Experiment – 1 a: TypeScript

Name of Student	**Riya Varyani**
Class Roll No	**D15A 61**
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:**

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) {
        case "add":
            return a + b;
        case "subtract":
            return a - b;
        case "multiply":
            return a * b;
        case "divide":
            return b !== 0 ? a / b : "Error: Division by zero";
        default:
            return "Error: Invalid operation";
    }
}

// Test cases
console.log(calculator(10, 5, "add"));       // Output: 15
```

```
console.log(calculator(10, 5, "subtract"));  // Output: 5
console.log(calculator(10, 5, "multiply"));  // Output: 50
console.log(calculator(10, 0, "divide"));    // Output: Error: Division by zero
console.log(calculator(10, 5, "modulus"));   // Output: Error: Invalid operation
```

Output:

```
15
5
50
Error: Division by zero
Error: Invalid operation
```

3. 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 total and 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(`Total Marks: ${totalMarks}`);
console.log(`Average Marks: ${averageMarks.toFixed(2)}`);
console.log(`Result: ${isPassed ? "Passed" : "Failed"}`);
```

Output:

```
Student Name: John Doe
Total Marks: 133
Average Marks: 44.33
Result: Passed
```

Theory:

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

TypeScript provides various built-in data types to enforce type safety in programs. These are:

4. **number**: Used for numeric values (integers and floating-point numbers).
   Example:
   let age: number = 25;
   let price: number = 99.99;
5. **string**: Represents textual data.
   Example:
   let name: string = "Alice";
   let message: string = Hello, ${name}!;
6. **boolean**: Represents true or false values.
   Example:
   let isActive: boolean = true;
   let hasPermission: boolean = false;
7. **array**: A collection of values of the same type.
   Example:
   let numbers: number[] = [1, 2, 3, 4, 5];
   let names: string[] = ["Alice", "Bob", "Charlie"];
8. **tuple**: An array with a fixed number of elements, where each element can have a different type.
   Example:
   let person: [string, number] = ["Alice", 25];
9. **enum**: A way to define a set of named constants.
   Example:
   enum Color { Red, Green, Blue }
   let favoriteColor: Color = Color.Green;

10. **any**: Allows a variable to hold values of any type (not recommended unless necessary).
    Example:
    let randomValue: any = 42;
    randomValue = "Hello"; // No error
11. **void**: Used for functions that do not return a value.
    Example:
    function logMessage(): void { console.log("This is a message."); }
12. **null and undefined**: Represent the absence of a value.
    Example:
    let nothing: null = null;
    let notDefined: undefined = undefined;
13. **object**: Represents complex data structures.
    Example:
    let student: { name: string; age: number } = { name: "Alice", age: 20 };
14. **never**: Represents values that never occur (e.g., functions that always throw errors).
    Example:
    function throwError(message: string): never { throw new Error(message); }

Type Annotations

Type annotations in TypeScript allow developers to explicitly specify the type of variables, function parameters, and return values.

let username: string = "John"; // Variable annotation

let score: number = 90; // Variable annotation

function add(a: number, b: number): number { // Function parameter and return type annotation

return a + b; }

2. How do you compile TypeScript files?

To compile TypeScript files, first install TypeScript globally by running npm install -g typescript in your terminal. Then, you can compile a single .ts file using tsc file-name.ts, which will generate a .js file (e.g., file-name.js). To compile multiple files, list them like tsc file1.ts file2.ts. For larger projects, it's recommended to use a tsconfig.json file to configure TypeScript compilation. Create it by running tsc --init, and then you can

compile all files in the project by simply running tsc, which will automatically use the settings in the tsconfig.json.

3. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Typing	Dynamically typed (no ty checking)	Statically typed (with type checking)
Compilation	Interpreted (runs directly)	Compiled to JavaScript before running
Type Inference	No type inference	Automatic type inference (can explicit or inferred)
Features Support	Supports modern JavaScr (ES6+)	Supports modern JavaScript (ES6- with optional extra features
Classes ar Interfaces	Classes are supported, but interfaces	Supports classes, interfaces, a advanced features like access modifie
Tooling & ID Support	Basic supp (autocompletion, debugging	Advanced support (autocompletio error checking, type validation)
Error Detection	Errors detected at runtime	Errors detected at compile-time due type checking
Development Spe	Faster (no need compilation)	Slower (requires compilation step)

Adoption for Lar Projects	Can become hard to mana as projects grow	Better suited for large projects due strong typing and maintainability

4. Compare how Javascript and Typescript implement Inheritance.

Aspect	JavaScript	TypeScript
Syntax	class and extends keywor (ES6)	Same as JavaScript, with ty annotations
Constructor	constructor method to initialize	Same, with optional typ parameters
Method Overridin	Overriding methods witho restriction	Methods can be overridden with ty safety
Access Modifiers	No built-in access control	Supports public, private, protected
Type Safety	No type checking	Type safety ensures correct types

In both languages, inheritance is done using class and extends, but TypeScript adds type safety and access modifiers for more structured and error-free inheritance.

5.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 you to write functions and classes that work with any data type while keeping type safety. This means you can reuse the same code for different types without errors.

Why Use Generics Over any:

1. **Type Safety**: Generics ensure the correct type is used, avoiding runtime errors.
2. **Reusability**: The same code works for different types without duplicating logic.
3. **Better Clarity**: Generics show that the code can handle various types safely, while any skips type checking and can lead to mistakes.

Why Generics Are Better Than any in Lab Assignment 3:

Generics are better because they provide type safety, reduce errors, and make the code more flexible and maintainable. Using any can lead to unexpected problems since it doesn't check types.

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

Difference Between Classes and Interfaces in TypeScript:

- **Class**: Defines both the structure (properties) and behavior (methods) of objects. Can be instantiated and have method implementations.
- **Interface**: Defines the structure (properties and method signatures) of objects, but does not provide method implementations. Used for type-checking.

Where Are Interfaces Used?

- **Object Structure**: To define the shape of an object.
- **Function Signatures**: To define types for function parameters and return values.
- **Class Contracts**: To ensure classes follow a specific structure (using implements).

Example:

typescript

CopyEdit

```
interface Animal {

  name: string;

  speak(): void;

}
```

```
class Dog implements Animal {
  constructor(public name: string) {}
  speak() {
    console.log(`${this.name} barks`);
  }
}

const dog = new Dog("Buddy");
dog.speak();  // Output: Buddy barks
```

In short, **classes** define both structure and behavior, while **interfaces** only define structure and are used for type-checking.

Conclusion

This experiment introduced fundamental **TypeScript concepts** such as **basic data types, operators, functions, interfaces, classes, and generics**. We implemented a **calculator** to perform arithmetic operations and a **student result management system** to demonstrate data handling and decision-making.

Through theoretical exploration, we compared **TypeScript with JavaScript**, studied **inheritance, compilation, and type safety**, and understood the importance of **generics** for writing flexible and reusable code. Overall, this experiment reinforced the benefits of **TypeScript's static typing, structured object-oriented programming, and enhanced error detection**, making it a powerful tool for scalable development.