` and increments by `inc` (similar to Python's `range`). The number and position of the arguments will determine what the `start`, `end` and `inc` values are:

- 1 argument: specifies only `end` (exclusive)
    - `start` defaults to `0`
    - `inc` defaults to `1`
- 2 arguemnts: specifies `start` (inclusive) and `end` (exclusive) in that order
    - `inc` defaults to `1`
- 3 arguemnts: specifies `start` (inclusive), `end` (exclusive), and `inc` in that order

Hint: although there are default values for `start` and `inc`, do not implement these as default values in the function signature. Instead, handle this as a function that can take an arbitrary number of arguments, and use the length to determine default values.

Example:

```
const r1 = range(3);        // r1 is [0, 1, 2]
const r2 = range(2, 6);     // r2  is [2, 3, 4, 5]
const r3 = range(1, 10, 3); // r3 is [1, 4, 7]
```

`generateDeck`

Parameters:

None

Returns:

An `Array` consisting of 52 card objects.

Description:

The array of card objects should adhere to the specifications described in the assumptions section above.

Example:

```
const deck = generateDeck(); // deck has length of 52, with each element being a card object
```

`shuffle`

Parameters:

- `deck` - an `Array` of card objects

Returns:

A new `Array` consisting of all cards from the original `Array` passed in, but shuffled

Description:

Shuffles the deck. Choice of algorithm is your discretion (remember to comment / add link if referencing a shuffle algorithm implementation found online). This function should:

- create a new shuffled version of the `Array` of cards passed in
- ⚠️ **without** modifying the original `Array` (that is, make a copy and return the modified copy)

Example:

```
const deck = generateDeck();

// note shuffled is a new Array, and the original deck passed in remains unchanged
const shuffled = shuffle(deck); ,
```

`draw`

Parameters:

- `cardsArray` - an `Array` of card objects
- `n` (optional, default value: 1) - the number of cards to draw

Returns:

An `Array` consisting of two elements (both sub arrays):

1. a new `Array` consisting of the cards in `cardArray` with the last `n` elements removed
2. a new `Array` consisting of the removed elements

Description:

Removes `n` cards (default to 1) from the deck and returns both the new deck and cards removed.

- ⚠️ the original `Array` of cards passed in is **not** modified.
- the returned `Array` will always have the new version of the cards array as the first element, and the cards drawn as the second element

Example:

```
const originalDeck = cards.generateDeck();

// originalDeck will be unchanged
// newDeck will be original deck with one less card
// drawnCards will be an Array consisting of one element, the last element from originalDeck
const [newDeck, drawnCards] = draw(originalDeck);

// alternatively, a second argument can be passed in
// this specifies how many cards to draw
const [newDeck2, drawnCards2] = draw(originalDeck, 2);
```

`deal`

Parameters:

- `cardsArray` - an `Array` of card objects
- `numHands` (optional, default value: 2) - the number of hands of cards to create
- `cardsPerHand` (optional, default value: 5) - the number of cards per hand

Returns:

An object consisting of two properties, both with an `Array` as a vlue:

1. `deck` : a new `Array` consisting of the cards in `cardArray` with the last `numHands * numcardsPerHand` removed
2. `hands` a new `Array` consisting of `numHands` sub arrays of card objects

For example, dealing 2 hands with 2 cards per hand might result in this object:

```
{
        deck: [ (copy of original array of cards with last 4 cards removed) ]
        hands: [
                [{suit: '❤', rank: '3'}, {suit: '❤', rank: '4'}],
                [{suit: '❤', rank: '5'}, {suit: '❤', rank: '6'}]
        ]
}
```

Description:

"Deals" out `cardsPerHand` number of cards into `numHands` number of separate card `Array`s.

- ⚠️ the original cards `Array` is not modified, and instead a new `Array` is return in the `deck` property.!
- the dealing strategy is your discretion, but cards must be removed from the **end** of the card array (think of the end of the `Array` as the top of a deck of cards)

Example:

```
const originalDeck = generateDeck();


const {deck, hands} = deal(originalDeck); // note that hands is an Array
const [hand1, hand2] = hands; // each hand is an Array of cards

// originalDeck is not modified!
```

`handToString`

Parameters:

- `hand` - an `Array` of card objects
- `sep` (optional, default value: `'  '` … two spaces) - the `String` to place between cards
- `numbers` (optional, default value: `false`) - a `Boolean` specifying whether or not to include the position of the card in the `Array` starting at index 1

Returns:

A `String` representing the `Array` of card objects

Description:

Produces a string representation of the cards object `Array`, `hand`, passed in.

- each individual card will concatenate the values of `rank` and `suit` in that order: `3♥`
- uses `sep` to "join" separate cards
- prefixes each card with its position (starting at 1), colon, and space if `numbers` is `true`: `1: 3♥`

Example:

```
const hand = [
        {suit: '♥', rank: '3'},
        {suit: '♥', rank: '4'},
        {suit: '♣'', rank: 'K'}
];

// default to two spaces as sep, no numbers
const s1 = handToString(hand);      // s1 is 3♥  4♥  K♣

// use pipe as sep
const s2 = handToString(hand, '|'); // s2 is 3♥|4♥|K♣

// using newline as a separator, and turning on "numbers"
import os from 'os';
const s3 = handToString(hand, os.EOL, true);
/*
s3 is...
1: 3♥
2: 4♥
3: K♣
*/
```

`matchesAnyProperty`

Parameters:

- `obj` - an object to test if it contains any of the keys and values of another object

- `matchObj` - the object with keys and values to test against

Returns:

A `Boolean`, `true` or `false` specifying whether or not `obj` contains **any** of the keys and values in `matchObj`

Description:

Will check all keys in `obj`. If **any** of of the keys in object match a key in `matchObj` **and** have the same value, return `true`. Otherwise, return `false`.

Example:

```
const obj = {a: 1, b: 2, c: 3};
const search = {x: 100, y: 200, b: 2};
const res1 = matchesAnyProperty(obj, search); // res1 is true
const res2 = matchesAnyProperty(obj, {a: 90, b: 91}); // res2 is false
```

`drawUntilPlayable`

Parameters:

- `deck` - an `Array` of cards to draw from
- `matchObject` - an object with `rank` and/or `suit` to search for

Returns:

An `Array` consisting of two elements (both sub arrays):

1. a new `Array` consisting of the cards in `deck` with some number of elements removed from the end
2. a new `Array` consisting of the removed elements

Description:

Finds a card matching the `rank` or `suit` of the `matchObj` **or** a card with a `rank` of `8`, starting from the end of the `deck` `Array` and going backwards.

Return an `Array` containing two sub arrays without changing the original `deck` `Array` (that is, use copies):

1. all cards from the beginning of the `deck` up to, but not including the found card
2. all cards from the found card to the end of the `deck`

Think of this as continually drawing cards until an `8` is drawn or a card with the same `rank` or `suit` as the `matchObj`.

If no match is found, the first element of the return `Array` will be an empty `Array`, and the second will contain the same elements as the original `deck` in reverse order.

Example:

```
const deck = [
  {suit: '♠'', rank: '3'},
  {suit: '♦'', rank: 'J'},
  {suit: '♠'', rank: 'J'},
  {suit: '♦'', rank: 'Q'}
];

const observed = cards.drawUntilPlayable(deck, matchObj);
const search = {suit: '♠', rank: '9'};

const [newDeck, drawnCards] = drawUntilPlayable(deck, search);
// newDeck should be:
// [
//   {suit: '♠', rank: '3'},
//   {suit: '♦', rank: 'J'}
// ]
// drawnCards should be:
// [
//       {suit: '♦', CLUBS, rank: 'Q'},
//   {suit: '♠', rank: 'J'}
// ]
```

Part 2 - Two Turns of Crazy Eights

Whew. That was a lot of work. But, ummmm… there's no game yet. What? **Let's use the module / helper functions you created in part 1 to implement an interactive game that supports the following features:**

1. Predefined cards for: the deck, player hand, and computer hand
2. An interactive game

You don't have to use *all* of the function you created in your `libcards.mjs` module (and you definitely won't need to use all of the features of each function). However, you'll likely end up doing a lot of redundant work. Of course, you are encouraged to create your own additional functions as well!

You MUST IMPLEMENT #1, Predefined Cards

- in order for this game to be tested properly, scripted moves must be implemented for both the computer and the human player
- consequently, this feature will be worth a **significant** number of points
- make sure you've implemented it (see the section below, "Predefined Cards")

Prep

You'll write your Crazy Eights game in the file called `bin/game.mjs`. Your first step is to bring in some required modules. Open up `bin/game.mjs` and…

1. bring in the `cards` module you created by using `import`
2. bring in the `question` function from the module, `readline-sync`, which you installed in the preparation portion of the homework

   ```
   import {question} from 'readline-sync';
   ```

3. bring in the `clear` module function from the module, `clear`, which you installed in the preparation portion of the homework

   ```
   import clear from 'clear';
   ```

4. bring in the built-in `readFile` function:

```
import {readFile} from 'fs';
```

Predefined Cards

To facilitate manual testing, add a feature to your game that allows you to pass in the game's starting deck, player hand, computer hand, discard pile, and next card to play.,

To pick up commandline arguments, use the built-in variable `process.argv`. `process.argv` is an array that contains parts of the command used to run your program. For example, if you run your program with `node my-program.mjs`, `process.argv[0]` will be `node` and `process.argv[1]` will be `my-program.mjs`.

We can use this feature to supply the name of a JSON file containing information to seed the deck

- `node filename.mjs path/to/file.json`
- …where the json file will have keys and values that look like this:

```
{
        "deck": [
        {"suit": "♠", "rank": "J"},
        {"suit": "♦", "rank": "A"},
        {"suit": "❤", "rank": "A"},
        {"suit": "♣", "rank": "J"}
      ],
        "playerHand": [
        {"suit": "❤", "rank": "9"},
        {"suit": "❤", "rank": "10"},
        {"suit": "♠", "rank": "3"}
        ],
        "computerHand": [
        {"suit": "♣", "rank": "8"},
        {"suit": "♦", "rank": "5"},
        {"suit": "♦", "rank": "4"}
        ],
        "discardPile": [{"suit": "♦", "rank": "2"}],
        "nextPlay":{"suit": "♦", "rank": "2"}
}
```

⚠️ No Predefined Cards

If a json file is not supplied:

- generate a deck
- shuffle the deck
- deal 2 hands of 5 cards
- create a discard pile
- draw a card from the deck to set the "starter" card
    - continue to draw if the card has `rank` `8`
    - set the next card to be played as this "starter" card

An Interactive Game

Now… for the actual game. Our demo will only implement one turn for the player and one turn for the computer

1. Display the state of the game

```
             CR🤪ZY 8's
------------------------------------------------
Next suit/rank to play: ➡  2❤  ⬅️
------------------------------------------------
Top of discard pile: 2❤
Number of cards left in deck: 4
------------------------------------------------
🤖✋ (computer hand): 8♣  5♦  4♦
😊✋ (player hand): 9❤  10❤  3♠
------------------------------------------------
```

2. If the player has a playable card, offer a menu so that they can play a card:

```
😊 Player's turn...
Enter the number of the card you would like to play
1: 9❤
2: 10❤
3: 3♠
```

3. If the player does not have a playable card, automatically draw cards and make the play for the player

```
😊 Player's turn...
😔 You have no playable cards
Press ENTER to draw cards until matching: 2, ♦, 8
.
Cards drawn: J♣  A❤  A♦
Card played: A♦
Press ENTER to continue
```

4. If the player plays an 8 (either by drawing cards or by explicitly playing an `8`), prompt the player to choose a suit:

```
CRAZY EIGHTS! You played an 8 - choose a suit
1: ♠
2: ❤
3: ♣
4: ♦
>
```

5. Clear the screen once the player's turn is done
6. Redisplay the game state
7. Allow the computer to make a play:
    - you can choose any strategy to choose a card to play (first found, random, etc.)
    - if an 8 is played, you can choose any strategy to choose a suit
    - the computer should be able to handle having to draw cards to make a play
8. Clear the screen
9. Redisplay the game state

(Optional) More than two turns

If it's easier for you to implement more than two turns, it's okay to implement a more complete game

(Optional) Note about tests

Depending on your implementation, there may be completely different objects created, but all with same values. For example:

```
{rank: 1} === {rank: 1}
// false
```

The example above shows two objects with the same value, but equality is false.

A similar issue may arise in some tests. You can modify the tests to use `deep` if some of these tests are failing (due to comparison of objects). For example, you can change something like this:

```
expect(shuffledDeck).to.have.all.members(originalDeck);
```

…to this:

```
expect(shuffledDeck).to.have.deep.members(originalDeck);
```

…to ensure "deep" comparison (checking object keys and values, rather than references).