IT211 Homework 1
v1.1
Score: _____/10

The purpose of this lab is to get experience with asynchronous programming in JavaScript.

Clone the assignment repository and create a directory inside of it called “workspace”. Create a Node.js project in the “workspace” directory using PhpStorm. When PhpStorm is open and the project is created, create a file called “guessingGame.mjs”. If you get a message about adding the file to Git, check the box that says “Don’t ask again” and click “Cancel”.

Asynchronous programming involves executing tasks without blocking the main execution thread. This is often done using functions like `setTimeout()`, `setInterval()`, or making use of promises, async/await, and callbacks. Asynchronous operations allow JavaScript programs to perform tasks such as fetching data from a server, handling user interactions, or performing animations without freezing the user interface. We’ll get a lot of experience with it in this class, but we’ll start with something simple by programming a guessing game.

1. Copy the `randomInteger()` function from the Random JavaScript page on Canvas. Paste it into the guessingGame.mjs file.
    a. You can copy and paste the comments above the function as well. The comments serve as documentation for that function.
2. To get input from the user, you have to import the `readline` module. Add this line to the top of your file:
    ```
    import readline from 'readline';
    ```
    a. Reading from the console is an asynchronous function. Therefore, the code we need to write must be nested in an asynchronous function.
3. The readline module requires that you setup an interface which defines the input and output streams. Add the following code outside of your:
    ```
    const rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout
    });
    ```
4. Define a function named `guessingGame()` after the interface you created in the last instruction. Make sure to use the `async` keyword before the `function` keyword to make the function asynchronous.
5. For the first instruction of this new function, generate the random number the user has to guess between 1 and 100. Store it as a constant.
6. Create another variable called `guessesRemaining` and initialize it to 5.
    a. The value of this variable will change, so which keyword should be used when defining this variable: `let` or `const`? (**Never `var`**)

7. After that line, finish the guessing game logic but without the inputs. You can put placeholder lines where you want the inputs to go. Test with hard-coded numbers. Ensure you have user-friendly prompts and echos.
    a. Ensure that you're calling your `guessingGame()` function. Do this after the function has been declared, in the global scope (which means outside of any other functions).
        i. Ignore any warnings the IDE gives you where you call your function. We will address that later.
    b. Remember to use `console.log()` to print to the console.
    c. Try to use string interpolation at least once!
8. Notice that your program doesn't actually end, even if you've programmed everything properly! This is because your input interface (which we haven't used yet) is still open. To resolve this, add the following line **after** you call `guessingGame()`:

    ```
    rl.close()
    ```

9. Now that you're ready for user input, add the following code where you want to take guesses from the user:

    ```
    rl.question('Please enter your guess: ', (answer) => {
        guess = parseInt(answer);
    });
    ```

    a. Note: `guess` can be replaced by whatever variable you are using to hold user's guesses.

According to the docs, "The `rl.question()` method displays the query by writing it to the output, waits for user input to be provided on input, then invokes the `callbackfunction` passing the provided input as the first argument." However, is this really true? Run your code to see what happens. (Scroll up in your output if needed.)

It never actually waits! `rl.question()` is an asynchronous function, meaning it doesn't block the execution of your code. When you call `rl.question()`, it starts the process of waiting for user input, but your code continues to execute without waiting for the user to actually input anything.

10. Change your code to this:

    ```
    guess = await new Promise((resolve) => {
        rl.question('Please enter your guess: ', (answer) => {
            resolve(parseInt(answer));
        });
    });
    ```

When you run your code, does it wait now? No! But this is caused by a different issue. Read the warning your IDE gives on the line where you call your `guessingGame()` function. What is a Promise?

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. It allows you to handle asynchronous operations more easily, especially when dealing with nested callbacks.

In simpler terms, think of a promise like a placeholder for a value that hasn't been computed yet, but will be at some point in the future. You can attach functions to promises to be executed when the promise is resolved (successfully completed) or rejected (encountered an error).

You're creating a new promise using `new Promise((resolve) => {...})`. This means you're saying "I'm going to do something asynchronous, and when it's done, I'll let you know."

A Promise object in JavaScript is like ordering a meal at a restaurant.

- Order placed (Promise created): When you place your order, the restaurant staff promises to deliver your meal. At this point, you don't have your food yet, but you're guaranteed that it's being prepared.
- Waiting (Pending state): While you wait, the kitchen is working on your order. You don’t know exactly when it will be ready, but you trust that it’s coming.
- Meal served (Promise resolved): When your meal arrives, the promise is fulfilled. You can now enjoy your food, just like how a resolved promise in code delivers the expected result.
- Order can't be fulfilled (Promise rejected): If the restaurant runs out of an ingredient and can't make your dish, they inform you that they can't complete the order. This is like a promise being rejected, where the expected outcome can’t be delivered.

```
guess = await new Promise((resolve) => {
    rl.question('Please enter your guess: ', (answer) => {
        resolve(parseInt(answer));
    });
});
```

Inside the promise, you're using `rl.question()` to prompt the user for input. This function takes a prompt (or query) as input and a callback function that will be executed when the user enters their response.

A callback function (shown in italics in the example above) is a function passed as an argument to another function, allowing it to be executed after the completion of that function's task. Pay careful attention as to where the italics start and end in the example above. I’ve underlined the start and end.

When the user enters their response (i.e., when `rl.question()`'s callback function is called), you're parsing their answer into an integer using `parseInt()`, and then resolving the promise with that value using `resolve(parseInt(answer))`.

Outside the promise, you're using the `await` keyword to wait for the promise to be resolved. This essentially means that the code will wait until the user has entered their input before proceeding.

So, in summary, your new code is waiting for the user to enter their guess, parsing it into an integer, and then making that guess available for further processing in your program.

Using what you’ve just learned, figure out why the program is ending before the first user input. The issue is related to how you’re calling the `guessingGame()` function. The fix is shown on the next page.

11. Change your call to `guessingGame()` to this:

```
await guessingGame();
```

The await keyword is needed when calling the guessingGame() function to ensure that the program waits for the asynchronous operations within that function to complete before proceeding. Without await, the guessingGame() function would start, but the program would not wait for it to finish.

Imagine you're baking a cake and you start preheating the oven. If you immediately move on to frosting the cake without waiting for it to bake, you'll be trying to frost raw batter. The cake isn't ready, and you've skipped a critical step, so the whole process fails.

In the same way, without the await keyword, the program would try to move on to closing the readline interface (like frosting the cake) without waiting for the guessing game (the baking process) to finish. This causes the program to end before the user even has a chance to interact, just like trying to frost a cake that hasn't been baked yet.

12. Ensure your game works.

Exercise

Let's test your understanding of these new concept.

1. Create a new file called "calculator.mjs".
2. Create functions called `generateEllipsis()`, `calculator()`, `getInput()`, and `start()`. All of the functions should be asynchronous.

- `generateEllipsis()`
  - This function takes no arguments.
  - This function should print three dots to the terminal, with a one second delay between each dot. Use a for loop to accomplish this.
    - To print to the terminal without a line break, use the `process.stdout.write()` method. It takes a string as an argument.
    - To wait for one second, you would use the `setTimeout()` function. However, using `setTimeout()` without a lecture of its own is too complicated for this assignment. Instead, drop this arrow-function into your code and call it in `generateEllipsis()` as needed:

```
const delay = (ms) => new Promise(resolve => setTimeout(resolve, ms));
```

  - Test the `generateEllipsis()` function by simply calling it in the global scope. Remember to use the `await` keyword when calling it!

- `calculator()`
  - This function should take three arguments: two operands (numbers) and one operator (string).

- This function should print out “Generating the result…”, The ellipsis **must** be generated by the `generateEllipsis()` function. Ensure a line break occurs after the ellipsis is generated.
- Use a switch statement to determine what math to do. For example, if the operand is a “+” sign, return the sum of the two operands.
  - Support only addition, subtraction, multiplication, and division. If an operator is passed into the method that is not those four, throw an `Error` with the message “Invalid operand”.

- `getInput()`
  - This function should take one argument: the prompt.
  - This function must do four things:
    - Create the interface for `readline`.
      - You can create the `readline` interface locally in a function!
    - Prompt the user.
    - Close the interface.
    - Return the user’s response.
  - Hint: Should you be parsing the user’s answer to an integer here? This function will get called to get operators ***and*** operands.
- `start()`
  - This function takes no arguments.
  - Get the input for your two operands using the `getInput()` function. Cast the operands to floats.
    - If you get the warning “Argument type unknown is not assignable to parameter type string”, you can resolve that by adding this JSDoc above your `getInput()` function:

      ```
      /**
      * @returns {Promise<string>}
      */
      ```

    - That effectively tells your IDE that your function will be returning a Promise object that should resolve to a string.
  - Get the input for your operator.
  - In a try block, call the `calculator()` function with your operands and operator.
  - Print what the `calculator()` function returns. You can make your output look nicer by formatting it however you want. Take a look at the example output on the next page.
  - The catch block will have a parameter called “error” and print `error.message` using `console.error()`.

3. Call the `start()` function after it has been defined.
4. Test to make sure it is working with a variety of inputs.
5. Commit and push your code to GitHub.

Example Output

```
Enter your first operand: 5
Enter your second operand: 4
Enter your operator: *
Generating the result...
5 * 4 = 20
```

Code Checklist

Check your code for the following:

☐ You are only using strict type equality checks (=== or !== instead of == or !=).
☐ You used semi-colons to terminate instructions.
☐ You did not use the `var` keyword.
☐ `let` is used only for variables that will change.
☐ Variables and functions are named using the camelCase convention.
☐ PhpStorm is not showing any warnings.

Submission

The submission for this lab is the commit ID you want graded. Please submit the commit ID on Canvas.

Tips

- While JavaScript is a semi-colon optional language, try to stay in the good habit of using them when terminating an instruction.
- Whenever you call an asynchronous function, you should use the `await` keyword before the call. While more complex tasks might require different approaches, using `await` is essential for this assignment.
- You can debug your code by clicking on the bug icon to the right of the play button.
- Syntax is available on Canvas. Avoid using search engines or AI.