TypeScript

LAB GUIDE

FUNCTION TYPE

Define the following functions and type their arguments and return values

1. 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
```

2. 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 ]
```

3. 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
```

4. 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 ]
```

5. 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 ]
```

6. 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 OVERLOADING

7. Define a function *push()* that accepts 2 arguments

* first argument is an array of numbers
* second is either a number or an array of numbers

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.

Your function push() should return the array (first argument it accepts).

Tip: You may use the spread operator to simplify your logic

8. 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

9. Define a function hireCar() that accepts either

a. Two arguments - endDate (Date), carType ('hatchback' | 'sedan' | 'suv' - defaults to 'sedan')

b. Three arguments - startDate (Date), endDate, car (all types same like 2 arguments case)

If its called with 2 arguments, return a Booking object with all details passed, the current date as startDate, price (number), and driverName (string).

If it is called with 3 arguments, a similar object with details passed is created and returned.

Define appropriate function overloads.

OBJECT TYPE

10. 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().

CLASS AND INHERITANCE

11. 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' );
```

12. 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 ] );
```

GENERICS

13. Create a generic interface for an Item, say IItem - an item would have an id (number), name (parameter type), and getName() that returns a string representation of the name. Define 2 classes ItemShortName (with name - a string) and ItemLongName (with name - an object with manufacturer and product names, both strings) that each implement IItem.

Now define 2 different generic functions - say, printItems1() and printItems2() which both accept an array of IItem<T> objects and print the details of each object in the array. printItems1() should take a generic parameter representing the type of name, and printItems2() takes a generic parameter which is restricted a subtype of IItem<any>.

14. Redo the BillableItem - Product - Service example using class hierarchies instead of interface hierarchies (i.e. BillableItem (abstract), Product, Service should all be classes)

15. Define generic versions of contains(), map() and filter() functions defined earlier.

16. Define a generic pick() function that accepts an array of objects (the actual object type is parameterized), and a "pick key". It should work like so.

```
type Coords = { x: string, y: number };
```

Example: The two inputs to the function - an array and the "pick key" - here 'y'

```
[ { x: "1", y: 2 }, { x: "10", y: 20 }, { x: "100", y: 200 } ],
'y'
```

The returned value would be

```
[ 2, 20, 200 ]
```

17. Define a generic interface IMap that represents a dictionary-like storage, i.e. a Map object should be able to store key-value pairs, and allow addition/removal/access of key-value pairs.

Example, a map of Employee id (key of number type) to corresponding Employee object (value of Employee type).

```
1 -> john
2 -> jane
3 -> mark
```

IMPORTANT: Make sure the key and value types are generic.

The interface defines signatures of 2 methods - get(key) that returns a value type, and set(key, value) that adds a key-value pair to the map.

Now define a generic class ArrayMap that implements the IMap interface (provide constructor and definition for methods). You can use any data structure to store the key-value pairs - for example you may use an array to store keys, and another array to store corresponding values. For example,

* keys may be stored like so [1, 2, 3]
* values may be stored as [john, jane, mark]

Define an ArrayMap<number, Employee> object and add all employees to the map against the employee's id (Employee class as defined in a previous exercise).

ADVANCED TYPESCRIPT CONCEPTS

18. Define separate type-guard functions for the TruckOrPlane union. Use these to refine types within the changeEngine() function.

19. Define a type for the API response from https://api.stackexchange.com/2.0/questions?site=stackoverflow. Key in to this type, to define the type for an Owner of a post. Also define a union type of all possible types for values of properties in Owner.

20. Fill in the blanks with an operator each to get a union of the john object keys

```
let john = {
   name: 'John',
   age: 32
};

type JohnKeys = _________ _________ john;
```

21. Build a type-safe pluck() function - you should not be able to pass a non-existent key to it. It works like so.

```
// Step 1: Make this type-safe
function pluck( o, k ) {
   return k.map( n => o[n] );
}

let john = {
   name: 'John',
   age: 32,
   city: 'San Jose'
};

// call pluck() - generic types are inferred and bound
let nameAndAge = pluck( john, [ 'name', 'age' ] ); // should work
let cityAndCountry = pluck( john, [ 'city', 'country' ] ); // should not work
```

```

**Solution**: https://www.typescriptlang.org/docs/handbook/advanced-types.html#index-types

22. Define an object type to represent permissions available on a file (flags - Read, Write, Execute). Create it as a mapped type using a union of type literals for the flags. Create an object that represents permission to Read and Write (but not Execute).

23. Define an object type to represent permissions available on a file (flags - Read, Write, Execute) for different user roles (roles - Owner, Group, Guest). Use the generic mapped type MyRecord and a union type of role types, along with the type created in the earlier exercise to define it.

24. Define a generic mapped type Nullable that creates a new type from an object type (same fields), but allows null value for each of its fields.

25. Define a generic mapped type Writable that creates a new type from an object type (same fields), but marks each of the fields as writable.

26. Both these types generate a number[] (when applied to number). Is there a difference between these? Make sure you understand how their definitions work.

```
type toArray1<T> = T[];
type toArray2<T> = T extends unknown ? T[] : T[];

let odds : toArray1<number> = [1, 3, 5, 7, 9];
let primes : toArray2<number> = [2, 3, 5, 7, 11];
```

27. Write your own implementation of the buit-in conditional type - Extract<T, U> which computes the types in T that you can assign to U. It should work like so.

```
type X = number | 'hello' | true | [];
type Y = 1 | string | boolean;
type Z = Extract<X, Y>; // Z = 'hello' | true (note that 1 is not part of Z);
```
```