Experiment – 1 a: TypeScript

Name of Student	**MANAV PUNJABI**
Class Roll No	**D15A_44**
D.O.P.	
D.O.S.	
Sign and Grade	

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 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?

1. Different Data Types in TypeScript

Primitive Types: string, number, boolean, bigint, symbol, null, undefined

Object Types: object, array, tuple, enum, function

Special Types: any, unknown, never, void

User-defined Types: interface, class, type alias, union, intersection

2. Type Annotations in TypeScript

Used to explicitly define the data type of variables, function parameters, and return values.

```
let age: number = 25;
function greet(name: string): string {
  return "Hello, " + name;
}
```

3. How to Compile TypeScript Files?

TypeScript code is written in files with the .ts extension. Since browsers and Node.js cannot directly run .ts files, we need to compile them into plain JavaScript (.js) using the **TypeScript Compiler (tsc)**.

Steps to Compile TypeScript:

1. Basic Compilation
   Compiles a single TypeScript file into JavaScript.
   tsc filename.ts
   ➢ Output will be filename.js in the same folder.
2. **Watch Mode (Auto Compile on Save)**
   Keeps watching the file and automatically recompiles it on changes.
   tsc filename.ts –watch
3. **Compile All Files in the Project**
   If you have multiple .ts files and a tsconfig.json configuration file, this command compiles all files at once.
   tsc

4. Difference Between JavaScript and TypeScript

Feature	**JavaScript**	**TypeScript**
Typing	Dynamically typed	Statically typed
Compilation	No compilation	Compiles to JavaScript
OOP Support	Limited	Strong OOP support
Error Handling	Runtime errors	Compile-time errors
Maintainability	Less structured	More structured

5. Inheritance Example:

JavaScript:

```
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:

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

6. Generics in TypeScript:

- **Function Example:**

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

Why Generics over any in Lab 3?
Generics preserve type safety and make the code reusable and maintainable.

7. Classes vs Interfaces:

Feature	Class	Interface
Definition	Blueprint for objects	Structure only, no implementation
Implementation	Has logic	Only structure
Inheritance	Single inheritance	Can extend multiple interfaces
Usage	Object creation	Type checking

Where Are Interfaces Used?

- **To define object shapes:**

```
interface User {
  name: string;
  age: number;
}
let user: User = { name: "Alice", age: 25 };
```

- To implement contracts in classes:

```
interface Animal {
  speak(): void;
}
class Dog implements Animal {
  speak(): void {
    console.log("Bark!");
  }
}
```

- For function types:

```
interface MathOperation {
  (x: number, y: number): number;
}
let add: MathOperation = (a, b) => a + b;
```

5. Output
   a. Calculator Code:

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

  calculate(operator: string, a: number, b: number): number | string {
    try {
      switch (operator) {
        case "+": return this.add(a, b);
        case "-": return this.subtract(a, b);
        case "*": return this.multiply(a,
        b); case "/": return this.divide(a,
        b);
        default: return "Invalid operator. Please use +, -, *, or /.";
      }
    } catch (error: any) {
      return error.message;
    }
  }
}

const calculator = new Calculator();
console.log(calculator.calculate("+", 5, 3));
console.log(calculator.calculate("-", 10, 4));
console.log(calculator.calculate("*", 2, 6));
console.log(calculator.calculate("/", 10, 2));
console.log(calculator.calculate("/", 10, 0));
console.log(calculator.calculate("%", 5, 2));
```

b. Student Result System Code:

```
interface Subject {
  name: string;
  marks: number;
}

interface Student {
  id: string;
  name: string;
  subjects: Subject[];
}

interface Result {
  studentId: string;
  averageMarks: number;
  isPassed: boolean;
}

function calculateAverage(subjects: Subject[]): number {
  if (subjects.length === 0) return 0;
  const totalMarks = subjects.reduce((sum, subject) => sum + subject.marks, 0);
  return totalMarks / subjects.length;
}

function determinePassStatus(averageMarks: number, passingThreshold: number = 40): boolean {
  return averageMarks >= passingThreshold;
}

function generateResult(student: Student): Result {
  const averageMarks = calculateAverage(student.subjects);
  const isPassed = determinePassStatus(averageMarks);
  return {
    studentId: student.id,
    averageMarks,
    isPassed,
  };
}

function displayResult(student: Student, result: Result): void {
  console.log(`Student ID: ${student.id}`);
  console.log(`Student Name: ${student.name}`);
  student.subjects.forEach(subject => {
    console.log(` ${subject.name}: ${subject.marks}`);
  });
  console.log(`Average Marks: ${result.averageMarks.toFixed(2)}`);
  console.log(`Result: ${result.isPassed ? "Passed" : "Failed"}`);
  console.log("---");
```

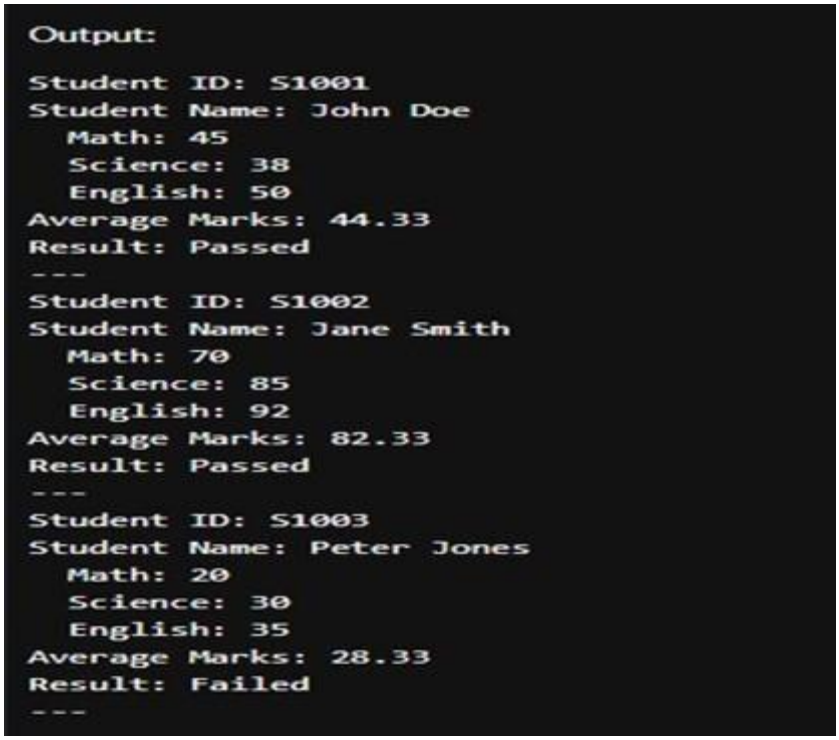
```
}

// Sample Students
const student1: Student = {
  id: "S1001",
  name: "John Doe",
  subjects: [
    { name: "Math", marks: 45 },
    { name: "Science", marks: 38 },
    { name: "English", marks: 50 }
  ]
};

const student2: Student = {
  id: "S1002",
  name: "Mary Smith",
  subjects: [
    { name: "Math", marks: 70 },
    { name: "Science", marks: 85 },
    { name: "English", marks: 92 }
  ]
};

const student3: Student = {
  id: "S1003",
  name: "Peter Jones",
  subjects: [
    { name: "Math", marks: 20 },
    { name: "Science", marks: 30 },
    { name: "English", marks: 35 }
  ]
};

displayResult(student1, generateResult(student1));
displayResult(student2, generateResult(student2));
displayResult(student3, generateResult(student3));
```



Conclusion:

Compiling TypeScript files is an essential step to convert modern TypeScript code into JavaScript that browsers and Node.js can understand. With simple tsc commands, developers can efficiently compile individual files or entire projects, making development faster and more manageable.