HomeWork-2

1. Assuming the below example files given in question:

```
//in file msgs_en_us.mjs
const MSGS = {
  ...
  'msg.hello': 'Hello',
  ...
};
export default MSGS;

//in file msgs_fr_ca.mjs
const MSGS = {
  ...
  'msg.hello': 'Bonjour',
  ...
};

export default MSGS;
```

The code for importing only the module for localeStr:

```
const localeStr : string = 'msgs_' +
navigator.language.toLowerCase().replaceAll('-', '_');

const mjs = await import(localeStr);
const MSGS = mjs['MSGS'];
```

2. Below is the code which wraps getService() within a random parameterizable time delay:

```
async function getservice():Promise<string>{
  const service: string = 'Here is your service';
  return service;
}

async function wait(delayTime: number){
      return new Promise(resolve => setTimeout(resolve, delayTime));
}

async function getServiceWrapper(minDelay: number,
maxDelay:number):Promise<string>{

   const delayTime: number = minDelay + (maxDelay-minDelay)*Math.random();
   await wait(delayTime);
   return await getservice();
}

async function test(){
    console.log('service requested');
    console.log(await getServiceWrapper(4000,5000));
}
test();
```

- To get delayed service the test function calls the getServiceWrapper function with min and max delay in milliseconds as parameters.
- The getServiceWrapper function generates delayTime by multiplying a random number between 0 and 1 with the difference of min-max delay and adding it to min Delay.
- The wait function is then called with the delayTime as param. Wait function uses the setTimeout method to create a specified delay before calling resolve.
- After this delay getServiceWrapper function calls the getService() method and returns the result.

3.

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

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

The problem with above code is that, we are using await before getImage in the async callback function of map. So, the callback function will wait for the getImage promise to complete. But map is not an async function and it will not wait before creating an array of incomplete promises, which if used can cause issue. Also, await before the getImage will cause the map to run sequentially which reduces performance. The correct way is to await for all the unresolved promises returned by map. We can do this by awaiting on promise.all() which resolves when all the promises inside it resolves:

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

4. There are some differences in functions defined using ‘functions’ keyword and those defined using fat arrow, which might have caused the bugs in codebase. Mainly, functions defined using fat arrow does not have its own ‘this’ context instead it inherits it from enclosing scope. On the other hand functions defined using ‘functions’ keyword have their own this context. Thus, we can create instances of the later functions. If the codebase was using such instances, it might have broken.

5.

```
function convertLegacyfunction(Fn) {

 function convertedFunction(...args) {

    return new Promise((resolve, reject) => {

      Fn(...args, (success, error) => {
        if (error) {
          reject(error);
        } else {
          resolve(success);
        }
      });

    });

  };

  return convertedFunction;
}
```

- Above function ‘convertLegacyfunction’ takes a legacy function ‘Fn’ as argument and returns a convertedFunction.
- This converted function can be called using await.
- This function returns a promise as does the modern async functions and inside that, the legacy function is called with arguments provided in convertedFunction call.
- This promise is resolved or rejected based on the result of callback function provided in legacy function.

6. In JavaScript, a common error is to forget to call an `async` function using an `await`. This kind of error less likely in a TypeScript program because TypeScript performs static type checking. So, it will detect that the function f() being called returns a Promise that resolves to an object that contains fn() function. If we try to call f().fn() without await on f(), TypeScript will detect that f() can return a promise that might not contain fn() method and hence gives error.

7.

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

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

- The unnecessary await before the synchronous function syncFn() does not affected the execution flow of the program nor gives any error.
- The await keyword is basically used to wait for asynchronous function's promise to evaluate and using it with a synchronous function that returns a value that is not of promise type does not cause any issue because await will convert that value to resolved promise immediately.
- Also, since the value is immediately returned by a synchronous function, using await will not delay the further execution of program.

8. Below is the fact function that returns the expected output:

```
function* fact(x:number) {
  for (let i = 1, j=1; i <= x; i++) {
    j = i * j;
    yield j;
  }
}

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

The function simply uses for loop with two variables i and j to yield the expected output in each iteration.

9.

```
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());
```

- R5The output of the above JavaScript code when run in non-strict mode is 2 because in the statement f = obj1.f.bind(obj2); bind is used to bind obj1.f to obj2. So, 'this' inside f will refer to obj2 and this.x is 3.

obj1.f() = 2

obj2.f() = 3

f() = 3

Finally, obj1.f() - obj2.f() + f() evaluates to 2.

But if the program is run in strict mode, this inside f will be undefined as it is not a member of any object.

10.

(a) For a fat arrow function, this cannot be changed using bind because functions defined using fat arrow does not have their own this context instead it is inherited from enclosing scope.

(b) Yes, In modern JavaScript, having both call() and apply() are redundant; i.e. each one can be implemented in terms of the other because both call() and apply() are used to call a function with specified this context. Only difference is call() takes arguments in form of list and apply() arguments as array. With the spread (…) in modern javascript you can use either of them.

(c) Yes, it is impossible to wrap an asynchronous function inside a synchronous function in javascript because even if asynchronous function is called from inside of a synchronous function, there is no guarantee that the asynchronous function will return resolved result before the completion of synchronous function.

(d) Yes, the return value of a then() is always a promise. It takes two callback functions as argument. One for the success and other failure case of the Promise. The callback functions might return some values but, then() always returns a Promise that resolves to the returned value.

(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."

Above statement is incorrect because Promise.all() becomes settled when all Promise inside it settles. But those promises can be completed concurrently and not require individual time for each promise, like in the case of map in question 3rd.