Detailed guide

Task 1: configuration and basic classes

1) Define class Person in file Person.ts, with constructor taking name as parameter
    a. Implement getInfo() function using string interpolation: it should return String representation of Person
    b. Implement get/set name; set name should validate name length: it should be not less than 3

    1) Define class Person in Person.js, export it

    ```
    export class Person {
        constructor(public name:string) {}
    }
    ```

    2) Define getInfo() method:

    ```
        getInfo() {
            return `person: ${this.name}`
        }
    ```

2) Define Employee class extending Person, with adding properties salary and position, and overriding getInfo()
    1) Create Employee.ts and import Person

    ```
        import {Person} from './Person'
    ```

    2) Create Position.ts and import it to Person.E

    3) Define Employee class which extends Person:

    ```
        export class Employee extends Person {
            constructor(_name:string, public
        position:Position, public salary:number) {
                super(name);
            }

        }
    ```

    4) Override getInfo() and call getInfo() from superclass

    ```
         getInfo() {
             return super.getInfo()+` ${this.position}
    ${this.salary}`
         }
    ```

3) Define Employees class with encapsulated list of employees with static metods:
   a. add() to add employee to hidden employees list; it should include type check and throw exception if added value is   not Employee
   b. list() which returns a copy of all employees list

1) Create Employees.ts and import Employee

```
    import { Employee } from "./Employee"
```

2) Create exported class Employees

```
    export class Employees { }
```

3) Define module variable which will keep list of employees:

```
    static employees:Array<Employee>=[];
```

4) Add static method add() which adds new employee to _employees array and check if argument is Employee

```
static add(employee:Employee) {
    this.employees.push(employee);
}
```

5) Add static method list() which returns copy of employees list

```
static list():Employee[] {
    return [...this.employees];
}
```

4) Create main.ts which should:
   a. create several employees and add to Employees using add() function
   b. print list of employees with use of getInfo() method

1) Create main.ts and import Employee, Position and Employees

```
    import {Employees} from "./Employees"
    import {Employee} from "./Employee"
    import {Position} from "./Position"
```

2) Create default exported method and add several employees and add it with use of Employees.add()

```
export default function() {
   Employees.add(new Employee("John","manager",1000));
   Employees.add(new Employee("Bill","developer",5000));
   Employees.add(new Employee("James","director",4000));
}
```

3) Retrieve list of employees

```
let employees:Employee[] = Employees.list();
```

4) Create variable as html placeholder

```
    let html=""
```

5) Iterate over employees to add html representation

```
for (let e of employees) {
    html += e.getInfo()+"<br>"
}
```

6) Put resulting html to the web page

```
document.getElementById("employees").innerHTML = html;
```

5) Create employees.html which should use main.js and show all information

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Employees</title>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="bundle.js"></script>
</head>
<body >
    <div id="employees"></div>
</body>
<script>
    SystemJS.import('main').then(function (m) {
        m.default();
    });
</script>
</html>
```

6) Open employees.html in browser. It should show list of employees with information generated by getInfo()

Task 2: using arrow functions and map/reduce

1) Add static method averageSalary() to Employees class which calculates average salary of all employees (use map/reduce)

   Add method averageSalary() to Employees class in Employees.js:

```
static averageSalary() {
    return Math.round(
        this.employees.map(e=>e.salary).reduce((a,b)=>a+b)
        /this.employees.length)
}
```

2) Update main.js which should print average salary of employees

```
html += `Average salary: ${Employees.averageSalary()} <p>`;
```

Task 3: using promises

1) Add method bonus() to Employee which should return Promise having randomly generated bonus in range 0…1000; bonus should be calculated after 1000ms timeout (it imitates long server-side request)

```
bonus():Promise<number> {
    return new Promise(resolve=>
        setTimeout(
            ()=>resolve(Math.round(Math.random()*1000)),
            1000))
}
```

2) Add method total() to Employee which should calculate sum of bonus and salary and return new Promise

```
total():Promise<number> {
    return new Promise(resolve=>
        this.bonus().then(bonus=>
            resolve(bonus+this.salary)))
}
```

Note that we are able to access `this.salary` because of lexical scoping of `this` in arrow function.
For not arrow function we would need to use bind() or this renaming.

3) In main.js move code which modifies html to separate function render:

```
function render() {
    document.getElementById("employees").innerHTML = html;
}
```

Add call to render() to the end of main.js

4) Print total income of every employee in main.js with use of promises:

```
for (let e of Employees) {
    e.total().then(total=>{
        html += `${e.name} total: ${total} <br>`;
        render();
    });
}
```

Task 4: adding exception handling to promises

1) Change bonus() method in Employee class so that it reject Promise if bonus is more than 700

```
bonus():Promise<number>  {
    var bonus = Math.round(Math.random()*1000);
    return new Promise((resolve,reject)=>
       setTimeout(()=>bonus<700?resolve(bonus):reject(bonus),1000))
}
```

2) Change total() method in Employee which handles exception in bonus() and rejects Promise as well

```
total():Promise<number> {
    return new Promise((resolve,reject)=>
        this.bonus()
            .then(bonus=>resolve(this.salary+bonus))
            .catch(bonus=>reject(bonus))
    )
}
```

3) Update printing list of employees in main.js (Promise verision) by adding catch block which will print «Bonus is impossibly big» for the employee in case of exception

```
for (let e of Employees) {
    e.total()
        .then(total=>
            html += `${e.name} total: ${total} <br>`)
        .catch(bonus=>
            html += `${e.name} bonus is too big (${bonus}!)
<br>`)
        .then(render)
}
```

Task 5: using async/await syntax (

1) Create async function in main.ts to print list of employees and bonuses

Add to main.js function with async/await syntax:

```
async function printBonus() {
    html += "<br>Async/await version:<br>";
    for (let e of Employees) {
        let bonus = await e.bonus();
        html += `${e.name} bonus: ${bonus}
               total: ${e.salary+bonus}<br>`;
        render();
    }
}
```

2) Execute it

```
printBonus();
```

Now you can reload employees.html and see the asynchronous work.

Task 6: using generators

1) Add these static functions to Employees:
   a. iterator which allows to iterate over all employees using for (let e of Employees) (**Hint**: function name should be *[Symbol.iterator])
   b. generator names() which allows to iterate over all employee names

Hint: you should add require("babel-polyfill") to allow generators support in browser

1) Define iterator method in Employees class:

```
static *[Symbol.iterator]() {
    yield* _employees;
}
```

2) Define names() generator in Employees wich iterates over employees names:

```
static *names() {
    yield* _employees.map(e=>e.name);
}
```

2) Update main.js which should:
   a. iterate over Employees using for…of and print info by calling getInfo()
   b. print all employees names separated by comma

1) Modify for loop in main.js with use of iterator:

```
for (let e of Employees) {
    html += e.getInfo()+"<br>"
}
```

2) Print all employees names separated by comma

```
let names = [...Employees.names()];
html += `Names: ${names.join(", ")} <p>`
```

Task 7: using decorators

1) Define property _age in Employee class, with getter and setter, decorate setter with @Range decorator – is should validate that setter parameter is between 18 and 80:

```
export class Employee extends Person {
    _age: number;

    set age(_age:number) {
        this._age = _age;
    }

    @Range(18,80)
    get age():number {
        return this._age;
    }
}
```

2) Create file Range.ts and define decorator @Range

```
export function Range(from:number, to:number):any {
    return function<T extends number>(target:any, key:string,
desc:any) {
        let oldFunc = desc.set;
        desc.set = function () {
            let value = arguments[0];
            if (value<from || value>to) throw new Error("Wrong
value of field "+key);
            oldFunc.apply(target,arguments);
        };
        return desc;
    }
}
```

3) Add this code to main.ts to check how @Range decorator is working:

```
let older = new Employee("Old", Position.MANAGER, 5000);
try {
    older.age=100;
} catch(e) {
    console.log(e);
}
Employees.add(older);
```

Task 8: higher order function

In class Employees create function sum():number which takes function (Employee)=>number as a parameter and calculates sum of values. Calculate the total salary of employees.

1) Define function sum():

```
static sum(f:(e:Employee)=>number):number {
    return this.employees.map(f).reduce((x,y)=>x+y);
}
```

2) Use sum() function to calculate total sum of salaries:

```
let html = "";
for (let e of employees) {
    html += e.getInfo()+"<br>"
}
html += "Total salary: "+Employees.sum(e=>e.salary);
```