

Experiment – 1 a: TypeScript

Name of Student	Sneha Patra
Class Roll No	D15A 40
D.O.P.	<u>23/01/2025</u>
D.O.S.	<u>30/01/2025</u>
Sign and Grade	

Experiment – 1 a: TypeScript

Aim:

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

Problem Statement:

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

Theory:

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.

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.

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.

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.

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.

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.

GitHub Link: https://github.com/Sneha0321/WebX_Exp1

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..

Code-

```
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";  
  
        }  
  
    }  
}
```

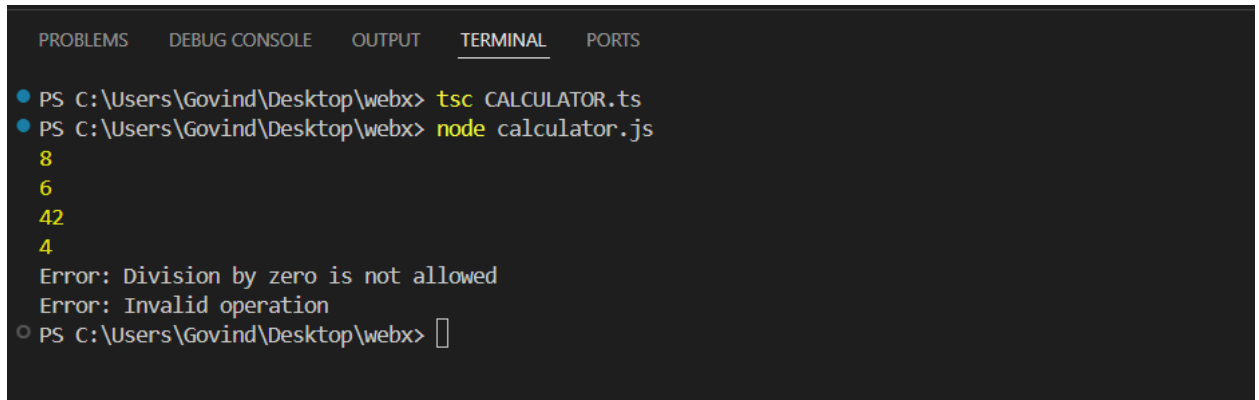
```
    }  
    return a / b;  
}  
  
static calculate(operation: string, a: number, b: number): number | string {  
    switch (operation) {  
        case 'add':  
            return this.add(a, b);  
        case 'subtract':  
            return this.subtract(a, b);  
        case 'multiply':  
            return this.multiply(a, b);  
        case 'divide':  
            return this.divide(a, b);  
        default:  
            return "Error: Invalid operation";  
    }  
}  
}
```

// Example usage

```
console.log(Calculator.calculate('add', 5, 3));  
console.log(Calculator.calculate('subtract', 10, 4));  
console.log(Calculator.calculate('multiply', 6, 7));  
console.log(Calculator.calculate('divide', 8, 2));
```

```
console.log(Calculator.calculate('divide', 5, 0));  
console.log(Calculator.calculate('modulus', 5, 2));
```

Output Screenshot:



```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS  
● PS C:\Users\Govind\Desktop\webx> tsc CALCULATOR.ts  
● PS C:\Users\Govind\Desktop\webx> node calculator.js  
8  
6  
42  
4  
Error: Division by zero is not allowed  
Error: Invalid operation  
○ PS C:\Users\Govind\Desktop\webx> █
```

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

Code:

```
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;  
  }  
  
  getResult(): string {  
    return this.getAverageMarks() >= 40 ? "Passed" : "Failed";  
  }  
}
```

```

displayResult(): void {
    console.log(`Student Name: ${this.name}`);
    console.log(`Average Marks: ${this.getAverageMarks().toFixed(2)}`);
    console.log(`Result: ${this.getResult()}`);
}
}

```

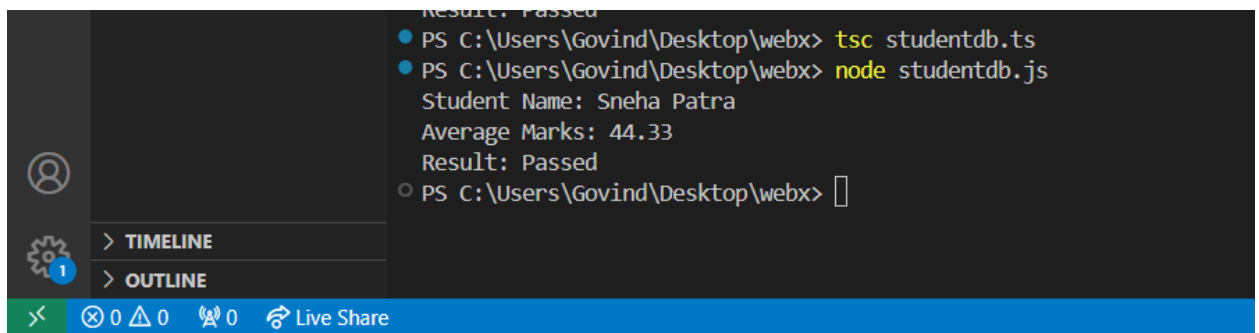
// Example usage

```

const student1 = new Student("Sneha Patra", [45, 38, 50]);
student1.displayResult();

```

Output Screenshot:



```

PS C:\Users\Govind\Desktop\webx> tsc studentdb.ts
PS C:\Users\Govind\Desktop\webx> node studentdb.js
Student Name: Sneha Patra
Average Marks: 44.33
Result: Passed
PS C:\Users\Govind\Desktop\webx>

```

Conclusion

In this experiment, we successfully implemented a simple calculator using TypeScript with proper error handling for invalid operations and division by zero. Additionally, we developed a student result management system that calculates total marks, average marks, and evaluates the result based on the average score. The experiment demonstrated the effectiveness of TypeScript's type safety, object-oriented features, and error-handling capabilities.