**Assignment 9.2 – Final: Card Game**

**Setup**

Each week you will be asked to create a new folder under web-231 following a naming convention of “week-<number>.” If we are on week two, the folder name should be “week-2.” All files associated with the weekly assignment will be added to the appropriate weekly folder. All programs must be linked in the index.html landing page under the “Weekly Assignments” section. Projects will be linked under the “Projects” section of the index.html landing page. To be clear, **all** of the JavaScript, HTML, images, and CSS files associated with a weekly assignment must be placed under the appropriate weekly folder. The page title for all HTML files in this course must say “WEB 231 – Enterprise JavaScript I.” And, all HTML and CSS files must be valid HTML/CSS, tested through the WC3 validator. The links were provided during WEB 200 and were added to the index.html landing page. Also, the blue border around the provided images is to show they are images and should not be included in your submission. In other words, do not add a blue border around your work, unless the instructions explicitly ask for it.

**User interface styling and formatting requirements are located in the HTML, CSS, and JavaScript Requirements document.**

HTML: **<yourLastName>-card-game.html**

CSS: **<yourLastName>-card-game.css**

**Grading Reminders**

1. (50%-points) All code sources (.html, .css, .js) must be cited in the opening programmers’ comments, following the format specified in the code attribution document.
2. (25%-points) All code sources (.html, .css, .js) must show evidence of code comments. This means each section of the program (.html, .css, .js) must include code comments that explains what the block of codes purpose is, what the required parameters are (data type, if any), and what the expected output is.
3. (rubric) All code sources (.html, .css, .js) are measured against
   1. Code functionality: Does it work? Does it meet requirements?
   2. Adherence to standards and conventions. Are you using the appropriate data types, including proper indention, are variables named appropriate (variable x is an example of poor naming conventions), is there an appropriate use of whitespace, is the code organized, and are semicolons being used to terminate code sentences?
   3. Efficiency: Use of language features. Are you practicing DRY (Don’t-Repeat-Yourself?), are you leveraging built-in language features where appropriate, and are you using classes/functions to reduce code clutter?
   4. Documentation: Self-documenting, naming conventions, code is maintainable by others. Is the code your write easy to read and maintainable by others?
   5. Error trapping/handling. Are there errors in the program? Is there evidence of coding best practices to reduce user errors?
   6. Assignment Specific Compliance. Does the delivered solution follow the instructions, as they are written? Does the output match what was provided in the screenshots (including spaces, styling, etc.)?

**Required Modifications**

* Cite any sources in your opening programmer’s comment
* Link the appropriate CSS files and Google fonts
* onload…

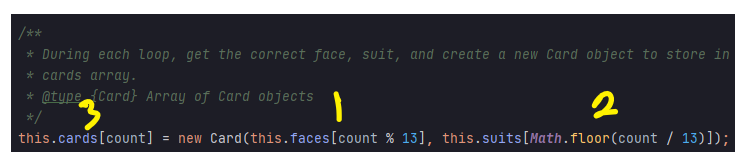
**Additional JavaScript Requirements**

1. Create a **Card** class with a constructor with parameters for **face** and **suit**.
2. Create a **Dealer** class with a constructor with no parameters.

**Additional JavaScript Requirements**

1. Create a variable named **CARD\_COUNT** and assign it a default value of 52.
2. Create a variable named **cards** and assign it a default value of an empty array.
3. Create a variable named **faces** and assign it an array of the faces in a card deck: A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K
4. Create a variable named **suits** as assign it an array of the suits in a card deck: Hearts, Diamonds, Clubs, Spades
5. Create a function named **getDeckOfCards** with no parameters.
   1. Add a for loop with 52 iterations (use the CARD\_COUNT variable for this: **count < this.CARD\_COUNT**).
   2. During each iteration create a new Card object and assign it to the cards array (Exhibit A, #3). To create the Card object, you will need to access the face and suit the faces and suits array. The count variable of the for loop will be used to get the correct suit and face.
   3. Faces expression (Exhibit A, #1), use the modulus operator by 13: **this.faces[count % 13]**
   4. Suits expression (Exhibit A, #2), use Math.floor and divide the count by 13: **this.suits[Math.floor(count / 13)]**

**Exhibit A. Card Assignment**

****

1. Call the **getDeckOfCards** function (b.5.) in the body of the class’s constructor
2. Create a function named **shuffle** with no parameters.
   1. Add a for loop with 52 iterations (use the CARD\_COUNT variable for this).
   2. During each iteration, generate a random number (see the below random card expression) between 0 and 52 and assign it to a variable named **secondCard**. Find the first card in the cards array by using the for loop’s iteration variable (I named mine **firstCard**) and assign it to a variable named **tempCard**. Reorder the cards array by setting the current card to the **secondCard** and the second card to the **tempCard**
   3. Random card expression: **Math.floor(Math.random() \* this.CARD\_COUNT)**

**Additional Programming Assistance**

let tempCard = this.cards[firstCard];

1. Create a function named buildPlayingCard with four parameters: card, suitIcon, faceColor, and suitColor.
   1. This function will return a string with the following div format: card > card-title > card-content > span
   2. Give the card div a CSS class of player-card

**Additional Styling Requirements**

Add a new CSS class to <yourLastName>-card-game.css named **player-card** and center it’s text.

* 1. Using inline CSS styling, style the card-title div by left aligning the text, setting the font size to 20 pixels, setting the left padding to 10 pixels, and setting the color to the parameter **faceColor**.
  2. Using inline CSS styling, style the card content div by setting the font size to 28 pixels, setting bottom padding to 25 pixels.
  3. Using inline CSS styling, style the span div by setting the color to the parameter **suitColor.** Give the span div a CSS class of the parameter **suitIcon**

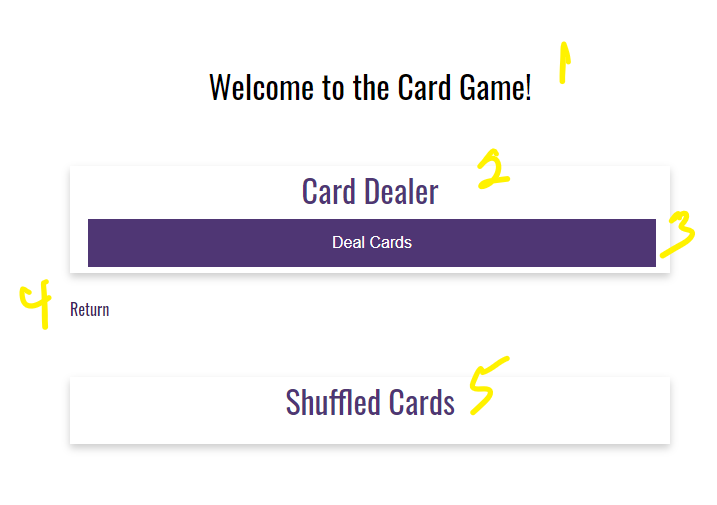
**Additional Styling Requirements**

This program uses Material Design Icons, so you will need to add a reference in the HTML header for this CDN library (refer to the provided resources for the CDN).

**Additional Programming Assistance**

* Icon: **`<span class=”${suitIcon}” style=”color: ${suitColor}></span>`**
* Notice how the above string is using back ticks. The **buildPlayerCard** function will need to use back ticks to return the correctly formatted string.
* To get you started: **`<div class=”card player-card”><div class=”card-title” style=”color: ${faceColor}”>${card.face}</div></div> `**
* Note: the above code is not complete and is only there to help get you on the right path.

**Exhibit B. User Interface (Initial view)**

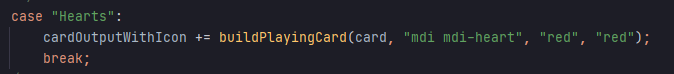
****

1. h1 with a CSS class of app-header and a text value of “Welcome to the Card Game!”
2. card-title with a text value of “Card Dealer.”
3. HTML button with an id of **btnDealCards** and a text value of “Deal Cards.”

**Additional JavaScript Requirements**

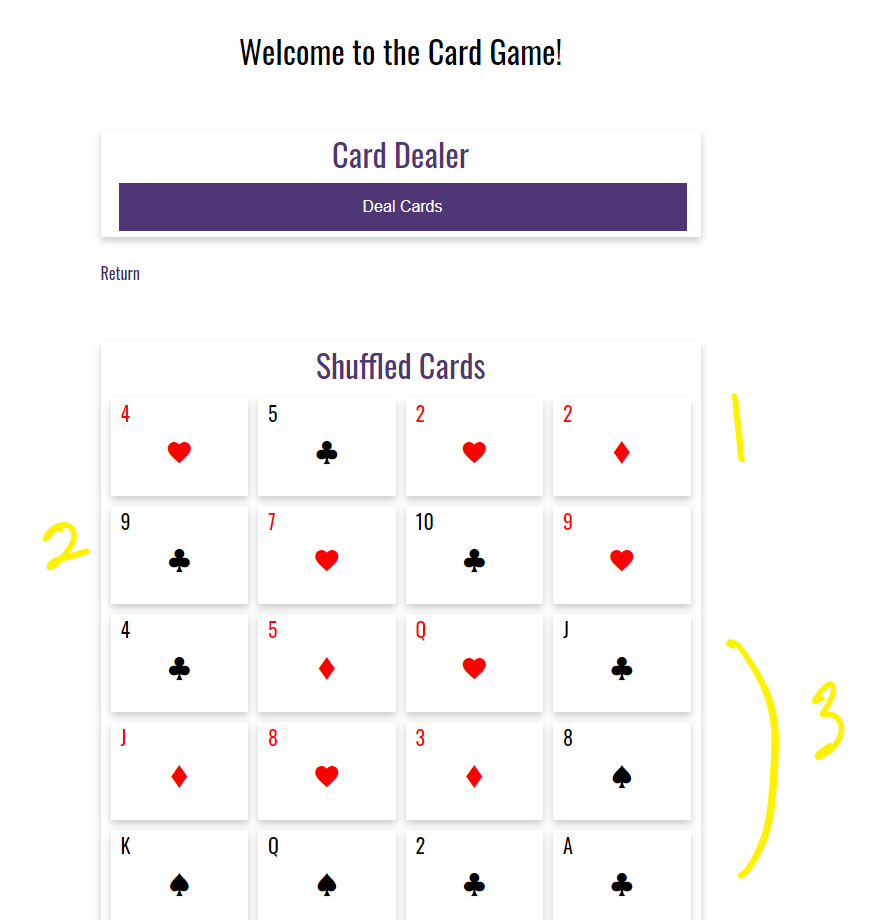
1. Register an onclick event for **btnDealCards**.
2. Create a new Dealer object.
3. Call the shuffle() function.
4. Create a function named
5. Create a string variable named **cardOutputWithIcon** and give it a default value of an empty string.
6. Using a for/of loop, iterate over the cards array (hint: dealer10.cards) and use a switch expression on the suit (hint: card.suit) to compare the suit against the string values: Hearts, Diamonds, Clubs, Spades.
7. For “Hearts,” call the **buildPlayerCard** function passing in the card object, the heart icon class (mdi, mdi-heart), the string value red, and the string value red. Assign the results to **cardOutputWithIcon** string. (Exhibit D, #1).

**Exhibit C. function call for hearts**



1. For “Diamonds,” call the **buildPlayerCard** function passing in the card object, the diamond icon class (mdi, mdi-cards-diamonds), the string value red, and the string value red. Assign the results to the **cardOutputWithIcon** string (Exhibit D, #1).
2. For “Clubs,” call the **buildPlayerCard** function passing in the card object, the club icon class (mdi mid-cards-club), the string value black, and the string value black. Assign the results to the **cardOutputWithIcon** string (Exhibit D, #2).
3. For “Spades,” call the **buildPalyerCard** function passing in the card object, the spade icon class (mdi mid-card-spade), the string value black, and the string value black. Assign the results to the **cardOutputWithIcon** string (Exhibit D, #2).
4. Outside of the body of the for/in bind the cardOutputWithIcon to the player-card-container div.

**Exhibit D. Card Output with Icons**

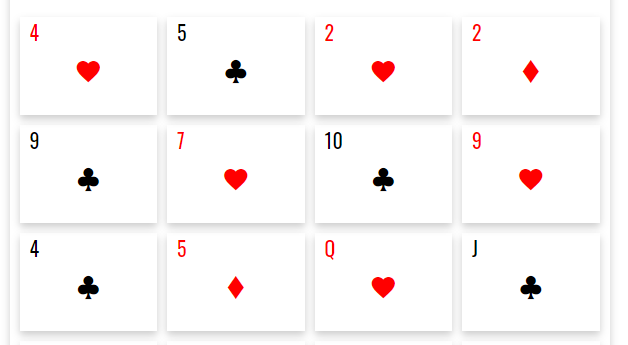
****

1. Notice how the suits Hearts and Diamonds are styled red (Exhibit D, #1).
2. Notice how the suits Club and Spades are styled black (Exhibit D, #2).
3. Notice how there are four cards per row with a 10-pixel gap between each card (Exhibit D, #3).
4. While Exhibit D does not show this, there are 13 rows of 4 columns for a total for 52 cards.

**Additional Styling Requirements**

1. Set the CSS #player-card-container to a display of grid with four columns, a padding of 10 pixels, and a grid gap of 10 pixels.

**Exhibit E. player-card-container**



1. Notice how there are four cards per row and there is a 10-pixel gap between each card. This is the expected styling.
2. anchor link with the CSS class return-home and a link back to the index.html landing page. This page will need to be linked under the “Projects” section of the index.html landing page.
3. card-title with a text value of “Shuffled Cards.” The cards will be outputted in the card-content portion of this div.