Experiment – 1 a: TypeScript

Name of Student	Pranav Pramod Titambe
Class Roll No	60
D.O.P.	06.02.2025
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.

3.Theory:

a. What are the different data types in TypeScript? What are Type Annotations in Typescript?

Different Data Types in TypeScript & Type Annotations

TypeScript supports various data types:

1. Primitive Types: number, string, boolean, null, undefined, symbol, bigint.
2. Object Types: arrays, tuples, enums, classes, interfaces.
3. Special Types: any, unknown, void, never.

Type Annotations in TypeScript allow specifying the expected data type of a variable, ensuring type safety. Example:

```
let age: number = 25;

let name: string = "Alice";

let isStudent: boolean = true;
```

b.What is the difference between JavaScript and TypeScript?

1. **Type Safety**:
    - **JavaScript**: Dynamically typed (no strict types).
    - **TypeScript**: Statically typed (types are explicitly defined).
2. **Compilation**:
    - **JavaScript**: Interpreted directly by browsers.
    - **TypeScript**: Compiled to JavaScript before execution.
3. **Code Maintainability**:
    - **JavaScript**: Can be difficult to maintain in large projects.
    - **TypeScript**: Easier to manage due to strong typing and error checking.
4. **Object-Oriented Features**:
    - **JavaScript**: Uses prototypes for inheritance.
    - **TypeScript**: Supports classes, interfaces, and generics.
5. **Error Detection**:
    - **JavaScript**: Errors occur at runtime.
    - **TypeScript**: Errors are caught at compile-time.

c. Compare how Javascript and Typescript implement Inheritance.

JavaScript: Uses prototype-based inheritance.

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = function() {
    console.log(`Hello, my name is ${this.name}`);
};
let john = new Person("John");
john.greet();
```

TypeScript: Uses class-based inheritance.

```
class Person {
    constructor(public name: string) {}
    greet() {
    console.log(`Hello, my name is ${this.name}`);
    }
}
class Student extends Person {
    constructor(name: string, public grade: number) {
    super(name);
    }
}
let student = new Student("Alice", 10);
student.greet();
```

d. 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 reusable and type-safe functions/classes without losing type information.

Without Generics (any type):

```
function identity(value: any): any {
    return value;
}
let result = identity("Hello"); // No type enforcement
```

With Generics:

```
function identity<T>(value: T): T {
    return value;
}
let result = identity<string>("Hello"); // Ensures type safety
```

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

Definition:

- **Class**: Blueprint for creating objects with methods and properties.
- **Interface**: Defines a structure without implementation.

Implementation:

- **Class**: Can be instantiated.
- **Interface**: Only defines properties and method signatures.

Usage:

- **Classes**: Used for creating objects and defining behavior.
- **Interfaces**: Used for enforcing object structure in applications.

```
interface Animal {
    name: string;
    makeSound(): void;
}
class Dog implements Animal {
    name: string;
    constructor(name: string) {
    this.name = name;
    }
    makeSound() { console.log("Woof!"); }
}
```

f. How do you compile TypeScript files?

To compile a TypeScript file (file.ts) into JavaScript:

Install TypeScript (if not installed):

```
nginx
npm install -g typescript
```

1. Compile the file:
   ```
   nginx
   tsc file.ts
   ```
2. This generates a file.js which can be run in browsers or Node.js.

3. OUTPUT:

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..

```
function calculator(a: number, b: number, operation: string): number | string {
    switch (operation.toLowerCase()) {
```

```
        case "add":
        case "+":
        return a + b;
        case "subtract":
        case "-":
        return a - b;
        case "multiply":
        case "*":
        return a * b;
        case "divide":
        case "/":
        return b !== 0 ? a / b : "Error: Division by zero is not allowed.";
        default:
        return "Error: Invalid operation.";
        }
}
console.log(`Addition: ${calculator(10, 5, "add")}`);
console.log(`Subtraction: ${calculator(10, 5, "subtract")}`);
console.log(`Multiplication: ${calculator(10, 5, "multiply")}`);
console.log(`Division: ${calculator(10, 5, "divide")}`);
console.log(`Division by zero: ${calculator(10, 0, "divide")}`);
console.log(`Invalid operation: ${calculator(10, 5, "modulus")}`);
```

OUTPUT:

```
[LOG]: "Addition: 15"
[LOG]: "Subtraction: 5"
[LOG]: "Multiplication: 50"
[LOG]: "Division: 2"
[LOG]: "Division by zero:
Error: Division by zero is not
allowed."
```

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

```
interface Student {
        id: number;
        name: string;
        marks: { [subject: string]: number };
}
class StudentDatabase {
        private students: Student[] = [];
  // Add a new student
        addStudent(id: number, name: string, marks: { [subject: string]: number }): void {
        this.students.push({ id, name, marks });
        }
        getStudent(id: number): Student | string {
        const student = this.students.find(s => s.id === id);
        return student ? student : "Student not found.";
```

```
        }
        calculateAverage(id: number): number | string {
        const student = this.getStudent(id);
        if (typeof student === "string") return student;

        const totalMarks = Object.values(student.marks).reduce((sum, mark) => sum + mark, 0);
        return totalMarks / Object.keys(student.marks).length;
        }
        displayStudent(id: number): void {
        const student = this.getStudent(id);
        if (typeof student === "string") {
        console.log(student);
        return;
        }
        console.log(`ID: ${student.id}, Name: ${student.name}`);
        console.log("Marks:", student.marks);
        console.log(`Average Marks: ${this.calculateAverage(id)}`);
        }
}
const db = new StudentDatabase();
db.addStudent(1, "Pranav", { Math: 85, Science: 90, English: 88 });
db.addStudent(2, "Bharati", { Math: 78, Science: 80, English: 74 });
db.displayStudent(1);
```

db.displayStudent(2);

db.displayStudent(3); // Non-existing student

OUTPUT:

```
[LOG]: "ID: 1, Name: Pranav"
[LOG]: "Marks:",  {
  "Math": 85,
  "Science": 90,
  "English": 88
}
[LOG]: "Average Marks: 87.66666666666667"
[LOG]: "ID: 2, Name: Bharati"
[LOG]: "Marks:",  {
  "Math": 78,
  "Science": 80,
  "English": 74
}
[LOG]: "Average Marks: 77.33333333333333"
[LOG]: "Student not found."
```