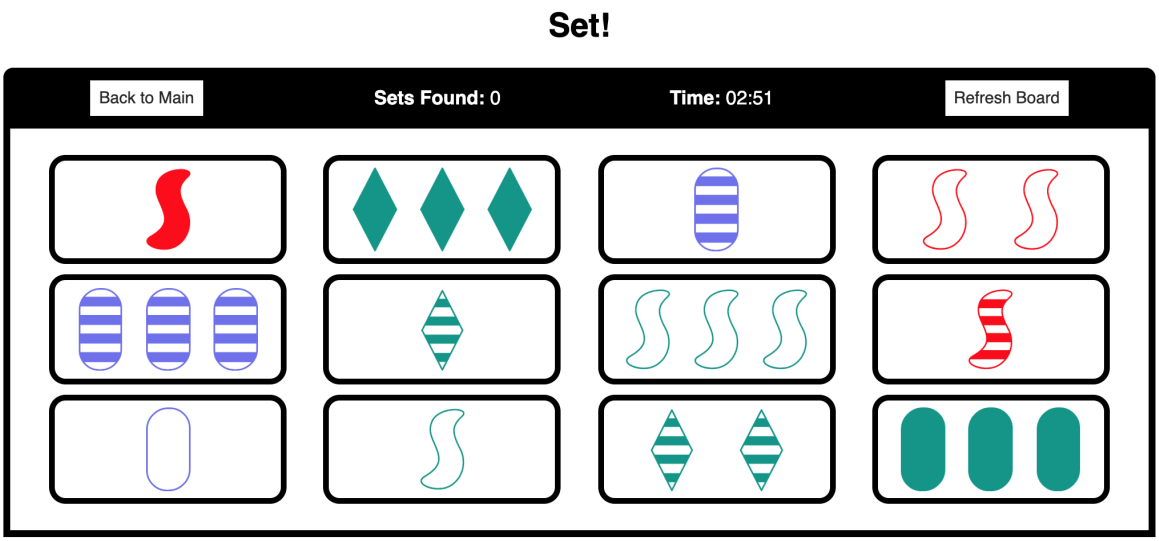
Homework 2: Set! Part B

Due: Friday, May 7th 2AM PST (morning)



HW2 Part B Overview

In Part B, you will use JS to add functionality to the Set webpage with responsive CSS to layout the two different views. Specifically, you will finish implementing the web-based version of the Set card game, providing features to generate new games for a user and to keep track of how many correct Sets a player has found.

You will also implement a basic timer for the user to keep track of how quickly it takes them to solve a given game. You can find a video demonstrating expected behavior for different cases here.

The rules of Set described in Part A are provided again here for reference.

Set! Overview

This assignment extends what we've learned in Module 1 (HTML/CSS) and puts into practice what you're learning in Module 2 (JavaScript, DOM, and Animations) to build an interactive webpage! A fully-functional UI (user interface) takes more time to plan and implement than just an HTML/CSS webpage (such as the one you created in HW1). As such, this HW is broken into two parts, based on the course schedule and the general process web developers take when implementing an interactive webpage like this one.

As a two-part assignment, it will be worth more points than HW1 and consists of two separate submissions:

1. Part A: Game View (CSS) and View-Switching (Start JS) - 10 points
2. Part B: Implementing the Game UI (JS DOM Manipulation and Timers) - 20 points

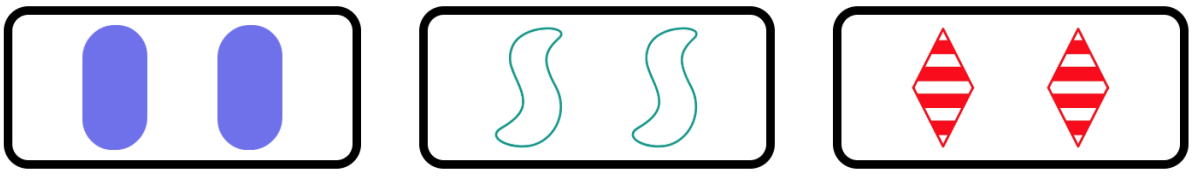
Note the point breakdown. You will have more time to work on Part B (do not treat Part A as "50%" of the work - Part B will take more time). The requirements for Part A are shorter, and will be due earlier so you can get feedback to implement in Part B.

Rules of Set

This assignment is inspired by the classic SET! Card game. A game consists of a board of cards (you will implement boards having 9 and 12 cards, and milestone.html is provided to test your styles.css with 9 cards). Each card has one of three options for 4 different "attributes":

Attribute		**Options**	
STYLE	solid	outline	striped
COLOR	green	purple	red
SHAPE	diamond	oval	squiggle
COUNT	1	2	3

The goal of the game is to find as many "Sets" of 3 cards such that for each attribute, all cards share the attribute or no cards share the attribute. For example, the following three cards build a Set because none share style, color, or shape attributes but they all share the count attribute.



However, the following three cards do not form a Set since the color attribute does not follow the "all or none" requirement (purple is shared by the first and third card, but not the second).

In Part B, you will be dynamically generating the board with JavaScript and use event listeners to support user interaction. You will also learn how to implement an increasing/decreasing game timer on this page, as well as how to show/hide feedback to users with a 1 second delay.

Starter Files and Final Deliverables

You will have the required `imgs`, `styles.css`, and `set.js` files from Part A. However, you will also need `set.html` for Part B (instead of the `milestone.html`). Note that `set.html` is a little different than `milestone.html` - it factors out the cards from HTML to dynamically generate them using JS based on selected options and gameplay. It also removes the "Unlimited" `<option>` tag from the Main Menu's `<select>` dropdown. You will just be implementing timed mode (5, 3, and 1 minute games). This is a helpful approach when working on a more complex UI and is similar to the approach where we started with the hard-coded `skittles-jar.html` and generalized it with skittles.js in Week 4's lectures.

In Part B (final submission) you must not change the provided `set.html` or any of the images, but you will be submitting your changes to `styles.css` and `set.js` to meet this specification's requirements.

File/folders	Provided files to stay unchanged
`set.html`	(**New to Part B**) Provided HTML linking to your `styles.css` and `set.js` files
`imgs`	A folder with 27 classic set cards, each named with the convention: *STYLE-SHAPE-COLOR.png* (replacing STYLE, SHAPE, COLOR with a value for that attribute as listed in "Rules of Set" section)
`styles.css`	(**Both Part A and Part B**) Stylesheet for `set.html`

With the exception of `milestone.html`, which will not be graded with the final turn in, your submission should include these starter files as well as the following file you are to implement:

File	Files you will implement
`set.js`	JS file for managing game UI and behavior for completed Set implementation.

Your final solution will be graded only on `styles.css` and `set.js` - any changes you make to `set.html` or the card images folder will not be eligible for full credit.

Part B Requirements

The final submission for this assignment requires that you fully implement the user interaction and game management for the Set! game. To be eligible for full credit on this assignment your submission:

- Must meet the expected behavior as outlined below and as demoed in the provided video
- Implement the 7 required functions described in the spec (including the `toggleView` function you implemented in Part A)

Beyond proper implementation, you are expected to implement the feedback from Part A before turning in your completed project. You will receive the feedback prior to the due date of this assignment and are responsible for correcting and clarifying anything noted by your grader. If you do not implement the feedback you will not be eligible for full credit on the assignment. Remember that this is your chance to iterate on early feedback to demonstrate an understanding of the material we've covered so far.

Notes:

- **IMPORTANT:** The defined functions described below must be implemented as described in the spec. You are expected to refactor sections of code into *smaller*, helper functions that are called by the defined functions. This makes your code significantly easier to read and reduces redundancy across your JS file.
- These functions are **not** exhaustive meaning that you are *still* expected to write the code to tie these functions together and connect them to the user interface. This will definitely require writing additional functions beyond what is outlined in the spec. Think of these defined functions as the building blocks of the game. It is your task to implement the fundamentals and connect the pieces.

Note about difficulty and generated cards: In both Easy (9 card) and Standard (12 card) difficulties, there are 4 attributes (style, color, shape, and count) and 3 possible options for any attribute (e.g. red, green, or purple options for the color attribute). For Standard difficulty, each card should have a randomly-selected value for each of the 4 attributes. For Easy difficulty, the "style" attribute should be fixed to "solid". To avoid redundancy in your program and allow for flexibility if users later want to change the images and attributes, **we suggest using 4 arrays as**

module-global constants, one for each attribute type (style, color, shape, count) - since count is an integer, you may alternatively use a different datatype to represent it as a constant**.** Remember that a module-global constant is any that is in the module pattern but defined outside of the local scope of functions within the module pattern (in general, you should minimize the number of module-global variables).

Behavior Details Overview

Below are the details describing the game Set. This description, the provided video, and the description of the required functions should provide you with all the information necessary to implement the game.

Starting a Game

- The current set count should be 0.
- A timer should be started depending on the currently-selected option in the dropdown. The game timer should start with the number of seconds determined by the selected option and decrement each second (never going below 0 seconds).
- The Main Menu view should be hidden.
- The Game View should be displayed (after the first two steps so that the initial game information is shown correctly when the view changes).
- A new randomly-generated board of cards is generated, where each of the 4 attributes of a card are randomly-chosen (recall each attribute has 3 options) in Standard difficulty. The only difference in the randomness of cards in Easy difficulty is that the "style" attribute should always be fixed to "solid".
- The generated board should never have duplicate cards (cards that share all 4 attributes). During this step, you should not create more than 12 card DOM elements (9 for Easy).
- The Refresh Board button should be reenabled (in case it was disabled in a previous game)

Game Play

Upon initialization a user should be able to select and deselect from the cards on the board. When three cards have been selected there will be a message displayed to the user indicating whether the three cards make a Set or not:

- If the cards make a Set, the message "SET!" will be briefly displayed and the cards will be replaced with three, new, unique cards.
- If the cards do not make a Set, the message "Not a Set :(" will be briefly displayed and the same cards will be re-displayed on the board (re-displaying should not involve re-creating the cards).

Refreshing the Board

When the "Refresh Board" button is clicked:

- All cards on the board should be replaced with a new collection of cards, following the same rules for generating a board outlined in "Starting a Game".

Ending the Game

A game ends when the user clicks the "Back to Main" button or when they run out of time. In both cases, the game timer should be stopped and not started again until a new game is started by the user (when the "Start" button is clicked again).

If the game ends due to running out of time:

- The current view should remain on the Game View until the "Back to Main" button is clicked.
- Any cards currently selected should appear unselected.
- To ensure a user cannot continue playing until a new game is started, nothing should happen anymore when a user clicks on a card.
- The "Refresh Board" button should be disabled until it is re-enabled if a user starts a new game later.
- The timer calling `advanceTimer()` should be stopped and cleared.

When the "Back to Main" button is clicked:

- The current view should be switched from the Game View to the Menu View having the same selected options for difficulty as the previous game.

Module Global Variables for Part B

For full credit, you may only have the following module-global variables in `set.js` (other module-global program *constants* are fine):

- `timerId` - keeps track of the timer variable that keeps track of game length in seconds
- `secondsRemaining` - represents the time in seconds left in the current game

You may optionally have a `totalCards` module-global variable to represent the total number of cards managed on the board for the current game. This isn't required, but there are reasonable trade-offs depending on different program implementations.

Required Functions For Part B

In addition to the `toggleView` function you were required to implement in Part A, there are 6 required functions in Part B. We have provided a table for a summary, but more details are provided below the table for each function.

As a reminder, you are expected to break down your program into smaller functions as necessary, but these will be used to test your code and will need to work with the different interactive elements and events using event listeners appropriately.

Function Name/Arguments	Brief Description/Return Value
`generateRandomAttributes(isEasy)`	Returns a randomly-generated array of string attributes in the form [STYLE, SHAPE, COLOR, COUNT].
`generateUniqueCard(isEasy)`	Return a `div` element with COUNT number of `img` elements appended as children.
`isASet(selectedCards)`	Returns true if a given list of 3 cards comprises a Set.
`startTimer()`	Starts the timer for a new game. No return value.
`advanceTimer()`	Updates the game timer (module-global and `#time` shown on page) by 1 second. No return value.
`cardSelected()`	Used when a card is selected, checking how many cards are currently selected. If 3 cards are selected, uses `isASet` to handle "correct" and "incorrect" cases. No return value.

`generateRandomAttributes(isEasy)`

Accepted parameters:

- `isEasy` {boolean} : if true, the style attribute (1 of the 4 card attributes) always be “solid”, otherwise the style attribute should be randomly selected (“outline”, “solid”, or “striped”)

Each card has four different attributes - style, shape, color and count.

- Use `Math.random()` to randomly generate values for each of these four different attributes. Store each attribute value in an array in the exact order shown here: `[STYLE, SHAPE, COLOR, COUNT]`.

- Implementation suggestion: focus on generating these attribute combinations one at a time to reduce redundancy.

Expected return:

- Returns a randomly generated array of attributes in the form `[STYLE, SHAPE, COLOR, COUNT]`

`generateUniqueCard(isEasy)`

Accepted parameters:

- `isEasy` {boolean} : if true, the style of card will always be solid, otherwise each of the three possible styles is equally likely.

Expected returns:

- Return a `div` element with COUNT number of `img` elements appended as children
  - `img` elements should have a `src` attribute following the appropriate image naming convention (i.e STYLE-SHAPE-COLOR) to reference the correct image inside of the `imgs/` directory.
  - `img` elements should also have `alt` text that stores the attributes of the form: STYLE-SHAPE-COLOR-COUNT
- For keeping track of unique cards, you are required to give your card `div`s an ID with the 4 attributes in the form: STYLE-SHAPE-COLOR-COUNT.
  - `generateUniqueCard()` should **only** generate cards that are **not** currently inside of `#board`, meaning they do not have the same ID.
  - **IMPORTANT**: You cannot make any assumptions about the current state of the game and cards previously put on the board, you should only use the information currently inside of `#board` and nothing you have kept track of previously.
- The card `div` should have the class `.card` added to it in order to style the cards.
- The card should have a click event listener attached to it that calls the `cardSelected` function defined below in the spec.

`isASet(selected)`

This function takes 3 card `div` elements, compares each of the attribute types and, if each four attributes for each card are either all the same or all different, then returns true. Note: The sample solution has 8-12 lines excluding { and } and depending on local variables factored out.

Accepted parameters:

- `selected` {DOMList}: A DOM list of 3 properly generated card `div` elements that are selected.
  - These cards should be generated from `generateUniqueCard(isEasy)` described above

Expected returns:

- Returns true if all the given cards are a Set, otherwise returns false

`startTimer()`

- Called when a game starts.
- Accesses the timing option selected from the `#menu-view` and sets the timer to display that time and updates the state of the game to keep track of the current time.
- Starts the periodic calling of `advanceTimer()` every 1 second (see specification for function below)

`advanceTimer()`

- With each call of this function, the time should be decremented by 1 second.
- This should both update the time kept track in the game and update the `#time` on the board
    - **Note about the game timer format**: When updating during the Game View, the timer format in `#time` must stay in proper MM:SS format (for example, if there are 8 seconds left, the timer should display 00:08).
- If there is no time left in the game (time reaches or goes below 0), the board should be disabled as described in the Ending the Game above:
    - The current view should remain on the Game View until the "Back to Main" button is clicked.
    - Any cards currently selected should appear unselected.
    - To ensure a user cannot continue playing until a new game is started, nothing should happen when a user clicks on a card.
    - The "Refresh Board" button should also be disabled until it is re-enabled if a user starts a new game later.
    - The timer calling `advanceTimer()` should be stopped and cleared.

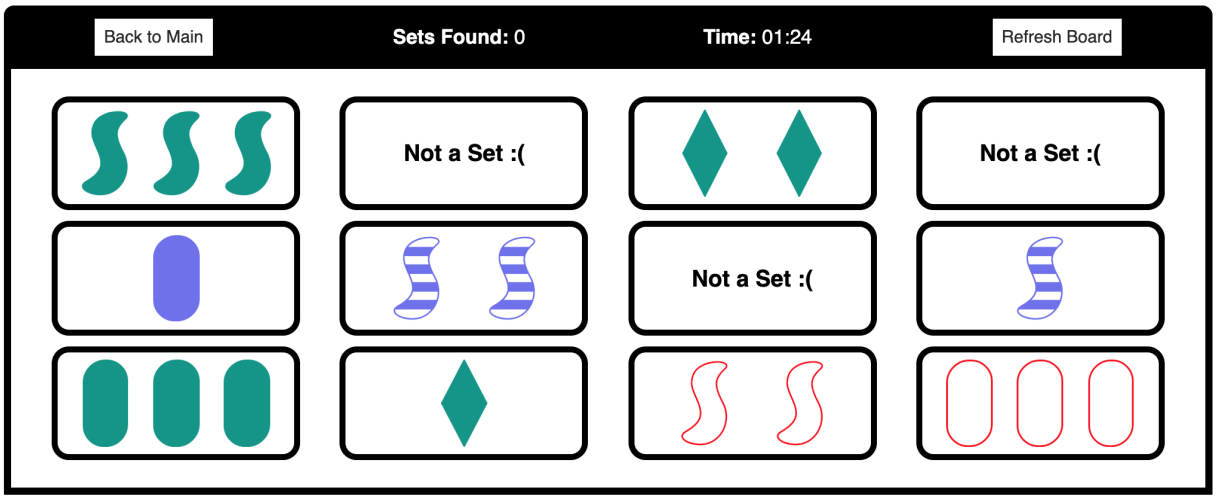
`cardSelected()`

- Once the game has been initialized, a user can click on cards to find a Set. Clicking on a card **when a game is running** should toggle its `.selected` state (all cards are initially unselected). When three cards are `.selected` on the board, there are two possible cases you should handle depending on what `isASet` returns.

Both Cases:

- For both cases, the three selected cards should lose the `.selected` appearance just prior to the appropriate 1-second message displaying (refer to provided video demo linked on the course website).
- **IMPORTANT STYLE ADDITION:** All `p` elements inside of the `.card`'s should be bold with a font size of 16pt. This text should be centered horizontally in the card (you will need to modify your `styles.css` file to accomplish this).

- Below is an example of the message displayed when an incorrect Set is selected:



Case 1: The three selected cards create a Set

- Number of sets found should be incremented (the set counter should be incremented by one in `#set-count`)
- A `<p>` element containing the text message "SET!" should appear in each card and the img elements within the card should be hidden by adding the class `.hide-imgs` to the `.card` itself.
    - **HINT:** You should modify your CSS to include the selector `.hide-imgs img` to hide the images. Make sure to minimize CSS redundancy by not having duplicate rules.
    - Remember that you may **not** modify the provided HTML file but the structure of your card should look like this for 1 second before populating a new randomly generated card to replace the previous one (we recommend setting a breakpoint in the Chrome Inspector Sources tab and check the card's HTML when you expect a card to display the message).

```
<div id="id-of-card" class="card selected hide-imgs">
  <img src="img-src" alt="id-of-card">
  <!-- possibly more images -->
  <p>SET!</p>
</div>
```

- After 1 second, you should replace the selected cards with a new **unique** card that is not currently on the `#board`. For this, you should use the `generateUniqueCard(isEasy)` function that you have already implemented.
    - When replacing cards in a found Set, the order/position of other cards on the board should remain unchanged - refer to the demo video for an example.

Case 2: The three selected cards do not create a Set

- A `<p>` element containing the text message "Not a Set :(" should appear in each card and the `img` elements within the card should be hidden by adding the class `.hide-imgs` to the .card itself
    - **HINT:** You should modify your CSS to include the selector `.hide-imgs img` to hide the images. Make sure to minimize CSS redundancy by not having duplicate rules.
    - Remember that you may **not** modify the provided HTML file but the structure of your card should look like this for 1 second before returning to its original structure (*which should not have the `<p>` element*).

```
<div id="id-of-card" class="card selected hide-imgs">
  <img src="img-src" alt="id-of-card">
  <!-- possibly more images -->
  <p>Not a Set :(</p>
</div>
```

- An invalid Set results in a 15 second deduction from the time remaining
    - **Note**: The time displayed should never be negative meaning that if there are less than 15 seconds remaining the clock should display 00:00 in addition to implementing the appropriate end game behavior.
    - Your time should be updated immediately on the page after clicking the third card and not wait for the `advanceTimer()` function or the message to go away to make it reflect on the page.

Development Strategies

If you're unsure where to start, the following is a roadmap of recommended steps to make the assignment more approachable:

1. Understand the overall behavior of the game to give important context in implementing the game
    - Watch the videos, read the Behavior Details Overview
    - Understand the expected returns of `isASet` for different arguments and when you might use it.
2. Start by implementing the defined functions in the order they're introduced in the spec
    - This should give you the majority of the behavior of the game.
    - Take each function one step at a time and remember to use your other functions to avoid reimplementing shared behavior.
    - When you are implementing the functions, make sure you only add the functionality outlined in the spec.

    - Remember to break down the defined functions into smaller functions to make the problem easier to tackle, read, and understand.
3. Implement the overall connection of the DOM and the defined functions and any other behavior to complete the game.
    - You may find it helpful to create a table similar to the one suggested in the CP2 spec while developing
4. Running into bugs? Use the Chrome console/sources tab as demonstrated in lecture/section, use `console.log` to see what the result of a statement is, add a breakpoint in the sources tab to access information at each step of the function, etc.
5. We will provide tests midway through Part B for you to test your solutions with. Run these provided tests before submitting your work to check the behavior of your solution and make sure you have not missed any important details. Please note that these tests are not exhaustive and you should not solely rely on the tests/assume your code is perfect if all the tests pass.
6. Rewatch the video to ensure you have not missed any details
7. Be proud of what you've done (this is a challenging assignment) and play the game!

External Requirements

For full credit, Your webpage should match the overall appearance of the provided screenshots and it **must** match the appearance specified in this document. We do not expect you to produce a pixel-perfect page that exactly matches the expected output image. However, your page should follow the specific style guidelines specified in this document and match the look, layout, and behavior shown here as closely as possible. Any properties unspecified in this spec or visually discernible in screenshots should be left to the default values for the respective page element (e.g the font size of the `h1` element on the page).

Internal Requirements

For full credit, your page must not only match the External Requirements listed above, you must also demonstrate that you understand what it means to write code following a set of programming standards. Your CSS and JS should also maintain good code quality by following the CS 101 Code Quality Guide. Make sure to review the section specific to JavaScript! We also expect you to implement relevant feedback from previous assignments. Some guidelines that are particularly important to remember for this assignment:

CSS:

- Aside from a few new CSS styles you need to add in Part B (e.g. for the temporary card messages), most of your Part B work will be JS. That said, you should review your feedback from Part A for CSS and part of your grade for Part B will be for CSS.

JavaScript:

- Your .js file must be in the module pattern and run in strict mode by putting `"use strict";` within the module pattern.
- Do not use any global variables, and minimize the use of module-global variables. Do not ever store DOM element objects, such as those returned by the `document.querySelector[All]` functions, as global variables.
    - *Note*: Our solution has the two module-globals described above (the game timer and number of seconds in a current game) and some solutions may also have a module-global to keep track of the count of cards for the current game difficulty. Otherwise, these are the **only** module-global variables you need to declare in this assignment.
- Use camelCase naming conventions for variables and functions
- There should be never be any DOM elements on the page sharing the same ID.
- All images in cards should be given an appropriate `alt` attribute.
- You should not have any unnecessary interval/time-out running on your page at any time (make sure you understand the difference between an interval and delay)
- If a particular literal value is used frequently, declare it as a module-global "constant" IN_UPPER_CASE and use the constant in your code. Our solution has an array for each attribute, a path to the images folder, and two constants for the number of cards associated with each difficulty. Constants make your code significantly more readable and modular, you are encouraged to declare them to eliminate "magic" values in your code.

Both:

- Format your CSS, and JS to be as readable as possible, similarly to the examples from class: Properly use whitespace and indent your CSS and JS code as shown in class. You can find information about JS conventions in the Code Quality Guide.
- You may not use any CSS/JS frameworks for this assignment.
- Your page must use valid CSS3 and JS and successfully pass both the W3C CSS3 validator and JSLint with no errors.

Documentation

Place a comment header **in each file** with your name, section, a brief description of the assignment, and the file's contents (examples have been given on previous assignments).

- You will be expected to properly document your functions in set.js using JSDoc with `@param` and `@return` where necessary. Refer to the Code Quality Guide for some examples.

Optional Challenges

When playing Set, it can be hard to know whether a Set exists on the board before deciding to try a new collection of cards. But each time you reset a board you are more likely to quickly find a new Set! You can earn up to an extra point (max of 20 pts for Part B) for implementing either of the following challenges:

- Adding a check when the "Refresh Board" button is clicked to see whether a valid set exists on the board
- If the button is clicked when a valid Set exists on the board, a 15 second penalty should be applied to the timer (-15 seconds in timed mode, +15 seconds in unlimited time mode) and an alert message "(15 second penalty) there were Sets left on the board!" should appear.

Hint: One possible way to check for a unique set of cards on the board uses the following pseudocode:

```
for every card A on the board:
    for every other card B on the board:
        for every other card C on the board:
            if isASet(A, B, C) return true
return false
```

Debugging Tools

We strongly recommend that you use the Chrome DevTools on this assignment, or use the similar tool built into other browsers. This tool shows syntax errors in your JavaScript code. As we've demonstrated in class, you can use it as a debugger, set breakpoints, type expressions on the Console, and watch variables' values. This is an essential tool for efficient JavaScript programming.

The ES6 JSLint tool can help you find common JavaScript bugs. Since this is your first JavaScript assignment, you will probably encounter tricky bugs. If so, paste your code into JSLint to look for possible errors or warnings. To be eligible for full credit, your JavaScript code must pass the provided JSLint tool with no errors reported ("Warnings" are okay.)

Grading

The grading distribution for Part B will be broken down as follows:

- External Correctness: 10pts
- Internal Correctness: 8pts
- Documentation: 2pts