SlapJack

Specification

Now that you've got the main foundations down to build out a frontend application, it's time to prove to yourself that you own those skills! You're going to be building a SlapJack card game from scratch!

Learning Goals

- Solidify and demonstrate your understanding of:
  - DRY JavaScript
  - localStorage to persist data
  - event delegation to handle similar event listeners
- Understand the difference between the data model and how the data is displayed on the DOM
- Use your problem solving process to break down large problems, solve things step by step, and trust yourself to not rely on an outside "answer" to a logical challenge

Solo Project Expectations

This project is an important step in demonstrating you are ready to start Module 2. To ensure we can accurately assess that, it's important you meet the expectations:

- You are the only one who should type code - no copy-pasting code!
- For any code that you didn't write entirely by yourself (mentor or rock supported), you should be able delete it and re-write it yourself
- The only resources you use are MDN, CSS Tricks, and lesson plans - no youtube videos or tutorials of programming Slap Jack. If you have an opportunity and are tempted, do the right thing for YOUR learning and don't do it!
- You are allowed to watch gameplay of the physical SlapJack game; just avoid coding videos.
- Any peer-to-peer collaboration should be discussions about IDEAS, not coding together or sharing code.

We want to see YOUR work.

Set Up & Submission

- Create a **private** repository and add your assigned instructor as a collaborator. (You cannot deploy private repositories to GitHub Pages; that's ok.)
- Create a project board or other planning gist where you will outline your anticipated progress through functionality
- Begin wireframing your application and planning out the architecture of your HTML (in your notebook is fine)

By EOD on Kick Off Day: DM your assigned instructor with two links: Project Board/Planning Gist & Repo, and include photographs/screenshots of your wireframes

Functionality

Here is a video demonstrating most functionality of the game (the only functionality not explicitly depicted is data persistence using Local Storage):

Player 1's keyboard controls:

- `q` to deal a card
- `f` to slap

Player 2's keyboard controls:

- `p` to deal a card
- `j` to slap

Gameplay:

- Players alternate turns playing cards face-up into the central pile (ex a player can't deal twice in a row)
- Any player can slap at any time, with several outcomes!
  - If a player slaps when a Jack is on top of the central pile, the entire central pile is added to the player's hand, and their hand is shuffled automatically.
  - If a player slaps when a Double or a pair (two cards of the same number - such as two Aces, or two Fives, or two Queens) is on top of the central pile, the entire central pile is added to the player's hand, and their hand is shuffled automatically.

    - If a player slaps when a Sandwich (two cards of the same number - such as two Aces, or two Fives, or two Queens, separated by a different card in the middle) is on top of the central pile, the entire central pile is added to the player's hand, and their hand is shuffled automatically.
    - If a player slaps when neither a Jack, Double, or Sandwich is on top of the central pile, the player who slapped loses the card on top of their hand and it is added to the bottom of their opponent's hand.
- If one player loses all their cards, they have one chance to not lose and continue the game:
    - The player with cards left continues to deal from their hand into the central pile (they are now allowed to deal multiple times in a row!)
    - If the player with cards left deals all their cards into the center without revealing a Jack, the central pile returns to that player's hand, is shuffled, and the player must continue to deal until a Jack is revealed
    - When a Jack is revealed, the player who is out of cards can slap it. The central pile is then their new hand, the game continues as normal.
    - If however, the player who is out of cards slaps something that is not a Jack, or if the player who still has cards slaps the Jack first, then the player who is out of cards loses and the game is over!
    - Doubles and Sandwiches are not valid when one player is completely out of cards - in this case, only a Jack can save them!
    - The only way the player who still has cards can win is by slapping the Jack before the player without cards left does

In this project, we will not be providing detailed iterations. We want you to exercise your skills in planning out work!

Note: No need to match colors or fonts, but the overall layout should be the similar. The card assets are provided.

Architecture

Your entire application will consist of one HTML page or template. You will have three JavaScript files:

1. A `player.js` file that contains a `Player` class.
    - `Player` methods must include, *but are not limited to*:
        1. `constructor` - properties should include: `id`, `wins`, `hand` (an array to track the player's hand of cards)
            1. `playCard`
        2. `saveWinsToStorage`
2. A `game.js` file that contains a Game class.
    - A `Game` should include:
    - Two `Player` instances
    - An array of all the possible cards
    - A way to shuffle the deck
    - A way to keep track of the central pile of cards the players will add to
    - A way to deal the deck out to the players
    - A way to keep track of which player's turn it currently is
    - A way for a player to deal a card into the middle pile
    - A way for players to attempt slapping the pile with varying outcomes ("legal" slaps are Jacks, doubles, and sandwiches - see the playthrough video for further explanation)
    - A way to update a player's wins count
    - A way to reset the deck and players to play a new game when one is won
3. A `main.js` file that instantiates the Game class and also contains all DOM related JavaScript

You will also have:

1. An `assets` folder, which will contain the card images: to be found **here**.

You *may* create additional properties, methods, Classes, or files, if you have a need for them, but beware of adding unnecessary complexity.

Data Model

In a game like SlapJack, it is tempting to manipulate the DOM first. Remember that the game logic exists exclusively in the data model. The DOM simply reflects/displays that data model.

Suggested Iterations

This workflow is not required, but may help you meet the overall requirements of the project.

1. Plan out the HTML layout (colors and fonts do not need to match, but overall layout should closely match the demo video)
2. Create the Player class
3. Create the Game class
4. Make game fully playable without the DOM (manually invoking instances of Game, Players, and running methods, etc, from your console) to force yourself to think data-model-first
5. Create 3 decks on the DOM
6. Connect Game data model to the DOM
7. Display the Player win data on the DOM
8. Automatically reset the game to allow for a new game to be played after the previous game is won
9. Persist Player win data using local storage (number of wins should persist across page refreshes)

Rubric

Rubric score key:

- **4: Over-achievement** - student did self-teaching and accomplished beyond scope of project, *without* sacrificing code quality in the pursuit of extensions; all code demonstrates strong mastery of DRY principles and SRP conventions
- **3: Right on track** - student is exactly where we want them to be, has a strong foundation, and demonstrates competency and comfort with the subject
- **2: A little behind** - student needs to focus study in this area to catch up to where they should be in terms of proficiency
- **1: Very behind** - student needs intervention to get back on track

Overall rubric scores will be **averaged**. Here is what the average score means in terms of completing the module:

- **4:** - Student will complete module if prior project work, attendance, and final assessment corroborate readiness
- **3+:** - Student will complete module if prior project work, attendance, and final assessment corroborate readiness
- **2+:** - Student might complete module if prior project work, attendance, and final assessment corroborate readiness

- **>2:** - Student needs more time with concepts and work covered in module

---

Professionalism

We still expect you to use good workflow practices! Atomic commits, use of branches and PRs (you will be merging these PRs yourself).

- **4:**
  - A PR template was used.
  - A code review was requested and completed by a mentor
  - developer can speak to how the feedback in code review was implemented (and point to the commit(s) that implemented the feedback).
  - Developer is ready to clearly present the evolution of this app during the eval - from architecture decisions made in the planning process, to wins and challenges throughout.
- **3:**
  - Developer is able to clearly answer questions regarding the evolution of this app during the eval - from architecture decisions made in the planning process, to wins and challenges throughout.
  - A project board/planning document is used and updated throughout the project.
  - Branches are used. Most commits are formatted correctly.
  - The README is formatted well and contains:
    - Overview of project and goals
    - Overview of technologies used, challenges, wins, and any other reflections
    - Screenshots/gifs of your app
- **2:**
  - Developer is able to answer questions regarding the evolution of the app, but may need some prompts to clearly articulate or provide enough context.
  - More than a few commits are formatted incorrectly.
  - The README is formatted well but may be lacking in detail.
- **1:**
  - Developer struggles to answer questions regarding the evolution of the app.
  - **OR:** Commit and PR history does not tell a story of the application
  - **OR:** A README has not been created/has minimal information.

JavaScript - Style and Implementation

- **4:**
  - **Code is well refactored** and demonstrates developer empathy.
  - No global variables are used aside from query selectors and an instance of `Game`. If you feel you need more because you are building out additional functionality that requires a global variable, please check in with an instructor.
  - There are no instances of repetitive code
  - No functions are longer than 10 lines
- **3:**
  - The separation of data model logic and presentational logic is clear and can be explained by developer.
  - The application correctly **implements a data model** for the `Player` and `Game` classes, including all required methods.
  - Developer can speak to the **role of each class**.
  - All DOM manipulation is handled exclusively in `main.js`.
  - Developer demonstrates understanding and ability to refactor code by having at least 1 example of DRY code that was intentionally reused.
  - SRP is evidenced by clear, concise function names; each function only does what the name describes.
- **2:**
  - Arguments and parameters are used to limit global variables.
  - The **event object** is used correctly, and is not accepted as a parameter if it is not necessary.
  - Developer can speak to **how the event object is utilized** for any given event handler.
  - **Function and variable names** describe their role in the program.
    - Examples: The name of the data type should not ever be in a variable name (ex: “petArray”); the name itself should be clear enough to indicate the type of data it holds (ex: “allPets”).
  - An event handler should not have “handler” in the name (ex: “clickHandler”); the name should indicate the handler’s purpose (ex: “addNewPet”).
- **1:**
  - Style and syntax meets the criteria of the **Turing JS Style Guide**.
  - **Function declarations** are used over anonymous functions in event listeners.

Functional Expectations

- **4:** Application is fully complete (matches all functionality from demo, without bugs), and implements additional functionality devised by the student
- **3:** Application is fully complete (matches all functionality from demo, and player win counts persist across refreshes, a new game starts automatically)
- **2:** Able to play an entire game successfully (dealing, slaps, win, displayed player wins count updates as needed)
- **1:** Unable to play an entire game successfully

/