1) Critique the specifications for Project 1. Specifically, discuss whether the validation required for the API is incomplete and allows inconsistent data.

Answer) One of the biggest problems that arises with the designed API is that false student records can be created because students are still able to be assigned specific value scores even though they are not technically enrolled in the section. Furthermore, the API still computes aggregates when data is still not technically correct and/or if it is missing and this can cause data conflicts with aggregating grades.

2. In JavaScript, a common error is to forget to call an `async` function using an `await`, as illustrated in the following REPL log:

```
> async function f() { return { fn: () => 22 }; }
undefined
> (await f()).fn()  //correct call
22
> f().fn()          //incorrect call
Uncaught TypeError: f(...).fn is not a function
>
```

Why is this kind of error less likely in a TypeScript program? *5-points*

Answer) The reason that the common error is less likely to happen in typescript is because typescript will be able to catch the incorrect use of wait in this code because it is going to expect a promise to be returned. Overall, it verifies that a specific type is able to be used correctly before actually running the program, known as static type checking and this code will cause a compile time error

3. Critique and fix the following code (where the specs should be obvious from the function names):

```
//no problem here
async function getImage(imgUrl) { ... }

async function getAllImages(imgUrls) {
  return await imgUrls.map(async u => await getImage(u));
}
```

Besides correctness, performance is also important. *10-points*

Answer) The issue with this code is that we have the chance to return false promises because we are using “map”. This will return the array right away and not wait for the callback that we need to ensure the promises are reliable and correct. Finally, the “await” blocks execution and we should remove this to improve performance. Thus, this is the fixed code

```
async function getAllImages(imgUrls){
          Return Promise.all(imgUrls.map(u => getImage(u)));
}
```

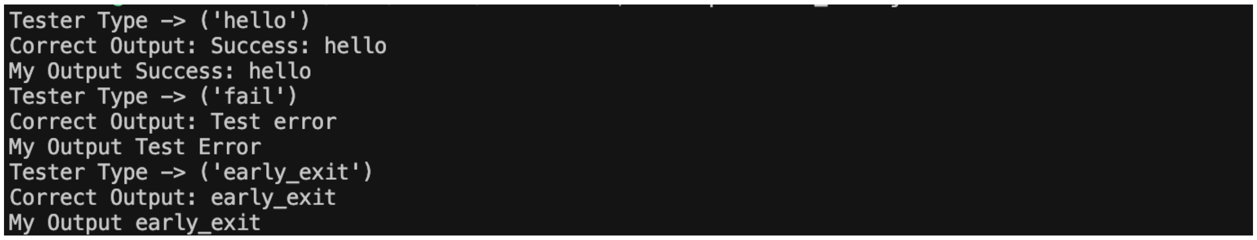
4. A legacy library has a large set of asynchronous functions all of which take a callback as their last argument:

Pseudo TS syntax:

```
function someLegacyFn(
  ...someArgs,
  callBackFn: (succ: any, error: any) => void)
{
  //do some async operation; if operation fails call
  //callBackFn with a non-null error value; otherwise
  //call callBackFn with success value as succ.
  ...
}
```

Show code for a function `makeAsyncAwait(Fn)` which takes a legacy function `Fn` and returns a function which can be called using modern `async` / `await` syntax. *15-points*

```
//---------------------------Start-of-Input-of-Question4---------------------------------------------------------------
function Q4<T extends(...inputParams: any[]) => void>(type: T){
  return function(){
     const args_type = Array.prototype.slice.call(arguments)
     return new Promise<any>((find, remove) =>{
        const find_output = (print_result: any, err: any) => {
           if(err){
                remove(err);
            }else{
                find(print_result);
            }
        };
        const call_find_output = function(){
            if (arguments[0] == "early_exit"){
                find_output("early exit handled", null);
            }else{
                find_output(arguments[0], arguments[1]);
            }
        };
        type.apply(null, args_type.concat(call_find_output));
    });
  };
}
//---------------------------End-Of-Implentation-Of-Question4------------------------------------------------
//Test My Code
async function processInput(que: string): Promise<any>{
  return new Promise((fulfill, deny) =>{
    setTimeout(() => {
        switch(que){
            case 'fail':
                deny(new Error('Test Error'))
                break;
            case 'early_exit':
                fulfill('early_exit');
                break;
            default:
                fulfill(`Success: ${que}`);
        }
    }, 500);
  });
}
async function executTrials(){
  const tester = [
    { que: 'hello', exp: 'success: hello'},
    { que: 'fail', exp: 'test error', inc: true},
    { que: "early_exit", exp: 'early_exit'},
  ];
  for(const { que, exp, inc} of tester){
    console.log(`Tester Type -> ${que}`);
    console.log(`Correct Output: ${que}`);
  try{
    const answer = await processInput(que);
    if (!inc){
        console.log('my output ->', answer);
    }else{
        console.log('worng answer', answer);
        }
    }catch (err){
        if (inc){
            console.log("my output", err.message);
        }else{
            console.error("my output:", err)
        }
    }
  }
}
executTrials();
```



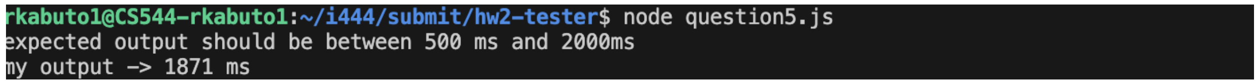
```
Tester Type -> ('hello')
Correct Output: Success: hello
My Output Success: hello
Tester Type -> ('fail')
Correct Output: Test error
My Output Test Error
Tester Type -> ('early_exit')
Correct Output: early_exit
My Output early_exit
```

5) A website depends on an external service accessed using an async function getService(). To facilitate development, a developer uses a local instance of that service running on their laptop. This results in a great dev-ex, but at some point the developer would like to experience a more realistic UX by adding random delays to the locally running service. Show code which will wrap getService() within a random parameterizable time delay.

```
function Q5(call: () => Promise<any>, min_type: number, max_type: number): Promise<any>{
    let GP: number;
    switch(true){
        case(max_type >= min_type):
            GP = Math.floor(Math.random() * (max_type - min_type + 1)) + min_type;
            break;
        case( max_type < min_type):
            GP = Math.floor(Math.random() * (max_type - min_type + 1)) + max_type;
            break;
        default:
            GP = 0;
            break;
    }
    switch(true){
        case(GP >= 0):
            return new Promise(resolve => setTimeout(resolve, GP)).then(() => {
                return 'service'
            });
        default:
            return Promise.reject('error -> pause');
    }
}

//Below is code to test Question 5
```

```
async function type_delay(){
  const min_type = 500;
  const max_type = 2000;
  const time_s = Date.now()
  const res = await Q5(() => Promise.resolve("service"), min_type, max_type);
  const time_e = Date.now()
  const real_delay = time_e - time_s;
  console.log(`expected output should be between ${min_type} ms and ${max_type}ms`)
  console.log(`my output -> ${real_delay} ms `)
}
type_delay();
```



```
rkabuto1@CS544-rkabuto1:~/i444/submit/hw2-tester$ node question5.js
expected output should be between 500 ms and 2000ms
my output -> 1871 ms
```

6. An `async` function `asyncFn()` calls a synchronous function `syncFn()` with an unnecessary await in front of it.

```
function syncFn() { return 22; }

async function asyncFn() {
  //some other code which really needs await
  const v = await syncFn();
  return v;
}
```

How does the unnecessary `await` affect the operation of the program? *10-points*

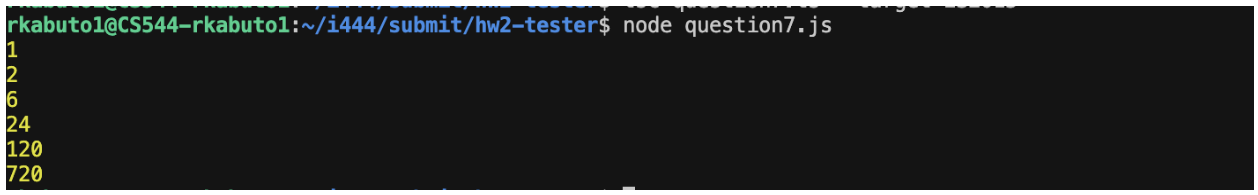
Answer) The unnecessary await affects the operation of the program because it causes the code to perform an operation before continuing to execute resulting in the output. This is due to the fact we are causing our function to return as a promise rather than just using syncFn() to return the value immediately. Overall, this is just unnecessary steps being done.

7. Write a function `fact(n)` which can be used as in

```
for (f of fact(6)) console.log(f);
```

to print out successive factorials 1, 2, 6, 24, 120, 720. The return value of `fact()` is not allowed to be a collection type like an array, `Set` or `Map` *10-points*

```
function* fact(n: number): Generator<number>{
  let fac = 1;
  for (let i = 1; i <= n; i++){
    fac = fac * i;
    yield fac;
  }
}
for(let f of fact(6)) console.log(f);
```



```
rkabuto1@CS544-rkabuto1:~/i444/submit/hw2-tester$ node question7.js
1
2
6
24
120
720
```

8. Assuming no earlier variable declarations:

```
x = 1;
obj1 = { x: 2, f: function() { return this.x; } }
obj2 = { x: 3, f: function() { return this.x; } }
f = obj1.f.bind(obj2);
console.log(obj1.f() - obj2.f() + f());
```

Explain why the output of the above JavaScript code when run in non-strict mode is 2. *10-points*

Answer) When non-strict mode is going, we are going to return 2, which we get from 2 - 3 + 3 = -1 + 3 which leads us to 2. The reason we are getting this is because obj1 is carrying the 2 and obj2 is going to carry the value 3. These are given things that we can clearly see in the code. However, f() is going to be holding the value of obj2 because we are in non-strict mode and this means that this is going to have the value of 3 as well. So therefore, this concludes on how we get the value 2.

9) A novice programmer is asked to fix a bug in a JavaScript codebase. While fixing the bug, they notice some anonymous functions defined using the function keyword and other anonymous functions defined using the fat-arrow =>. They prefer the pleasanter syntax of the fat-arrow function, and having just read a recent blog post on Code Consistency, they decide to replace all functions defined using function with functions defined using fat-arrow. Why did they just break the codebase?

Answer) The reason is because "this" is going to be working in a global scope which is going to cause "this" to do weird behaviors in objects, functions and more. Fat arrow is just going to rely on the surrounding scope and is going to change the functionality of "this" and this is why the novice programmer just broke the codebase.

10) Discuss the validity of the following statements.

a. **this for a fat-arrow function can be changed using bind().**
    i. **False.** The reason that this is not a true statement is because the fat-arrow functions are based on the scope in which it was defined so bind() will have zero effect on "this"

b. **In modern JavaScript, having both call() and apply() are redundant; i.e. each one can be implemented in terms of the other.**
    i. **True.** The reason that this is a true statement is because call and apply will generally output the same thing. The only difference between call and apply is call requires the arguments to be passed in one by one while apply takes the arguments in an array

c. **It is impossible to wrap an asynchronous function within a synchronous function.**
    i. **False.** The reason that this is a false statement is because you are able to wrap an asynchronous function with a synchronous function using async and await

d. **The return value of a then() is always a promise.**
    i. **True**. This is just a factual statement and will always return a new promise for the result of a callback

e. **The promise returned by Promise.all() will become settled after a minimum time which is the sum of the times required for settlement of each of its individual argument promises.**
    i. **False.** The reason that this is a false statement is because Promise.all() will be settled when the longest-running promise in the iterable settles. If any promise is rejected, Promise.all() rejects right away and stops waiting for the rest.