Experiment – 1 a: TypeScript

Name of Student	Mahvish L Siddiqui
Class Roll No	D15A 58
D.O.P.	
D.O.S.	
Sign and Grade	

Experiment – 1 a: TypeScript

1. **Aim:** Write a simple TypeScript program using basic data types (number, string, boolean) and operators.
2. **Problem Statement:**

    a. Create a calculator in TypeScript that uses basic operations like addition, subtraction, multiplication, and division. It also gracefully handles invalid operations and division by zero..

    b. Design a Student Result database management system using TypeScript.

```
// Step 1: Declare basic data types
const studentName: string = "John Doe";
const subject1: number = 45;
const subject2: number = 38;
const subject3: number = 50;

// Step 2: Calculate the average marks
const totalMarks: number = subject1 + subject2 + subject3;
const averageMarks: number = totalMarks / 3;

// Step 3: Determine if the student has passed or failed
const isPassed: boolean = averageMarks >= 40;

// Step 4: Display the result
```

```
console.log(Student Name: ${studentName});
console.log(Average Marks: ${averageMarks});
console.log(Result: ${isPassed ? "Passed" : "Failed"});
```

3. **Theory:**
    a. What are the different data types in TypeScript? What are Type Annotations in Typescript?

    In TypeScript, the primary data types are:

    Primitive Types:

    - Number: Represents both integer and floating-point numbers.
    - String: Represents a sequence of characters.
    - Boolean: Represents true or false values.
    - Null: Represents the intentional absence of any object value.
    - Undefined: Represents a variable that has not been assigned a value.
    - Symbol: Used for creating unique identifiers.
    - BigInt: Used for large integers beyond the safe range of Number.

    Object Types:

    - Object: Represents any non-primitive type.
    - Array: Represents a collection of elements of a specific type.
    - Tuple: Represents an array with a fixed number of elements, where each element can have a different type.
    - Function: Represents a callable block of code.

    Specialized Types:

    - Any: A type that allows any kind of value (essentially opting out of type checking).
    - Unknown: Similar to any, but safer because you must perform some type checking before performing operations.
    - Never: Represents values that never occur (e.g., functions that always throw exceptions or have infinite loops).
    - Void: Represents the absence of a value, often used for functions that don't return a value.

Type annotations in TypeScript

Type annotations explicitly define the type of variables, function parameters, and return values. They ensure type safety by checking that values match the expected type. For example, let x: number = 5; ensures x is always a number.

b. How do you compile TypeScript files?

To compile TypeScript files, you can use the TypeScript compiler (`tsc`), which converts `.ts` files into JavaScript.

Steps to compile TypeScript files:

1. Install TypeScript: First, install TypeScript globally or locally using npm:

   - Globally: `npm install -g typescript`

   - Locally: `npm install --save-dev typescript`

2. Compile a TypeScript File:

   - If you have a TypeScript file named `app.ts`, run: tsc app.ts

   - This generates a JavaScript file `app.js` in the same directory.

3. Compile All TypeScript Files in a Project:

   - If your project has multiple `.ts` files, you can compile them all at once by running `tsc` in the project directory. TypeScript will look for a `tsconfig.json` file (configuration file) to know how to compile the files.

4. Using `tsconfig.json`:

   - You can create a `tsconfig.json` by running `tsc --init`. This file allows you to customize the TypeScript compilation options.

   - To compile all files defined in the `tsconfig.json`, just run: tsc

This will compile the TypeScript files according to the configuration and generate the corresponding JavaScript files.

c. What is the difference between JavaScript and TypeScript?
d. Compare how Javascript and Typescript implement Inheritance.

JavaScript: Uses prototype-based inheritance.

TypeScript: Uses class-based inheritance with the extends keyword.

Example: JavaScript

```
function Animal(name) { this.name = name; }

Animal.prototype.speak = function() { console.log(this.name); };
```

Example TypeScript:

```
class Animal {
  constructor(public name: string) {}
  speak() { console.log(this.name); }
}

class Dog extends Animal { }
```

e. How generics make the code flexible and why we should use generics over other types. In the lab assignment 3, why the usage of generics is more suitable than using any data type to handle the input.

Generics allow functions, classes, and interfaces to work with any data type while maintaining type safety. Using generics avoids the risks associated with any, which doesn't provide type checking, preventing potential runtime errors.

In the lab assignment 3, using generics ensures type safety while handling various input types, unlike any, which skips type checking.

f. What is the difference between Classes and Interfaces in Typescript? Where are interfaces used?

Classes: Define objects with both implementation (methods and properties).

Interfaces: Define the structure (shape) of an object without implementation.

Interfaces Usage: Used to enforce structure in objects, function signatures, and class contracts.

4. **Output:**

Code:

Ts File

```
class Calculator {
  add(a: number, b: number): number {
    return a + b;
  }
 subtract(a: number, b: number): number {
    return a - b;
  }
  multiply(a: number, b: number): number {
    return a * b;
  }
  divide(a: number, b: number): number | string {
    if (b === 0) {
      return "Error: Division by zero is not allowed.";
    }
    return a / b;
  }
```

```
  operate(a: number, b: number, operation: string): number | string {
    switch (operation) {
      case "add":
        return this.add(a, b);
      case "subtract":
        return this.subtract(a, b);
      case "multiply":
        return this.multiply(a, b);
      case "divide":
        return this.divide(a, b);
      default:
        return "Error: Invalid operation.";
    }}}


// Create an instance of the Calculator class
const calc = new Calculator();
console.log(calc.operate(10, 5, "add"));
console.log(calc.operate(10, 5, "subtract"));
console.log(calc.operate(10, 5, "multiply"));
console.log(calc.operate(10, 0, "divide"));
console.log(calc.operate(10, 5, "modulus"));
```

JavaScript:

```
"use strict";
class Calculator {
    add(a, b) {
        return a + b;
    }
    subtract(a, b) {
        return a - b;
    }
    multiply(a, b) {
        return a * b;
    }
    divide(a, b) {
        if (b === 0) {
            return "Error: Division by zero is not allowed.";
        }
        return a / b;
    }
    operate(a, b, operation) {
        switch (operation) {
           case "add":
               return this.add(a, b);
            case "subtract":
```

```
            return this.subtract(a, b);
        case "multiply":
            return this.multiply(a, b);
        case "divide":
            return this.divide(a, b);
        default:
            return "Error: Invalid operation.";
      }
   }
}
// Create an instance of the Calculator class
const calc = new Calculator();

console.log(calc.operate(10, 5, "add"));
console.log(calc.operate(10, 5, "subtract"));
console.log(calc.operate(10, 5, "multiply"));
console.log(calc.operate(10, 0, "divide"));
console.log(calc.operate(10, 5, "modulus"));
```

```
C:\Users\Student\Desktop\webx>node calculator.js
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Quit
Your choice: 1
Enter first number: 23
Enter second number: 12
Result: 35
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Quit
Your choice: 2
Enter first number: 54
Enter second number: 27
Result: 27
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Quit
Your choice: 3
Enter first number: 12
Enter second number: 5
Result: 60
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Quit
Your choice: 4
```

```
Enter first number: 12
Enter second number: 0
Result: Error: Division by zero is not allowed.
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Quit
Your choice: 5
Goodbye!
```

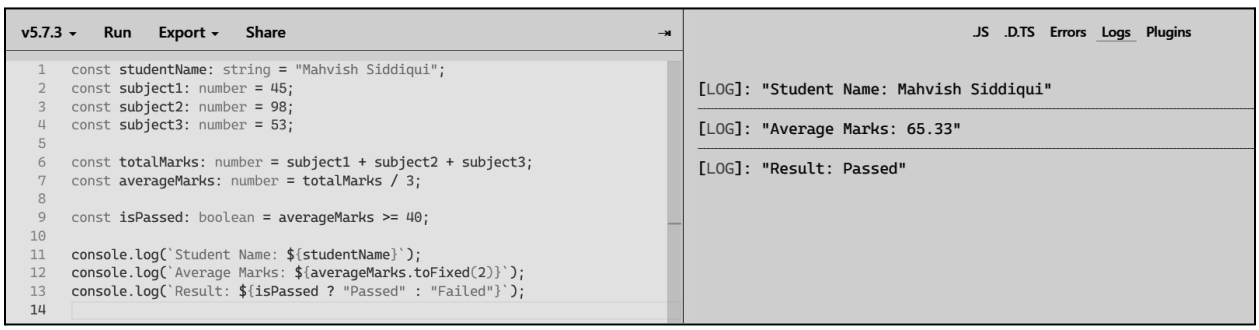
B.
CODE:
Ts File

```
const studentName: string = "Mahvish Siddiqui";
const subject1: number = 45;
const subject2: number = 98;
const subject3: number = 53;

const totalMarks: number = subject1 + subject2 + subject3;
const averageMarks: number = totalMarks / 3;

const isPassed: boolean = averageMarks >= 40;

console.log(`Student Name: ${studentName}`);
console.log(`Average Marks: ${averageMarks.toFixed(2)}`);
console.log(`Result: ${isPassed ? "Passed" : "Failed"}`);
```

Js File

```
  "use strict";
const studentName = "Mahvish Siddiqui";
const subject1 = 45;
const subject2 = 98;
const subject3 = 53;
const totalMarks = subject1 + subject2 + subject3;
const averageMarks = totalMarks / 3;
const isPassed = averageMarks >= 40;
console.log(`Student Name: ${studentName}`);
console.log(`Average Marks: ${averageMarks.toFixed(2)}`);
console.log(`Result: ${isPassed ? "Passed" : "Failed"}`);
```

OUTPUT:



Conclusion:

In this experiment, we successfully created a simple TypeScript program that implements a basic calculator with fundamental operations such as addition, subtraction, multiplication, and division. By utilizing basic data types (number, string, and boolean) and operators, we ensured that the program can handle common arithmetic operations effectively.

Also designed a Student Result Database Management System, showcasing TypeScript's capability to handle structured data and logic. This system provides a robust framework for storing and managing student results.