Experiment – 1 a: TypeScript

Name of Student	**Ronak Katariya**
Class Roll No	**D15A 23**
D.O.P.	
D.O.S.	
Sign and Grade	

Experiment – 1 a: TypeScript

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

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 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"});

Theory:

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

   TypeScript, being a superset of JavaScript, supports the same basic data types as JavaScript, but it also adds some additional types to make type-checking more robust.

   Primitive Types:

b. `number`: Represents both integer and floating-point numbers. (e.g., `let age: number = 25;`)

c. `string`: Represents textual data. (e.g., `let name: string = "John";`)

d. `boolean`: Represents a true or false value. (e.g., `let isActive: boolean = true;`)

e. `null`: Represents an intentional absence of value. (e.g., `let value: null = null;`)

f. `undefined`: Represents a variable that has not been assigned a value. (e.g., `let unassigned: undefined;`)

g. `symbol`: Represents a unique identifier. (e.g., `let id: symbol = Symbol('id');`)

h. `bigint`: Represents large integers (e.g., `let largeNumber: bigint =`

`1234567890123456789012345678901234567890n;`)

Special Types:

i. `any`: Represents any value, similar to JavaScript's loosely-typed system. (e.g., `let randomValue: any = "Hello";`)

j. `unknown`: Similar to `any`, but safer because you must first check its type before performing operations on it.

k. `never`: Represents a value that never occurs. It is often used for functions that never return (e.g., throwing an error).

Arrays and Tuples:

l. Arrays: Represented by `Array<type>` or `type[]`. (e.g., `let numbers: number[] = [1, 2, 3];`)

m. Tuples: Fixed-length arrays where each element can have a different type. (e.g., `let tuple: [string, number] = ["Alice", 25];`)

Enums:

n. Enum: A set of named constants. (e.g., `enum Direction { Up, Down, Left, Right };`)

o. How do you compile TypeScript files?

To compile TypeScript files (`.ts`) into JavaScript files (`.js`), you need to have TypeScript installed.

Steps:

Install TypeScript globally (if you haven't already):

bash
Copy
```
npm install -g typescript
```

Compile a TypeScript file:

bash
Copy
```
tsc filename.ts
```

To compile multiple files, you can use a TypeScript configuration file (`tsconfig.json`).

Watch Mode (automatically compiles TypeScript files on change):

bash
Copy
```
tsc --watch
```

p. What is the difference between JavaScript and TypeScript?

Aspec	**JavaScr t**	**TypeScript**
Type System	Dynamic ly typ	Statically typed (suppo type annotations).

	(no ty annotati s).	
Compila on	Interpret d direc by browsers	Needs to be compiled ir JavaScript.
Error Detectic	Errors a found during runtime.	Errors are caught duri compile-time.
Feature	Limited ES standard	Supports modern ES featur and additional features (e. generics, interfaces).
Object-(iented Support	Supports basic OOP.	Fully supports OOP w classes, interfaces, a access modifiers.
Support for Modules	ES6 module system.	Fully supports ES6 modul and namespa management.

q. Compare how Javascript and Typescript implement Inheritance.

JavaScript (Prototype-based Inheritance):

JavaScript uses prototype-based inheritance, where objects inherit directly from other objects. You can define classes using function constructors and the `prototype` property.

Example:

javascript

Copy

```
function Animal(name) {

    this.name = name;

}

Animal.prototype.sayHello = function() {

    console.log(`Hello, ${this.name}`);

};


const dog = new Animal('Buddy');

dog.sayHello();  // Output: Hello, Buddy
```

-

TypeScript (Class-based Inheritance):

TypeScript, being a superset of JavaScript, adds support for class-based inheritance with a more structured and formal approach.

Example:

typescript
Copy

```
class Animal {

    constructor(public name: string) {}

    sayHello() {

        console.log(`Hello, ${this.name}`);

    }

}
```

```
class Dog extends Animal {

    constructor(name: string) {

        super(name);

    }

}


const dog = new Dog('Buddy');

dog.sayHello();  // Output: Hello, Buddy
```

-

TypeScript provides type safety with classes, supports access modifiers (`public`, `private`, `protected`), and allows for better OOP design.

r. 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 in TypeScript:

Generics allow you to write functions, classes, and interfaces that work with any data type while maintaining type safety. They provide flexibility by enabling the reusability of code for different types.

- **Why Use Generics**:
    - **Type Safety**: With generics, you can avoid the pitfalls of using `any`, which bypasses TypeScript's type system. `any` can accept any value, leading to runtime errors.
    - **Reusability**: You can create reusable components that work with any data type.

Example:

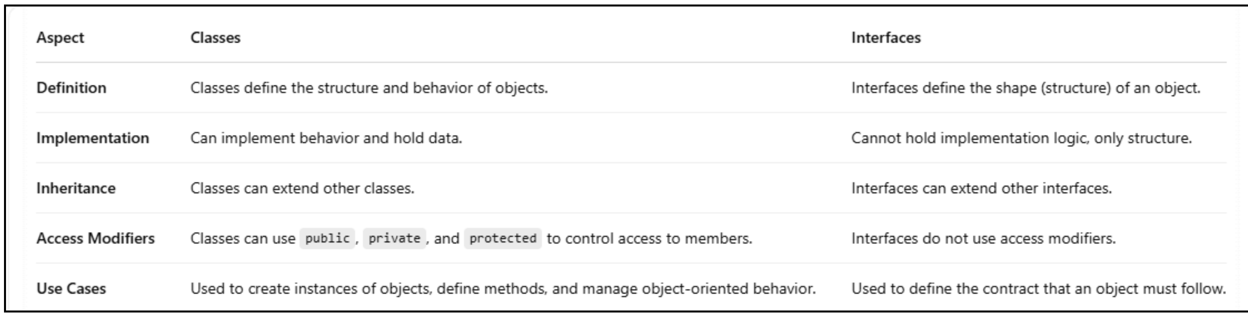
typescript
Copy
```
function identity<T>(value: T): T {

    return value;

}


let result = identity<number>(42);  // type of result is
number

let  result2  =  identity<string>("Hello");   //  type  of
result2 is string
```

- **Why Generics Over any**:
    - any disables type checking, making your code more prone to errors.
    - Generics allow you to define and enforce the type, making the code safer and reducing potential bugs.

Example in a Lab Assignment:

If the assignment requires handling different types of input (like `string`, `number`, or `boolean`), using `any` would make it difficult to maintain type safety. Generics allow the creation of flexible functions or classes without sacrificing the benefits of type safety.

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



Aspect	Classes	Interfaces
Definition	Classes define the structure and behavior of objects.	Interfaces define the shape (structure) of an object.
Implementation	Can implement behavior and hold data.	Cannot hold implementation logic, only structure.
Inheritance	Classes can extend other classes.	Interfaces can extend other interfaces.
Access Modifiers	Classes can use `public`, `private`, and `protected` to control access to members.	Interfaces do not use access modifiers.
Use Cases	Used to create instances of objects, define methods, and manage object-oriented behavior.	Used to define the contract that an object must follow.

Output:

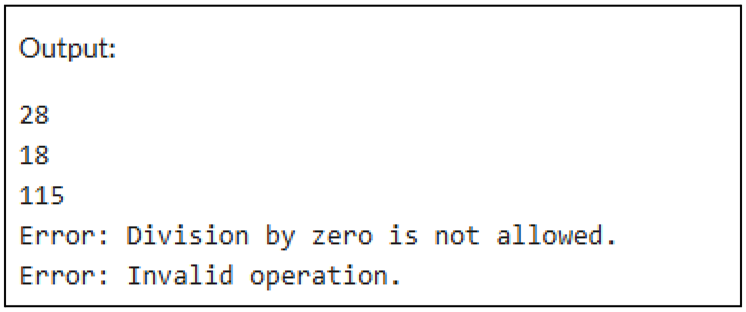
```
function calculator(num1: number, num2: number, operation: string): number | string
{
   switch (operation) {
      case "addition":
         return num1 + num2;
      case "subtraction":
         return num1 - num2;
      case "multiplication":
         return num1 * num2;
      case "division":
         if (num2 === 0) {
            return "Error: Division by zero is not allowed.";
         }
         return num1 / num2;
      default:
         return "Error: Invalid operation.";
   }
}

// Example usage
console.log(calculator(23, 5, "addition")); // Output: 15
```

console.log(calculator(23, 5, "subtraction")); // Output: 5

console.log(calculator(23, 1, "multiplication")); // Output: 50

console.log(calculator(23, 0, "division")); // Output: Error: Division by zero is not allowed.

console.log(calculator(23, 5, "modulus")); // Output: Error: Invalid operation.



```
Output:

28
18
115
Error: Division by zero is not allowed.
Error: Invalid operation.
```

// Step 1: Declare basic data types

const studentName: string = "Ronak Katariya";

const subject1: number = 45;

const subject2: number = 48;

const subject3: number = 50;

// Step 2: Calculate the total and average marks

const totalMarks: number = subject1 + subject2 + subject3;

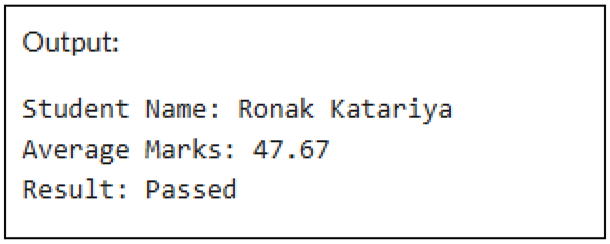
const averageMarks: number = totalMarks / 3;

// 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(`Average Marks: ${averageMarks.toFixed(2)}`);

console.log(`Result: ${isPassed ? "Passed" : "Failed"}`);
```



Output:

```
Student Name: Ronak Katariya
Average Marks: 47.67
Result: Passed
```