Experiment – 1 a: TypeScript

Name of Student	Jai Talreja
Class Roll No	59
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, there are several data types that help define the type of a variable. These include:

Primitive types: string, number, boolean, null, undefined, symbol, bigint.
Array: An ordered collection of items, e.g., number[] or Array<number>.
Tuple: An array with fixed size and specific types for each element, e.g., [string, number].
Enum: A way to define named constants, e.g., enum Color { Red, Green, Blue }.
Any: A type that allows any value (but should be used cautiously).
Void: Represents the absence of a value, often used in function return types.
Never: Represents a function that never returns (e.g., throwing an error).
Object: Represents a non-primitive type (excluding null and undefined).

Type Annotations in TypeScript are used to explicitly declare the type of a variable, function parameter, or return type. This allows TypeScript to enforce type safety, providing early error detection at compile-time.

b. How do you compile TypeScript files?

To compile TypeScript files, you first need to install TypeScript globally or as a development dependency via npm. Then, use the tsc command in the terminal to compile .ts files into .js files. You can run tsc filename.ts to compile a single file, or tsc to compile all TypeScript files in the project based on the tsconfig.json configuration file.

c. What is the difference between JavaScript and TypeScript?

JavaScript is a dynamic, loosely-typed programming language used for both front-end and back-end development. TypeScript, on the other hand, is a superset of JavaScript that adds static typing and other advanced features like interfaces, enums, and generics. TypeScript code must be compiled into JavaScript before execution. This compilation process checks for type errors, making it easier to catch bugs before runtime, unlike JavaScript.

d. Compare how JavaScript and TypeScript implement Inheritance.

In JavaScript, inheritance is implemented using prototype chains. Objects can inherit properties and methods from other objects. JavaScript uses the class keyword (introduced in ECMAScript 6) to define classes, but inheritance is still based on prototypes under the hood.

In TypeScript, inheritance is implemented similarly to JavaScript with extends keyword in classes, but TypeScript adds static typing, which ensures that class properties and methods conform to the specified types. TypeScript also supports interfaces for defining contracts that classes can implement, ensuring stricter type safety than JavaScript.

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

Generics in TypeScript allow you to write functions, classes, and interfaces that work with any data type while still maintaining type safety. They enable you to define placeholder types, which will be specified when the function or class is used. This makes code more flexible and reusable across different types, without sacrificing type checking.

Using any negates the benefits of TypeScript's type system because it allows any value without checking the type. Generics, on the other hand, ensure that input types are consistent, which improves code reliability and maintainability. In lab assignment 3, using generics is more appropriate because it guarantees the input type is checked, preventing errors that might arise when using any, which could lead to unexpected behavior.

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

In TypeScript, Classes are blueprints for creating objects, defining properties, methods, and constructors. They can have implementation details, such as method bodies and data initialization. Classes support inheritance using the extends keyword, and can be instantiated to create objects.

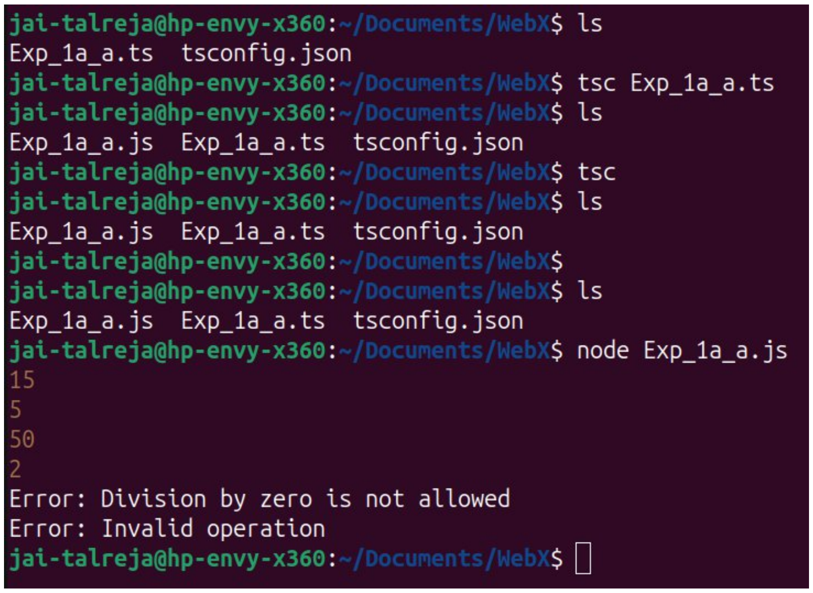
Interfaces, on the other hand, define a contract that classes or objects must adhere to. They don't contain implementation details but specify the structure of properties and methods. Interfaces are used to enforce shape consistency across classes or objects and can be extended or implemented by classes.

Interfaces are commonly used when you need to define the structure of data that multiple classes or objects should follow, ensuring consistency and type safety across your application. They also enable dependency injection and help with testability by providing clear and consistent contracts.

4. **Output:**

[a. Calculator]

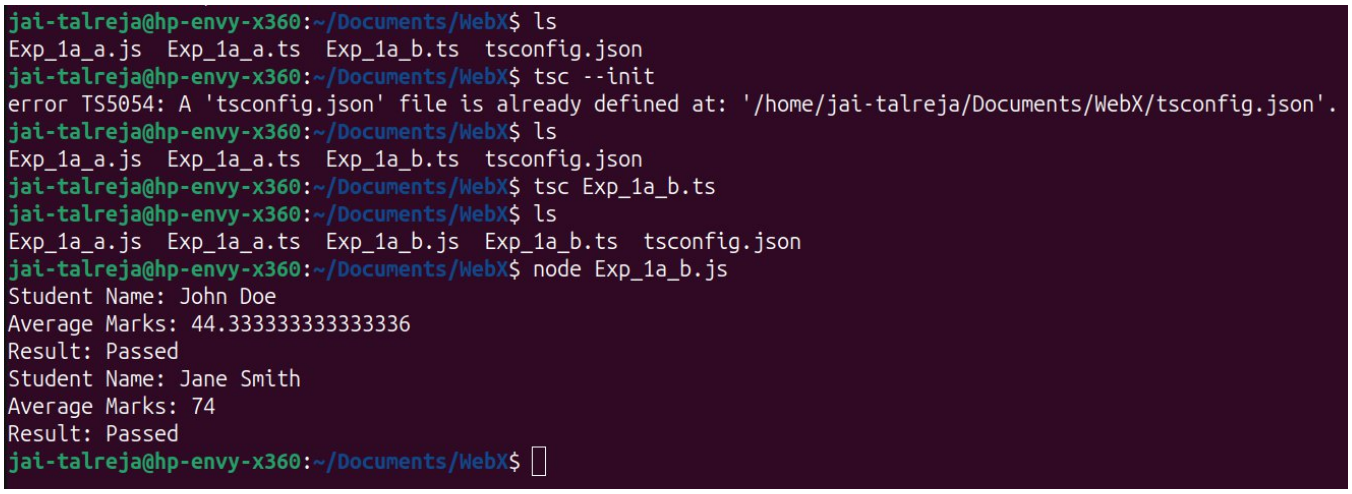
```
class Calculator {
  // Method for addition
  static add(a: number, b: number): number {
    return a + b;
  }

  // Method for subtraction
  static subtract(a: number, b: number): number {
    return a - b;
  }

  // Method for multiplication
  static multiply(a: number, b: number): number {
    return a * b;
  }

  // Method for division with error handling for division by zero
  static divide(a: number, b: number): number | string {
    if (b === 0) {
      return 'Error: Division by zero is not allowed';
    }
    return a / b;
  }

  // Method to handle invalid operation
  static calculate(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';
    }
  }
}

// Example Usage
const num1 = 10;
const num2 = 5;
console.log(Calculator.calculate(num1, num2, 'add')); // 15
console.log(Calculator.calculate(num1, num2, 'subtract')); // 5
console.log(Calculator.calculate(num1, num2, 'multiply')); // 50
console.log(Calculator.calculate(num1, num2, 'divide')); // 2
console.log(Calculator.calculate(num1, 0, 'divide')); // Error: Division by zero is not allowed
console.log(Calculator.calculate(num1, num2, 'modulus')); // Error: Invalid operation
//Program ends here.
```


```
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ tsc Exp_1a_a.ts
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ tsc
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ node Exp_1a_a.js
15
5
50
2
Error: Division by zero is not allowed
Error: Invalid operation
jai-talreja@hp-envy-x360:~/Documents/WebX$
```

[b. Student result Database]

```
class Student {
  studentName: string;
  subjectMarks: number[];

  constructor(studentName: string, subjectMarks: number[]) {
    this.studentName = studentName;
    this.subjectMarks = subjectMarks;
  }

  // Step 3: Calculate the total marks
  getTotalMarks(): number {
    return this.subjectMarks.reduce((acc, mark) => acc + mark, 0);
  }

  // Step 4: Calculate the average marks
  getAverageMarks(): number {
    return this.getTotalMarks() / this.subjectMarks.length;
  }

  // Step 5: Determine if the student has passed or failed
  isPassed(): boolean {
    const average = this.getAverageMarks();
    return average >= 40;
  }

  // Step 6: Display the student result
```

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

// Example usage
const student1 = new Student("John Doe", [45, 38, 50]);
const student2 = new Student("Jane Smith", [75, 82, 65]);

student1.displayResult();
student2.displayResult();
//Program ends here.
```



```
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  Exp_1a_b.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ tsc --init
error TS5054: A 'tsconfig.json' file is already defined at: '/home/jai-talreja/Documents/WebX/tsconfig.json'.
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  Exp_1a_b.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ tsc Exp_1a_b.ts
jai-talreja@hp-envy-x360:~/Documents/WebX$ ls
Exp_1a_a.js  Exp_1a_a.ts  Exp_1a_b.js  Exp_1a_b.ts  tsconfig.json
jai-talreja@hp-envy-x360:~/Documents/WebX$ node Exp_1a_b.js
Student Name: John Doe
Average Marks: 44.333333333333336
Result: Passed
Student Name: Jane Smith
Average Marks: 74
Result: Passed
jai-talreja@hp-envy-x360:~/Documents/WebX$
```