Typescript

FUNCTION TYPE

Define the following functions and type their arguments and return values

Write a function *sumArray()* that calculates sum of items in an array of numbers and returns it. Type the function arguments and return value inline.
let result = sumArray([1, 2, 3, 4]); console.log(result); // 10

```
function sumarray(a:number, b:number):number {
   return a+b;
}
let res =sumarray(10,19);
console.log(res);
```

1. Write a function *squareEach()* that creates a new array with squares of numbers in a supplied array, and returns it. Type the function separately and assign the type to the function.

   let result = squareEach([1, 2, 3, 4]);
   console.log(result); // [1, 4, 9, 16]

```
type squareEach = (arr : number[])=> number[];
let ans:number[];
let res : number[];
const fun : squareEach = function(arr){
   let x=[];
   for(let i=0; i<arr.length; i++)
   {
      x [i]= arr[i]*arr[i];

   }
   return x;
```

}

2. Write a function *contains()* that accepts an array that can have any primitive value, and another primitive value as the second argument, and returns true if the second argument appears in the array, and false otherwise.

```
console.log( contains( [ 1, 'hello', 3, true ], 3 ) ); // prints true
console.log( contains( [ 1, 'hello', 3, true ], 5 ) ); // prints false
```

```
function contains(x:any[], y:any):boolean{
    let ans = false;
    for(let i=0; i<x.length; i++)
    {
        if(x[i]==y)
        {
            ans = true;
            return ans;
        }

    }
    return ans;
}
console.log(contains([1,"divya", true],"divya"));
```

3. Write a function *map()* that accepts an array as the first argument, and another function as the second argument. The second argument is a function that accepts an item of the array and returns some value. The *map()* functions applies the passed function (second argument) on each item of the passed array (first argument) in the order they appear in the array, and groups the results into an array and returns it.

```
function square( x ) { return x * x };
```

```
function cube( x ) { return x * x * x };


console.log( map( [ 1, 2, 3, 4 ], square ) ); // prints [ 1, 4, 9, 16 ]

console.log( map( [ 1, 2, 3, 4 ], cube ) ); // prints [ 1, 8, 27, 64 ]



type transFun = (x:number)=>number;

const square : transFun = (x)=>x*x;

const cube : transFun =(x)=>x*x*x;


function map(arr:number[], transform: transFun): number[]{

   let ans = [];

   for(let i=0; i<arr.length; i++){

      ans[i]= transform(arr[i]);

   }

   return ans;

}

console.log(map([1,2,3,4], square));

console.log(map([1,2,3,4], cube));
```

4. Write a function filter() that accepts an array and another function f (which returns a boolean value). The filter function should work like so.

```
 function isOdd( x )
    { return x % 2 ===
    1;
```

```
}

let filteredList = filter( [ 1, 2, 3, 4, 5, 6, 7, 8 ], isOdd ); // [ 1, 3, 5, 7 ]
```

```
ype isOddfun = (x:number) => boolean;

const isOdd : isOddfun = function(x:number):boolean{
        if(x%2==0)
        {
            return false;
        }
        else
        {
            return true;
        }
}
function filter(arr:number[], transform :isOddfun): number[]{
    let aa = [];
    for(let i=0; i<arr.length; i++)
    {
        if(transform(arr[i])== false)
        {
            aa.push(arr[i]);
        }
    }
    return aa;
}
```

```
console.log(filter([1,2,3,4,6], isOdd));
```

5. Write a function *exponentFactory* that accepts a number, say x. Define 2 functions *square* and *cube* within it (which accept a number each, and return the square and cube respectively). If x is 2, exponentFactory returns the square function, if 3 it returns the cube function. For any other input it returns a function that returns the number it accepts as such. Call the exponentFactory() function and then the returned function, and log the result.

Example:

```
var fn;



fn = exponentFactory( 2 );
console.log( fn( 5 ) ); // prints 25;


fn = exponentFactory( 3 );
console.log( fn( 5 ) ); // prints 125;


fn = exponentFactory( 4 );
console.log( fn( 5 ) ); // prints
5;


function
exponentFactory(x:number):n
umber{

   function
square(y:number):number{

      return y*y;
```

```
    }
    function
cube(y:number):number{
        return y*y*y;
    }
    function
same(y:number):number{
        return y;
    }
    if(x==2)
    {
        return square(x);
    }
    else if(x==3)
    {
        return cube(x);
    }
    else
    {
        return same(x);
    }
}
console.log(exponentFactory(
3));
```

FUNCTION OVERLOADING

Define a function *push()* that accepts 2 arguments.first argument is an array of numbers

6. second is either a number or an array of numbers
7. If second argument is a number, the function adds the number to the end of the array. If second argument is an array of numbers, the items of the array are pushed to the end of the array.
8. Your function push() should return the array (first argument it accepts).
9. *Tip*: You may use the spread operator to simplify your logic

```
function push(arr:number[], val : (number| number[])): number[]{

    if(typeof val == 'number')
    {
       arr.push(val);
       return arr;
    }
    return arr.concat(val);

}
console.log(push([1,2,3], [1,2]));
```

10. Define a function *log()* that accepts either

a. One argument - message (string)

b.Two arguments - format ('standard' | 'verbose') and message (string)

If its called with one argument, it simply prints the message. If it is called with 2 arguments it prints the message if format is 'standard', and prints the message with current date if format is 'verbose'

Define appropriate function overloads

```
function log(message: string , format?: ('standard'|'verbose')){

   if(message && !format)
```

```
    {
        console.log(message);
    }
    else
    {
        if(format=='standard'){
        console.log(message);}
        if(format=='verbose')
        {
            console.log(message+ new Date());
        }
    }
}
```

OBJECT TYPE

11. Define an interface IClock with type ('digital' | 'analog'), and a time property (an object) with properties - hours, minutes, seconds (all numbers). Your interface also defines a method setTime(hours, minutes, seconds) that sets the time, and getTime() that returns a string representation of the time. Create 2 objects of IClock type - one of type 'digital' and other of type 'analog', set the time through setTime() and log the time using getTime().

```
type timeObject = {
    hours :number,
    minutes:number,
    seconds:number
}

interface Iclock {
    type: String,
    time:timeObject,
    setTime:(hours:number , minutes:number, seconds:number) =>void,
    getTime:()=>string
}

let time1:timeObject = {hours : 0,
                        minutes:0,
                        seconds:0};

let clock1:Iclock = {
    type :'digital',
    time:time1,
    setTime(hours,minutes,seconds){
```

```
        this.time.hours = hours;
        this.time.minutes = minutes;
        this.time.seconds = seconds;
    },
    getTime(){
        let time:string = "";
        if(this.type === 'digital'){
            time = this.time.hours.toString() + ":" +this.time.minutes.toString() + ":" +
this.time.seconds.toString();
        }
        else{
            time = this.time.hours.toString() + ":" +this.time.minutes.toString();
        }
        return time;
    }
}
let clock2:Iclock = {
    type :'analog',
    time:time1,
    setTime(hours,minutes,seconds){
        this.time.hours = hours;
        this.time.minutes = minutes;
        this.time.seconds = seconds;
    },
    getTime(){
        let time:string = "";
        if(this.type === 'digital'){
            time = this.time.hours.toString() + ":" +this.time.minutes.toString() + ":" +
this.time.seconds.toString();
        }
        else{
            time = this.time.hours.toString() + ":" +this.time.minutes.toString();
        }
        return time;
    }
}
clock1.setTime(10,34,56);
clock2.setTime(3,56,59);
console.log(clock1.getTime());
console.log(clock2.getTime());
```

CLASS AND INHERITANCE

12. Define a Project class with id (number - public), name (string - public), client (string - private). Define some Project objects (suggest using sample data below).

```
const dbsPayroll = new Project( 1001, 'DBS payroll', 'DBS' );
const intranetDeployment = new Project( 2001, 'Intranet v2 deployment', 'Internal' );


class Project{
   //public id: number;
   //public name:string;
   //private client:string;
   constructor(public id:number, public name:string, private client:string){
      this.id= id;
      this.name = name;
      this.client = client;
   }
}
const dbsPayroll = new Project( 1001, 'DBS payroll', 'DBS' );
const intranetDeployment = new Project( 2001, 'Intranet v2 deployment', 'Internal' );
console.log(dbsPayroll);
console.log(intranetDeployment);
```

13. Define an Employee class with id (number - public), name (string - public), department (string - public), projects (array of Projects - private). **Use the access specifiers in constructor arguments** to setup the initial values for data members automatically. Define some Employee objects (suggest using sample data below).

```
const john = new Employee( 1, 'John', 'Web Developer', 'IT', [ dbsPayroll ] );
const jane = new Employee( 2, 'Jane', 'Project Manager', 'IT', [ dbsPayroll,
intranetDeployment ]
);
const mark = new Employee( 3, 'Mark', 'System Administrator', 'Operations',
[ intranetDeployment ] );
```

```

class Project{

    constructor(public id:number, public name:string, private client:string){
        this.id= id;
        this.name = name;
        this.client = client;
    }
}

class Employee{

    constructor(public id:number, public name:string, public department:string , private projects?:Project[]){
        this.id = id;
        this.name = name;
        this.department = department;
        this.projects = projects;
    }

    /*public addProject:(project:Project)=>void = (project)=>{
        this.projects?.push(project);*/

    }

const dbsPayroll = new Project( 1001, 'DBS payroll', 'DBS' );
const intranetDeployment = new Project( 2001, 'Intranet v2 deployment', 'Internal' );

const john = new Employee( 1, 'John', 'Web Developer', [ dbsPayroll ] );
const jane = new Employee( 2, 'Jane', 'Project Manager',  [ dbsPayroll, intranetDeployment ] );
const mark = new Employee( 3, 'Mark', 'System Administrator', [ intranetDeployment ] );

console.log(john);
console.log(jane);
```

```
console.log(mark);
//mark.addProject(dbsPayroll);
//console.log(mark);
```