Module: Web Systems Engineering **Module Code: ELEE1159**	**Coursework Number:** 1	**Contribution:** 50%
Module Leader: Fazle Chowdhury	**Date Set:** 15 November 2024	**Due Date:** 13 December 2024

Assignment: JavaScript Programming

Plagiarism is presenting somebody else's work as your own. It includes: copying information directly from the Web or books without referencing the material; submitting joint coursework as an individual effort; copying another student's coursework; stealing or buying coursework from someone else and submitting it as your own work. Suspected plagiarism will be investigated and if found to have occurred will be dealt with according to the procedures set down by the University.
All material copied or amended from any source (e.g. internet, books) must be placed in quotation marks and in italics, with a full reference to the source directly underneath the material. **Your work will be submitted for electronic plagiarism checking. Any attempt to bypass our plagiarism detection systems will be treated as a severe Assessment Offence. All requirements and advice are available via links on Moodle**.

Task Overview:

You are required to write the following for this assessment:

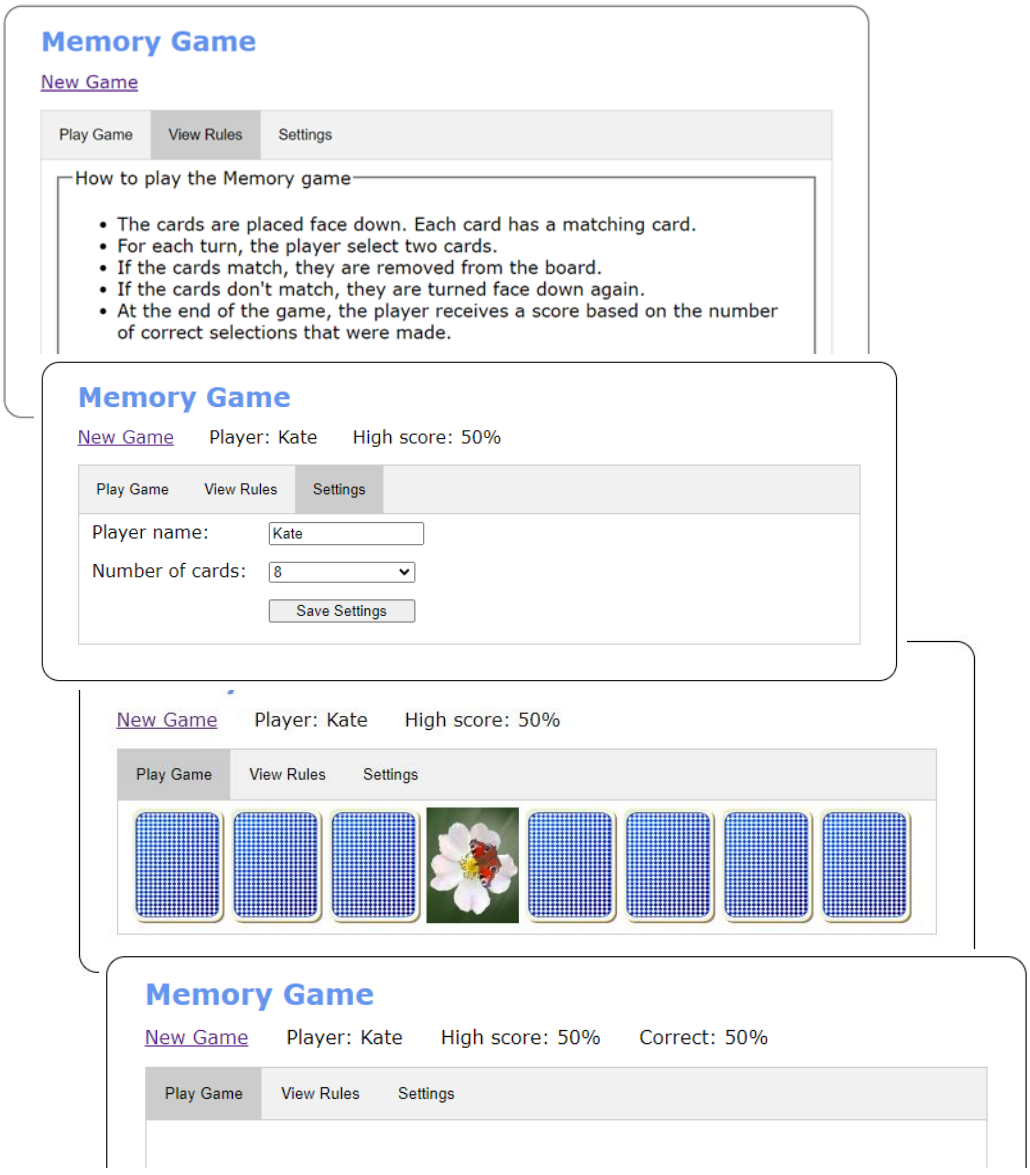
1. Create an app as per the Project Specification given below.
    - All the HTML, CSS, and initial JavaScript and a folder with Images can be found here:

      ELEE1159 Assignment 1
    - To use the files, download the files to your personal area first.
    - Ensure all code files has your banner id and name, along with the course ID as comments at the beginning of the code.

Project Specification:

Create an app that allows the user to play the Memory game. The objective of this game is to turn over two cards at a time, trying to find cards that match until all cards have been matched.

User interface



Specifications

- The main portion of the user interface should be displayed within the three tabs shown above.
- **Images are provided for 24 cards**, the back of the cards, and a blank card. All these images should be preloaded when the app starts.
- By default, the game should display six rows of cards with eight cards in each row, for a total of 48 cards (two of each card). The **user can change the number of cards** used **and enter a player name** in the Settings tab.

- When the user clicks the Save Settings button, the player name and number of cards should be saved in local storage. In addition, the page should reload so the player's name, the player's high score if previous games have been played, and the correct number of cards are displayed. (You can use the reload() method of the location property of the window object to do this.)
- Create the HTML for the displayed cards by randomly selecting from the cards array. The cards for each row should be in a element, and the elements for all the rows should be in the element with an id of "cards".

The Game

- When the user clicks on a card whose back is displayed, it should flip to show the front of the card. The back of the card should fade out and the front of the card should fade in. (You can use the setInterval() method and the style.opacity property of an image to create a fade effect.)
- When the user clicks on a second card whose back is displayed, it should also flip as described above. Then, the second card should display for 1 second. If the cards match, they should flip to a blank image. If the cards don't match, they should flip to show the back of the cards.
- Each time the user completes a game, the user's high score should be updated and displayed. In addition, the percentage of correct selections for the game that was just completed should be calculated and displayed. The high score should also be stored in local storage so it can be compared against the score for the user's next game.
- When the game ends, the user can click the New Game link to start another game.

The HTML for each card

- The HTML for each card should consist of an <a> element with its id set to the src attribute for the card image and its href attribute set to "#". The <a> element should contain an <img> element with its src attribute set to the image for the back of the card and its alt attribute set to "".

2. **Marks will be given as per the Rubric outlined at the end of this document.**

In your coding consider the following:

1. **Variables, Data Types, and Operations:**
2. **Control Statements**
3. **Arrays and Strings**
4. **Functions**

5. **DOM Manipulation**
6. **Testing and Debugging**
7. **Downloading to local storage**
8. **Look and feel of the application and code**

3. Write **A ReadMe instruction document** outlining what the code does and how to run it. This too should be uploaded to Turtitin.

    - The ReadMe instruction document should also have your banner id and name, along with the course ID at the beginning. In addition, Name the file ELEE1159 <Your first name surname banner-id>.

4. All files need to be submitted in a **ZIP file** within Turnitin (in Moodle)
    - The Zip file should include:
        - the HTML code, CSS and JavaScript code and images
        - The ReadMe instruction document
    - Name the zip file - ELEE1159 <Your first name surname banner-id>

IF UNCERTAIN on any aspect, please EMAIL, ALSO, THERE WILL BE SOME TIME DURING THE coming LECTURES TO DISCUSS.

Remember: NO TWO CODE WILL BE THE SAME.

Appendix A: Rubric

	80-100% Exceptional	70-79% Excellent	60-69% Very Good	50-59% Good	40-49% Adequate	30-39% Fail	1-29% Bad Fail	0 Fail
Code Correctness - 30% **The program should run without any syntax errors or runtime crashes.** **The code should produce the correct output for all given test cases, including edge cases (e.g., large inputs, empty inputs, special characters).** **Functions and logic should work as described in the project specifications.** **Any external APIs or libraries used should be integrated correctly, and the data flow should work as intended.** **Input validation should be done to prevent incorrect or invalid data from being processed.**	No syntax errors or runtime crashes. Produces correct output for all test cases, including complex edge cases (e.g., large inputs, special characters). Functions and logic fully meet project specifications. External APIs and libraries are flawlessly integrated, with no issues in data flow. Input validation is thorough and prevents all invalid data, with appropriate user feedback.	Minimal syntax errors, no crashes during execution. Correct output for almost all test cases, with minor issues in some edge cases. Functions are well-structured but could benefit from minor optimization. External APIs work well, though occasional minor delays may occur. Input validation covers most cases, with minor oversights.	A few syntax errors, but the program runs without crashing. Output is correct for most cases, but some edge cases are not handled properly. Functions work as intended, but the logic could be simplified or optimized. APIs and libraries are integrated, though some issues with data flow or performance may exist. Input validation works but is incomplete for some edge cases.	Several syntax errors, no major crashes. Mostly correct output, but struggles with more complex cases. Functions work but are inefficient or lack clarity. APIs/libraries integrated, but there are notable issues or delays. Basic input validation, but significant gaps exist in handling edge cases.	Significant syntax errors, occasional crashes. Handles simple cases but produces incorrect results for many test cases. Functions have noticeable logic errors or inefficiencies. APIs/libraries are unreliable, with frequent issues. Weak input validation; invalid data often slips through without proper handling..	Frequent syntax errors and runtime crashes. Incorrect output for most cases. Functions are incomplete or largely incorrect. APIs/libraries are poorly integrated, leading to failures. Little input validation.	Severe syntax errors, constant crashes. The program fails to produce correct output even for basic test cases. Functions and logic are mostly missing or completely incorrect. APIs/libraries not integrated at all or non-functional. No input validation.	Nothing submitted.
Use of control structures - 15% **The appropriate control structures (if-else, switch, loops such as for, while, do-while, etc.) should be used to implement logic in a clear and efficient manner.** **The project should demonstrate a correct understanding of when and how to use control structures to solve problems (e.g., using loops for iteration instead of repetitive code).** **Nested control structures (e.g., if-else inside a loop) should be handled carefully to avoid unnecessary complexity or confusion.**	Excellent and efficient use of if-else, switch, and loops to implement logic. Appropriate nested control structures without unnecessary complexity. Clear and well-optimized logic.	Control structures used efficiently, with minimal redundancy. Some minor improvements needed for nested logic	Appropriate use of multiple control structures, though some inefficiencies in logic (e.g., overly complex nesting).	Basic control structures used correctly, but logic is inefficient in some parts. Some unnecessary complexity in loops or conditionals.	Inconsistent use of control structures, with significant inefficiencies or confusion in the logic. Overuse or underuse of conditionals and loops.	Control structures are misused or incomplete. Poor logic implementation leading to incorrect outputs.	No proper use of control structures.	Nothing submitted.
Functionality - 20% **The project should fulfill all the requirements as outlined in the project brief.** **All features should be implemented correctly and be fully functional.** **The program should be modular, meaning that functions should be used appropriately to break down tasks. Repeated code should be avoided by creating reusable functions.** **Error handling should be incorporated where appropriate, ensuring that the program can gracefully handle unexpected situations (e.g., network errors, invalid inputs).**	All features are fully implemented, meeting or exceeding project requirements. Code is modular, with reusable functions and minimal repetition. Handles all error cases gracefully.	Meets almost all project requirements, with minor omissions. Modular, though some code could be further reused.	Most features are implemented, though some are incomplete or missing. Code is somewhat modular but has room for improvement.	Basic functionality is achieved, but several features are missing or not fully implemented. Code is repetitive.	Partial functionality; many requirements are unfulfilled. Modularisation is poor, with significant code repetition.	Major features missing, and the program does not meet most requirements.	Almost no functionality; the program is non-functional.	Nothing submitted.

	80-100% Exceptional	70-79% Excellent	60-69% Very Good	50-59% Good	40-49% Adequate	30-39% Fail	1-29% Bad Fail	0 Fail
Code Readability and Comments - 15% **The code should be well-organized, with meaningful variable and function names that describe their purpose. Avoid vague names like x, y, or data without context.** **The code should be properly indented, with consistent spacing and formatting throughout.** **Comments should be used to explain complex parts of the code. However, avoid over-commenting (e.g., comments explaining obvious code).** **Functions should be concise and focused on one task. Large functions should be broken down into smaller, reusable ones.**	Code is well-structured and consistently formatted. Variable names are meaningful and well-chosen. Comments are concise and help explain complex logic.	Code is clear and readable, with minimal formatting inconsistencies. Comments are present but could be expanded.	Code is readable, but there are some issues with formatting and clarity. Comments are basic but insufficient.	Code is somewhat readable, but there are issues with formatting and poorly named variables. Comments are sparse or incomplete.	Unclear code, with poor formatting and missing or vague comments.	Disorganized code, with no comments and unclear variable names.	No attention to readability. Code is chaotic and lacks any comments.	Nothing submitted.
Error Handling, Testing and Debugging - 15% **Descriptive error messages should be provided to help users understand what went wrong and how to fix the issue.** **Unit tests carried out for key functions to verify that they work as expected. Tests should cover both typical and edge cases.** **Debugging tools like console.log() or browser developer tools should be used effectively to track down and resolve issues.** **Students should demonstrate that they can predict possible errors and handle them gracefully (e.g., handling null values, handling incorrect user inputs).**	Descriptive error messages and complete unit tests covering all edge cases. Effective use of debugging tools to ensure no runtime issues.	Most errors are handled, with good test coverage for edge cases. Debugging is mostly effective, with few minor issues.	Basic error handling, but a few cases are missed. Testing covers most cases, but not all edge cases.	Basic error handling, but some issues go unhandled. Testing is incomplete, with many missing cases.	Minimal error handling and debugging. Little test coverage, leading to undetected errors.	No meaningful error handling or testing.	Little, if any error handling or testing. Code is untested and riddled with errors.	Nothing submitted.
Documentation - 5% **A well-structured README file or project documentation should be provided. This documentation should explain how to set up and run the project, what dependencies are needed, and how to use the key features.** **Instructions for any setup or installation should be clear (e.g., how to install or adda library function, how to run the program locally).** **The documentation should also include any known issues, limitations, or future enhancements.**	Detailed README with clear setup instructions and full project documentation. All features and dependencies are explained, with future enhancements listed.	Comprehensive README, though some minor improvements needed in explanation.	Basic documentation, but missing some details or clarity.	Minimal documentation, with vague or missing instructions.	Documentation is present but incomplete.	Documentation is present but with inadequate instructions.	Documentation is present but with instructions incorrect.	No documentation provided.

END OF SPECIFICATION.