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?
    b. How do you compile TypeScript files?
    c. What is the difference between JavaScript and TypeScript?
    d. Compare how Javascript and Typescript implement Inheritance.
    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.
    f. What is the difference between Classes and Interfaces in Typescript? Where are interfaces used?

a. Data Types in TypeScript and Type Annotations

Data Types in TypeScript: TypeScript offers a range of data types for defining variables and function return types. Some of the main data types in TypeScript are:

Primitive Types:
number: Represents both integer and floating-point numbers.
string: Represents text data.
boolean: Represents true/false values.
void: Represents the absence of a value, commonly used in functions that don't return a value.
null: Represents the intentional absence of any object value.
undefined: Represents a variable that has been declared but not initialized.

Object Types:
object: Represents any non-primitive type.
array: A list of values of a specified type (e.g., number[] or Array<number>).
tuple: Represents an array with a fixed size and fixed types for each element.

Special Types:
any: A type that can hold any value (used when the type is not known ahead of time).
never: Represents a value that will never occur, used for functions that always throw errors or infinite loops.

unknown: Similar to any but safer because you must perform some type-checking before performing operations on it.

Type Annotations in TypeScript: Type annotations are a way of explicitly defining the types of variables, function parameters, and return types in TypeScript. This enables the TypeScript compiler to perform type checking during development, reducing the likelihood of runtime errors.

Example:

```
let age: number = 30; // Type annotation for variable age
function greet(name: string): string { // Type annotations for function parameters and return type
  return "Hello, " + name;
}
```

b. How to Compile TypeScript Files

To compile TypeScript files, you need to use the TypeScript compiler (tsc). Here's how to compile TypeScript files:

Install TypeScript globally (if not already installed):

```
npm install -g typescript
```

Compile a single .ts file:

```
tsc filename.ts
```

Compile multiple .ts files or an entire project by specifying a tsconfig.json file:

```
tsc
```

The tsconfig.json file holds configuration options for compiling TypeScript code, like target JavaScript version, module system, etc.

c. Difference Between JavaScript and TypeScript

Type System:
JavaScript: Dynamically typed, meaning variables can change types during runtime.
TypeScript: Statically typed, allowing types to be explicitly defined and checked at compile time.

Compilation:
JavaScript: Is an interpreted language and does not require compilation.

TypeScript: Needs to be compiled into JavaScript using the TypeScript compiler (tsc).

Support for Classes and Interfaces:

JavaScript: Supports ES6 classes, but lacks interfaces and other advanced features.

TypeScript: Adds support for interfaces, enums, and generics, making it more powerful for large-scale applications.

Tooling:

JavaScript: Has basic support for autocompletion and syntax highlighting in modern editors.

TypeScript: Offers better tooling support, including autocompletion, refactoring, and type checking in IDEs like VSCode.

d. Difference Between JavaScript and TypeScript Implementation of Inheritance

JavaScript: Uses prototype-based inheritance. Objects can inherit from other objects through their prototype chain.

Example:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayHello = function() {
  console.log("Hello, " + this.name);
};

function Dog(name) {
  Animal.call(this, name); // Inherit properties
}
Dog.prototype = Object.create(Animal.prototype); // Inherit methods
```

TypeScript: Supports class-based inheritance with the extends keyword, making inheritance more structured and easier to understand.

Example:

```
class Animal {
  constructor(public name: string) {}
  sayHello() {
    console.log("Hello, " + this.name);
  }
```

```
}

class Dog extends Animal {
  constructor(name: string) {
    super(name); // Call the parent class constructor
  }
}
```

e. Generics in TypeScript and Their Suitability Over any

Generics in TypeScript allow you to write flexible, reusable functions, classes, and interfaces that can work with different types. They provide a way to define a function, class, or interface that works with a variety of data types without losing the benefits of type safety.

Generics make the code flexible by enabling type parameters to be passed, which can be used to define types that can vary.

Example with Generics:

```
function identity<T>(arg: T): T {
  return arg;
}

let result = identity(10);  // result is of type 'number'
let strResult = identity("hello"); // strResult is of type 'string'
```

Why generics are better than using any:

With any, TypeScript loses the benefit of type safety. You could assign any value to the any type, leading to runtime errors.

Generics preserve type safety by ensuring that the type is consistent throughout the code while remaining flexible.

In the context of Lab Assignment 3, using generics would ensure that the input type is preserved, allowing for better type-checking at compile time. Using any would lose this safety, potentially leading to unexpected behaviors during runtime.

f. Difference Between Classes and Interfaces in TypeScript

Classes:

A class is a blueprint for creating objects with shared properties and methods.

Can have constructors, methods, and properties that are implemented.

Can be instantiated to create objects.

Example:

```
class Car {
  constructor(public make: string, public model: string) {}
  drive() {
    console.log(`Driving a ${this.make} ${this.model}`);
  }
}
```

Interfaces:

An interface is a contract that defines the structure of an object, including the properties and methods it should have.
Cannot be instantiated directly; they are used for type-checking.
Interfaces can be implemented by classes to ensure they follow the specified structure.

Example:

```
interface Vehicle {
  make: string;
  model: string;
  drive(): void;
}

class Car implements Vehicle {
  constructor(public make: string, public model: string) {}
  drive() {
    console.log(`Driving a ${this.make} ${this.model}`);
  }
}
```

Where are Interfaces Used?

Interfaces are commonly used for defining contracts for objects or function signatures.
They help in ensuring that a class or an object adheres to a specific structure.
Useful in scenarios where different objects or classes must share a common set of properties or methods.

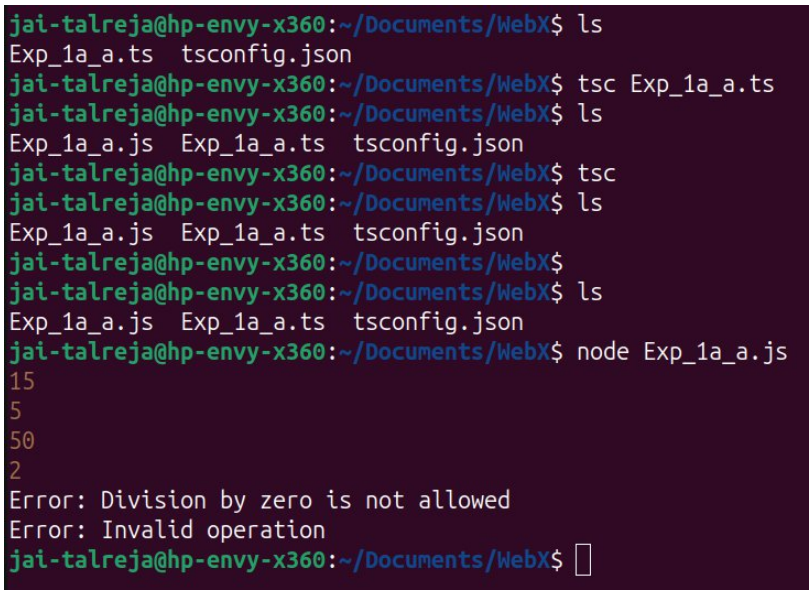
4. **Output:**

[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.
```



[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$
```