Experiment – 1 a: TypeScript

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

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 provides a variety of data types to ensure type safety and maintainability. These data types can be categorized into primitive types, non-primitive types, and special types.

Primitive Types

string	Represents textual data.	let studentName: string = "Alice";
number	Used for numeric values.	let age: number = 20;
boolean	Represents true or false values.	let isPassed: boolean = true;
null & undefined	Represent empty or uninitialized values.	let emptyValue: null = null; let notAssigned: undefined = undefined;
bigint	Used for very large numbers.	let bigValue: bigint = BigInt(9007199254740991);
symbol	Used for creating unique identifiers.	let uniqueKey: symbol = Symbol("id");

Non-Primitive Types

Array<T>	A collection of similar types.	let marks: number[] = [85, 90, 78];
Tuple	A fixed-size array with different types.	let student: [string, number] = ["Alice", 20];
Enum	Defines a set of named constants.	enum Grade { A, B, C, D, F } let myGrade: Grade = Grade.A;

Special Types

any	Can store any type of value but removes type safety.	let value: any = "Hello"; value = 10;
void	Used when a function doesn't return anything.	function greet(): void { console.log("Hello, world!"); }
never	Represents a function that never returns.	function throwError(): never { throw new Error("Something went wrong!"); }

Type annotations in TypeScript explicitly define the expected type of a variable, function parameter, or return value. This helps in catching type-related errors during compilation rather than runtime. Examples:

```
let studentName: string = "Alice";  // `studentName` can only hold string values
 let age: number = 20;  // `age` can only hold numbers

 function add(a: number, b: number): number
    { return a + b;
 }
 console.log(add(5, 10)); // Output: 15
```

Using type annotations improves code readability, debugging, and maintainability.

b. How do you compile TypeScript files?

To compile TypeScript files, follow these steps:

1. Install TypeScript globally (if not installed):
   npm install -g typescript

2. Create a TypeScript file (app.ts) and write your TypeScript code.

3. Compile the TypeScript file:
   tsc app.ts

This generates a JavaScript file (app.js).

4. Run the JavaScript file using Node.js:
   node app.js

5. To automatically compile files when changes are detected:
   tsc --watch

c. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Type System	Dynamic (Weakly Typed)	Static (Strongly Typed)
Compilation	Interpreted	Compiled to JavaScript
OOP Support	Prototype-based	Class-based with interfaces
Error Handling	Errors appear at runtime	Errors detected at compile-time
Generics	Not available	Fully supported

Example	function add(a, b)
	{ return a + b;
	}
	console.log(add("5", 10));
	
	Output: "510"
		function add(a: number, b: number): number {
		return a + b;
		}
		console.log(add(5, 10));
		
		Output: 15

d. Compare how Javascript and Typescript implement Inheritance.

JavaScript uses prototypal inheritance, not classical inheritance like Java or C++. Typescript uses class-based inheritance which is simply the syntactic sugar of prototypal inheritance. TypeScript supports only single inheritance and multilevel inheritance. In TypeScript, a class inherits another class using extends keyword.

- JavaScript uses prototype-based inheritance:

```
functionAnimal(name){ this.name
    = name;
}
Animal.prototype.makeSound=function(){ con
    sole.log("Some sound");
};
let dog = new Animal("Buddy");
dog.makeSound();//Output:Somesound
```

1. Constructor Function (Animal)
   Creates an instance with a name property.

2. Prototype Method (makeSound)
   Defined on Animal.prototype, meaning all instances share the same method.

3. Object Instantiation (dog)
   dog is created using new Animal("Buddy"), inheriting makeSound() via the prototype chain

- TypeScript uses class-based inheritance:

```
classAnimal{
    constructor(publicname:string){}
    makeSound() {
        console.log("Somesound");
    }
}
classDogextendsAnimal{ makeSound()
    {
        console.log("Bark!");
    }
}
let dog = new Dog("Buddy");
dog.makeSound();//Output:Bark!
```

1. Base Class (Animal)
   Defines a constructor and a method (makeSound()).
2. Derived Class (Dog)
   Inherits Animal using extends and overrides the makeSound() method.
3. Instance Creation (dog)
   new Dog("Buddy") creates an object that inherits from Animal.

TypeScript provides better structure, readability, and type safety.

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 allow us to write reusable, type-safe code. Instead of using any, which removes type safety, generics retain type information.

- Example Without Generics (Using any):

```
functionidentity(value:any):any{ return
   value;
}
console.log(identity(10));//Works,butnotypesafety
```

- With Generics:

```
functionidentity<T>(value:T):T{ return
   value;
}
console.log(identity<number>(10));//Ensurestypesafety
```

Why Generics Are Suitable:

Generics allow handling different input types dynamically while maintaining type safety. Using any would remove type checking and lead to potential errors.

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

Interface	Classes
We can Create the interface with the use of the interface keyword. i.e interface Interface_Name { \\ Interface Body }	We can create the class with class keyword. i.e class Class_Name{ \\ Class Body }
The interfaceblueprint is mainly the Type structure of object. i.e It is object with only defining the type of parameter inside.	Class is the blueprint of the object i.e. the create purposes class is how we implement the object of our code.
It is used for type checking purpose. Use of interface if TypeScript language is mainly focused on the checking the type of parameters in object.	Classes in Types script is used to made the object for something. It is used for implementing the object.
We cannot create the instance of interface with new in typescript. It means that we cannot create the copy of instance in Typescript.	We can create a new instance of the class in TypeScript. It means that we can create the copy of class with new keyword.
Interface is virtual structure. Means it only present in TypeScript code not in TypeScript compiled JavaScript code.	It always exists in code after the compilation of TypeScript to JavaScript.

Output:

a)

```
class Calculator {
  add(a: number, b: number): number
    { return a + b;
  }

  subtract(a: number, b: number): number
    { return a - b;
  }

  multiply(a: number, b: number): number
    { return a * b;
  }

  divide(a: number, b: number): number | string
    { if (b === 0) {
        return "Error: Division by zero is not allowed.";
    }
    return a / b;
  }

  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.";
    }
  }
}

const calc = new Calculator();
console.log(calc.calculate('add', 22, 7));
console.log(calc.calculate('subtract', 22, 11));
console.log(calc.calculate('multiply', 22, 10));
console.log(calc.calculate('divide', 22, 5));
console.log(calc.calculate('divide', 22, 0));
console.log(calc.calculate('modulus', 22, 2));
```

Output:

```
30
12
230
4.6
Error: Division by zero is not allowed.
Error: Invalid operation.
```

b)

```
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;
  }

  hasPassed(): boolean {
     return this.getAverageMarks() >= 40;
  }

  displayResult(): void {
     console.log(`Student Name: ${this.name}`);
     console.log(`Total Marks: ${this.getTotalMarks()}`);
     console.log(`Average Marks: ${this.getAverageMarks().toFixed(2)}`);
     console.log(`Result: ${this.hasPassed() ? "Passed" : "Failed"}`);
  }
}
const student = new Student("Sannidhi Kailaje", [55, 60, 72]);
student.displayResult();
```

Output:

```
Student Name: Ronak Katariya
Total Marks: 190
Average Marks: 63.33
Result: Passed
```