## WEBX EXP 1A

**Aim:** Write a simple TypeScript program using basic data types (number, string, boolean) and operators.

**Theory:**

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

TypeScript provides several built-in data types, including primitive types like string, number, boolean, null, undefined, symbol, and bigint. It also includes object types such as Array, Tuple, Enum, Class, and Interface. Additionally, TypeScript has special types like any, unknown, void, and never.

Type annotations in TypeScript allow developers to explicitly define the type of a variable, function parameter, or return value. For example, let age: number = 25; ensures that age can only hold numeric values, reducing runtime errors and improving code clarity

**b.How do you compile TypeScript files?**

To compile a TypeScript file, use the TypeScript compiler (tsc) by running the command tsc filename.ts in the terminal. This command translates the TypeScript code into JavaScript, generating a filename.js file that can be executed in a browser or Node.js environment.

**c.What is the difference between JavaScript and TypeScript?**

JavaScript is a dynamically typed language, whereas TypeScript is statically typed, which helps catch errors at compile time. JavaScript does not support interfaces or strong type checking, while TypeScript allows defining interfaces for better structure. Additionally, JavaScript is interpreted, whereas TypeScript must be compiled into JavaScript before execution. TypeScript also provides modern ES features along with additional enhancements like generics and type annotations, making it more robust for large-scale applications.

**d.Compare how Javascript and Typescript implement Inheritance.**

JavaScript uses prototype-based inheritance, where objects inherit properties and methods from other objects through the prototype chain. TypeScript, on the other hand, supports class-based inheritance using the class and extends keywords, making it more structured and similar to object-oriented programming in languages like Java or C#. This

makes TypeScript more readable and maintainable compared to JavaScript's traditional prototype-based approach.

**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.**

Generics provide type safety by ensuring that functions and data structures work with a specific type rather than accepting any type, as the any keyword does. This prevents runtime errors and improves code reusability while maintaining strict type rules. For example, a function using generics like function identity<T>(value: T): T { return value; } ensures that it always returns the same type that it receives as input. In Lab Assignment 3, using generics is more suitable than using any because it ensures that the input data type remains consistent while still allowing flexibility.

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

Classes in TypeScript define both the structure and implementation of an object, meaning they can have methods and properties, and they support instantiation. Interfaces, however, only define the structure without providing any implementation and cannot be instantiated. Interfaces are primarily used to define contracts for objects, ensuring that they have specific properties and methods. For example, an interface Person { name: string; age: number; } can be implemented by multiple classes to ensure they follow the same structure.

**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.**
**Code:**
```
var Calculator = /** @class */ (function () {
    function Calculator() {
    }
    Calculator.add = function (a, b) {
        return a + b;
    };
    Calculator.subtract = function (a, b) {
        return a - b;
    };
    Calculator.multiply = function (a, b) {
        return a * b;
    };
    Calculator.divide = function (a, b) {
```

```
        if (b === 0) {
            return "Error: Division by zero is not allowed";
        }
        return a / b;
    };
    Calculator.calculate = function (operation, a, b) {
        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";
        }
    };
    return Calculator;
}());
// Example usage:
console.log(Calculator.calculate('add', 10, 5)); // 15
console.log(Calculator.calculate('subtract', 10, 5)); // 5
console.log(Calculator.calculate('multiply', 10, 5)); // 50
console.log(Calculator.calculate('divide', 10, 5)); // 2
console.log(Calculator.calculate('divide', 10, 0)); // Error: Division by zero is not allowed
console.log(Calculator.calculate('modulo', 10, 5)); // Error: Invalid operation
```

**Output:**

```
Ria Chaudhari@Ria MINGW64 ~/Ria/typescript-calculator (master)
$ node calculator.js
15
5
50
2
Error: Division by zero is not allowed
Error: Invalid operation
```

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

```typescript
class Student {

  constructor(

    public id: number,

    public name: string,

    public marks: Record<string, number>

  ) {}

      getTotalMarks(): number {

      return Object.keys(this.marks).map(key => this.marks[key]).reduce((sum, mark) => sum + mark, 0);

  }



  getAverageMarks(): number {

    return this.getTotalMarks() / Object.keys(this.marks).length;

  }



  getGrade(): string {

    const avg = this.getAverageMarks();

    if (avg >= 90) return 'A+';

    if (avg >= 80) return 'A';

    if (avg >= 70) return 'B';

    if (avg >= 60) return 'C';

    return 'F';

  }

}
```

```typescript
class StudentDatabase {

  private students: Student[] = [];

  addStudent(student: Student): void {

    this.students.push(student);

  }

  getStudentById(id: number): Student | undefined {

    for (const student of this.students) {

      if (student.id === id) return student;

    }

    return undefined;

  }

  getAllStudents(): Student[] {

    return this.students;

  }

}

const db = new StudentDatabase();

db.addStudent(new Student(1, "Alice", { Math: 85, English: 90, Science: 80 }));

db.addStudent(new Student(2, "Bob", { Math: 70, English: 75, Science: 65 }));

db.addStudent(new Student(3, "Charlie", { Math: 95, English: 85, Science: 88 }));

db.addStudent(new Student(4, "David", { Math: 60, English: 65, Science: 70 }));

db.addStudent(new Student(5, "Eve", { Math: 78, English: 80, Science: 82 }));

db.addStudent(new Student(6, "Frank", { Math: 88, English: 92, Science: 85 }));
```

```
db.addStudent(new Student(7, "Grace", { Math: 55, English: 60, Science: 58 }));

db.addStudent(new Student(8, "Hank", { Math: 40, English: 45, Science: 50 }));

db.addStudent(new Student(9, "Ivy", { Math: 82, English: 88, Science: 86 }));

db.addStudent(new Student(10, "Jack", { Math: 91, English: 94, Science: 89 }));


db.getAllStudents().forEach(student => {

        console.log(`${student.name}  -  Total:  ${student.getTotalMarks()},  Average:
${student.getAverageMarks().toFixed(2)}, Grade: ${student.getGrade()}`);

});
```

**Output:**

```
Ria Chaudhari@Ria MINGW64 ~/Ria/student-db (master)
$ node student.js
Alice - Total: 255, Average: 85.00, Grade: A
Bob - Total: 210, Average: 70.00, Grade: B
Charlie - Total: 268, Average: 89.33, Grade: A
David - Total: 195, Average: 65.00, Grade: C
Eve - Total: 240, Average: 80.00, Grade: A
Frank - Total: 265, Average: 88.33, Grade: A
Grace - Total: 173, Average: 57.67, Grade: F
Hank - Total: 135, Average: 45.00, Grade: F
Ivy - Total: 256, Average: 85.33, Grade: A
Jack - Total: 274, Average: 91.33, Grade: A+
```