Module 3 Labs
JS Advanced

1. `makeCounter` below is a decorator function which creates and returns a function that increments a counter.
   a) Create a second counter `counter2` using the `makeCounter` function and test to see if it remains independent to `counter1`
   b) Modify `makeCounter` so that it takes an argument `startFrom` specifying where the counter starts from (instead of always starting from 0)
   c) Modify `makeCounter` to take another argument `incrementBy`, which specifies how much each call to `counter()` should increase the counter value by.

```
function makeCounter() {
    let currentCount = 0;

    return function() {
        currentCount++;
        console.log(currentCount)
        return currentCount;
    };
}

let counter1 = makeCounter();

counter1(); // 1
counter1(); // 2
```

2. The following `delayMsg` function is intended to be used to delay printing a message until some time has passed.
   a) What order will the four tests below print in? Why?
   b) Rewrite `delayMsg` as an arrow function
   c) Add a fifth test which uses a large delay time (greater than 10 seconds)
   d) Use `clearTimeout` to prevent the fifth test from printing at all.

```
function delayMsg(msg)
{
    console.log(`This message will be printed after a delay: ${msg}`)
}

setTimeout(delayMsg, 100, '#1: Delayed by 100ms');
setTimeout(delayMsg, 20, '#2: Delayed by 20ms');
setTimeout(delayMsg, 0, '#3: Delayed by 0ms');
delayMsg('#4: Not delayed at all')
```

3. 'Debouncing' is a concept that refers to 'putting off' the execution of multiple, fast-timed, similar requests until there's a brief pause, then only executing the most recent of those requests. See https://www.techtarget.com/whatis/definition/debouncing
   It's often used to handle fast-firing scrolling events in a browser, or to prevent multiple server requests being initiated if a user clicks repeatedly on a button.

   Using the following code to test and start with:
   a) Create a `debounce(func)` decorator, which is a wrapper that takes a function `func` and suspends calls to `func` until there's 1000 milliseconds of inactivity. After this 1 second pause, the most recent call to `func` should be executed and any others ignored.
   b) Extend the `debounce` decorator function to take a second argument `ms`, which defines the length of the period of inactivity instead of hardcoding to 1000ms
   c) Extend `debounce` to allow the original debounced function `printMe` to take an argument `msg` which is included in the `console.log` statement.

```
function printMe() {
    console.log('printing debounced message')
}

printMe = debounce(printMe); //create this debounce function for a)

//fire off 3 calls to printMe within 300ms - only the LAST one should print, after
1000ms of no calls
setTimeout( printMe, 100);
setTimeout( printMe, 200);
setTimeout( printMe, 300);
```

4. The Fibonacci sequence of numbers is a famous pattern where the next number in the sequence is the sum of the previous 2.
   e.g. 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.

   a) Write a function `printFibonacci()` using `setInterval` that outputs a number in the Fibonacci sequence every second.
   b) Write a new version `printFibonacciTimeouts()` that uses nested `setTimeout` calls to do the same thing
   c) Extend one of the above functions to accept a `limit` argument, which tells it how many numbers to print before stopping.

5. The following `car` object has several properties and a method which uses them to print a description. When calling the function normally this works as expected, but using it from within `setTimeout` fails. Why?

```
let car = {
    make: "Porsche",
    model: '911',
    year: 1964,
```

```
    description() {
        console.log(`This car is a ${this.make} ${this.model} from ${this.year}`);
    }
};

car.description(); //works

setTimeout(car.description, 200); //fails
```

a) Fix the `setTimeout` call by wrapping the call to `car.description()` inside a function
b) Change the `year` for the car by creating a clone of the original and overriding it
c) Does the delayed `description()` call use the original values or the new values from b)? Why?
d) Use `bind` to fix the description method so that it can be called from within `setTimeout` without a wrapper function
e) Change another property of the car by creating a clone and overriding it, and test that `setTimeout` still uses the bound value from d)

6. Use the `Function` prototype to add a new `delay(ms)` function to all functions, which can be used to delay the call to that function by `ms` milliseconds.

```
function multiply(a, b) {
    console.log( a * b );
}

multiply.delay(500)(5, 5); // prints 25 after 500 milliseconds
```

a) Use the example `multiply` function below to test it with, as above, and assume that all delayed functions will take two parameters
b) Use `apply` to improve your solution so that delayed functions can take any number of parameters
c) Modify `multiply` to take 4 parameters and multiply all of them, and test that your `delay` prototype function still works.

7. The following `DigitalClock` class uses an interval to print the time every second once started, until stopped.

```
class DigitalClock {

    constructor(prefix) {
        this.prefix = prefix;
    }

    display() {
      let date = new Date();
      //create 3 variables in one go using array destructuring
      let [hours, mins, secs] = [date.getHours(), date.getMinutes(), date.getSeconds()];
```

```
        if (hours < 10) hours = '0' + hours;
        if (mins < 10) mins = '0' + mins;
        if (secs < 10) secs = '0' + secs;

        console.log(`${this.prefix} ${hours}:${mins}:${secs}`);
    }

    stop() {
        clearInterval(this.timer);
    }

    start() {
        this.display();
        this.timer = setInterval(() => this.display(), 1000);
    }
}

const myClock = new DigitalClock('my clock:')
myClock.start()
```

a) Create a new class `PrecisionClock` that inherits from `DigitalClock` and adds the parameter `precision` – the number of `ms` between 'ticks'. This `precision` parameter should default to 1 second if not supplied.
b) Create a new class `AlarmClock` that inherits from `DigitalClock` and adds the parameter `wakeupTime` in the format `hh:mm`. When the clock reaches this time, it should print a 'Wake Up' message and stop ticking. This `wakeupTime` parameter should default to 07:00 if not supplied.

8. Using the following starter code, create a decorator function to validate function arguments as strings. Test it by decorating the given `orderItems` function below.

```
function orderItems(itemName) {
    return `Order placed for: ${itemName}`;
}

// create a decorated version of the original function
const validatedOrderItem = validateStringArg(orderItems);

console.log(validatedOrderItem("Apple Watch")); // should run the function
console.log(validatedOrderItem(123)); // should throw an error
```

a) Create a decorator function `validateStringArg(fn)` which will validate an argument passed to `fn` to ensure that it is a string, throwing an error if not
b) Extend `orderItems` to use the `...` rest operator, allowing multiple item name arguments, and include them all in the returned string
c) Extend the decorator function to validate as strings all arguments passed to `fn`
d) When testing the decorated function, use try-catch blocks to handle errors thrown for non-string arguments

9. We can delay execution of a function using `setTimeout`, where we need to provide both the callback function and the delay after which it should execute.
   a) Create a promise-based alternative `randomDelay()` that delays execution for a random amount of time (between 1 and 20 seconds) and returns a promise we can use via `.then()`, as in the starter code below
   b) If the random delay is even, consider this a successful delay and `resolve` the promise, and if the random number is odd, consider this a failure and `reject` it
   c) Update the testing code to `catch` rejected promises and print a different message
   d) Try to update the `then` and `catch` messages to include the random `delay` value

```
function randomDelay() {
    // your code
}
randomDelay().then(() => console.log('There appears to have been a delay.'));
```

10. Fetch is a browser-based function to send a request and receive a response from a server, which uses promises to handle the asynchronous response.
    The below `fetchURLData` uses `fetch` to check the response for a successful status code, and returns a `promise` containing the JSON sent by the remote server if successful or an error if it failed. *(To run this code in a node.js environment, follow the instructions in the comments before the function.)*
    a) Write a new version of this function using `async`/`await`
    b) Test both functions with valid and invalid URLs
    c) (Extension) Extend your new function to accept an array of URLs and fetch all of them, using `Promise.all` to combine the results.

```
// run 'npm init' and accept all the defaults
// run 'npm install node-fetch'
// run 'npm pkg set type=module'

import fetch from 'node-fetch'
globalThis.fetch = fetch

function fetchURLData(url) {
    let fetchPromise = fetch(url).then(response => {
        if (response.status === 200) {
            return response.json();
        } else {
            throw new Error(`Request failed with status ${response.status}`);
        }
    });

    return fetchPromise;
}
```

```
fetchURLData('https://jsonplaceholder.typicode.com/todos/1')
    .then(data => console.log(data))
    .catch(error => console.error(error.message));
```