Experiment – 1 a: TypeScript

Name of Student	**Prajjwal Pandey**
Class Roll No	**32**
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?

Different Data Types in TypeScript:

1. **Primitive Types:**
   - number – Represents integers and floating-point numbers.
   - string – Represents textual data.
   - boolean – Represents true or false.
   - null – Represents an explicitly empty value.
   - undefined – Represents an uninitialized value.
   - symbol – Represents a unique and immutable primitive value.
   - bigint – Represents large integers beyond number limits.
2. **Object Types:**
   - object – Represents non-primitive values.
3. **Special Types:**
   - any – Allows any type (disables type checking).
   - unknown – Safer alternative to any, requiring type assertion before usage.
   - void – Used for functions that do not return a value.
   - never – Represents functions that never return (e.g., infinite loops or errors).
4. **Array Types:**
   - number[] or Array<number> – Array of numbers.
   - string[] or Array<string> – Array of strings.

5. **Tuple:**
    - [number, string] – A fixed-length array with defined types.
6. **Enum:**
    - enum Direction { Up, Down, Left, Right } – A set of named constants.
7. **Union and Intersection Types:**
    - string | number – A variable can be a string or a number.
    - type Person = { name: string } & { age: number } – Combination of multiple types.

Type Annotations in TypeScript: Type annotations explicitly specify variable, function parameter, and return value types.

Example:

```
let age: number = 25;

let name: string = "Alice";

function add(x: number, y: number): number {

   return x + y;

}
```

---

b. How do you compile TypeScript files?

To compile TypeScript files into JavaScript, use the TypeScript compiler (tsc).

Steps:

1. Install TypeScript (if not installed):
2. npm install -g typescript
3. Compile a single TypeScript file:
4. tsc filename.ts
5. Compile multiple files:
6. tsc file1.ts file2.ts

7. Compile and watch for changes:
8. tsc --watch
9. Compile using a tsconfig.json file:
10. tsc

---

c. What is the difference between JavaScript and TypeScript?

Feature	**JavaScript**	**TypeScript**
Type System	Dynamic typing	Static typing
Compilation	Interpreted	Compiled to JavaScript
Error Checking	Runtime errors	Compile-time errors
Interfaces	Not supported	Supported
Generics	Not supported	Supported
Tooling	Basic support	Advanced support with strong IDE integration

---

d. Compare how JavaScript and TypeScript implement Inheritance.

Both JavaScript and TypeScript use prototypal inheritance and the class syntax.

JavaScript Example:

```
class Animal {

  constructor(name) {

    this.name = name;

  }
  speak() {
    console.log(`${this.name} makes a sound`);
  }
}
class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks`);
  }
}
```

TypeScript Example:

```
class Animal {
  constructor(public name: string) {}
  speak(): void {
    console.log(`${this.name} makes a sound`);
  }
}
class Dog extends Animal {
  speak(): void {
    console.log(`${this.name} barks`);
  }
}
```

TypeScript ensures type safety and allows better structure with public, private, and protected access modifiers.

---

e. How generics make the code flexible and why we should 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: Generics provide type safety while allowing flexibility. They enable reusability and prevent using any, which removes type checking.

Example:

```
function identity<T>(value: T): T {

    return value;

}
```

Why use Generics over any?

- Ensures type safety while allowing different data types.
- Provides better code readability and maintainability.
- Prevents runtime errors due to unexpected types.

Usage in Lab Assignment 3: Using generics instead of any ensures that inputs retain their expected types, reducing errors and improving performance.

---

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

Feature	**Classes**	**Interfaces**
Instantiation	Can create objects	Cannot create objects
Implementation	Defines implementation	Only defines structure
Members	Contains fields & methods	Only contains type definitions
Access Modifiers	Supports public, private, protected	No access modifiers

Example of Class:

class Person {

    constructor(public name: string, public age: number) {}

}

Example of Interface:

interface IPerson {

    name: string;

    age: number;

}

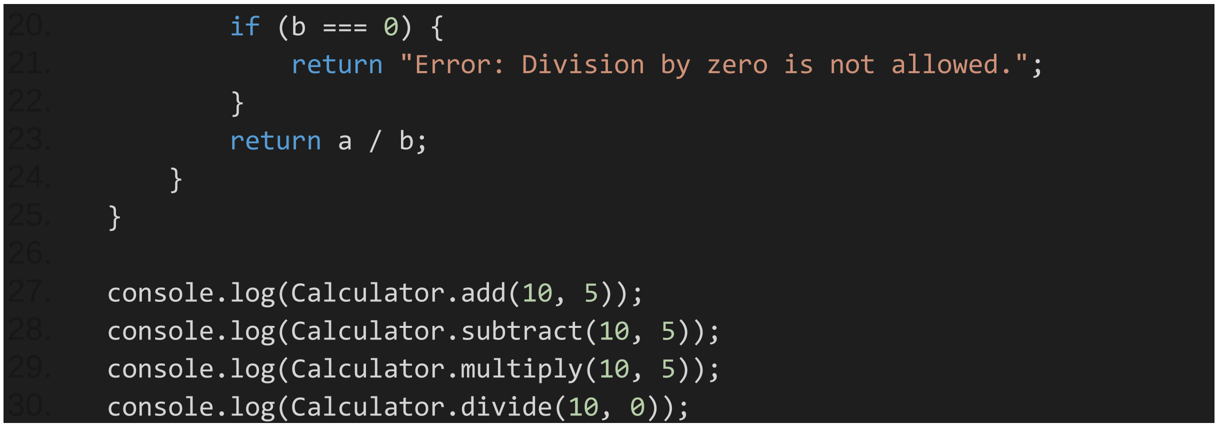
Where are interfaces used?

- Defining object structures.
- Enforcing class contracts.
- Type-checking function parameters.
- Implementing dependency injection in applications.

4. **Output:**

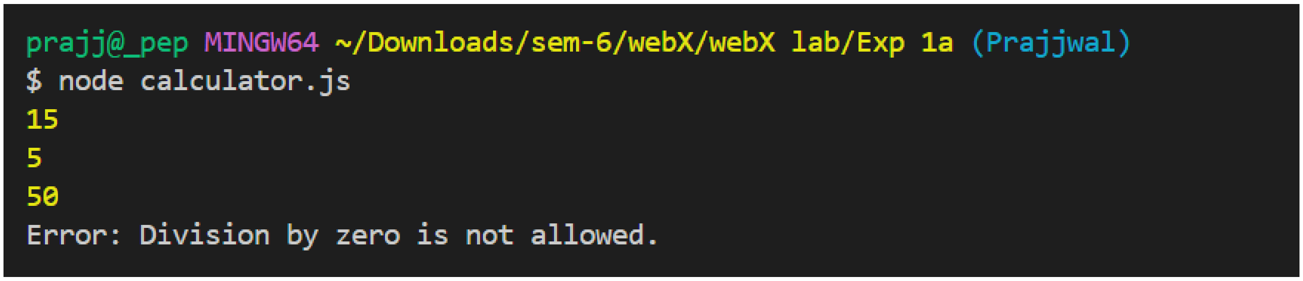
    a. Source code:

```
// Calculator in TypeScript
class Calculator {
    static add(a: number, b: number): number {
        return a + b;
    }

    static subtract(a: number, b: number): number {
        return a - b;
    }

    static multiply(a: number, b: number): number {
        return a * b;
    }

    static divide(a: number, b: number): number | string {
```



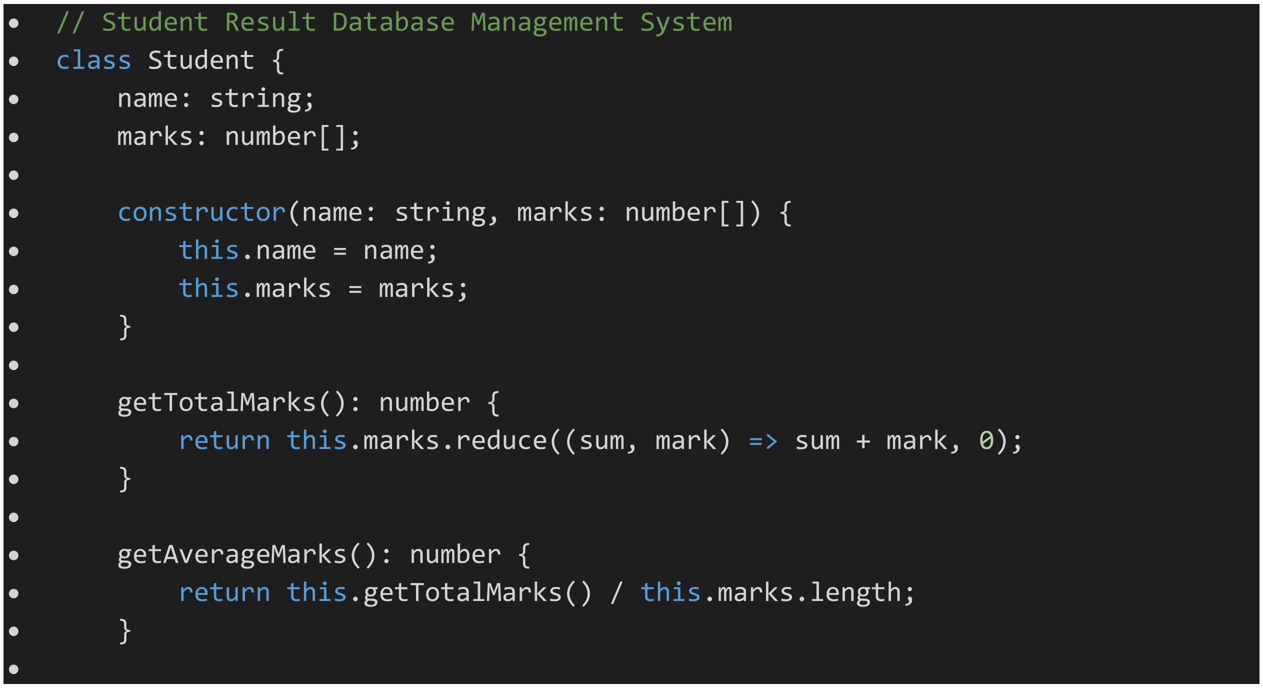
```
        if (b === 0) {
            return "Error: Division by zero is not allowed.";
        }
        return a / b;
    }
}

console.log(Calculator.add(10, 5));
console.log(Calculator.subtract(10, 5));
console.log(Calculator.multiply(10, 5));
console.log(Calculator.divide(10, 0));
```

output:



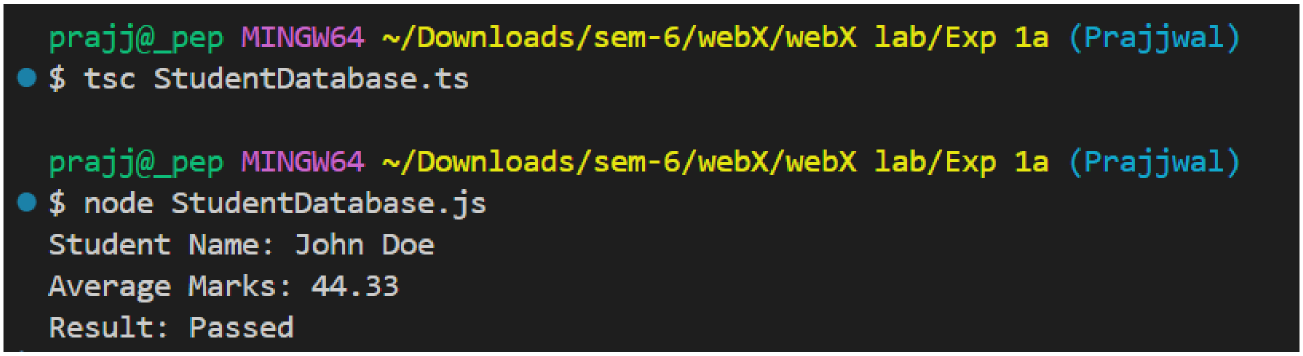
```
prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1a (Prajjwal)
$ node calculator.js
15
5
50
Error: Division by zero is not allowed.
```

b. source code:



```
// Student Result Database Management System
class Student {
    name: string;
    marks: number[];

    constructor(name: string, marks: number[]) {
        this.name = name;
        this.marks = marks;
    }

    getTotalMarks(): number {
        return this.marks.reduce((sum, mark) => sum + mark, 0);
    }

    getAverageMarks(): number {
        return this.getTotalMarks() / this.marks.length;
    }

    hasPassed(): boolean {
        return this.getAverageMarks() >= 40;
    }

    displayResult(): void {
        console.log(`Student Name: ${this.name}`);
        console.log(`Average Marks: ${this.getAverageMarks().toFixed(2)}`);
        console.log(`Result: ${this.hasPassed() ? "Passed" : "Failed"}`);
    }
}

const student = new Student("John Doe", [45, 38, 50]);
student.displayResult();
```

output:



```
prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1a (Prajjwal)
$ tsc StudentDatabase.ts

prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1a (Prajjwal)
$ node StudentDatabase.js
Student Name: John Doe
Average Marks: 44.33
Result: Passed
```