Experiment – 1 a: TypeScript

Name of Student	Sanket More
Class Roll No	D15A_29
D.O.P.	23/01/2025
D.O.S.	30/01/2025
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:**

   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:-

1. Data Types in TypeScript:

TypeScript extends JavaScript's data types and adds some of its own. Here's a summary:

- Basic Types:

  - `number`: For all numeric values (integers and floating-point).

  - `string`: For text.

  - `boolean`: `true` or `false`.

  - `null`: Represents the intentional absence of a value.

  - `undefined`: Represents a variable that has not been assigned a value.

  - `symbol`: Unique and immutable values (often used as keys in objects).

  - `bigint`: For arbitrarily large integers.

- Structural Types:

  - `object`: Represents non-primitive values, like objects, arrays, and functions.

  - `array`: Ordered collections of values.

  - `tuple`: Fixed-length arrays where each element can have a different type.

  - `function`: Represents functions, including their parameters and return types.

- Special Types:

  - `any`: Disables type checking (use with caution!).
  - `void`: Represents the absence of a return value from a function.
  - `never`: Represents values that never occur (e.g., a function that always throws an exception).
  - `unknown`: Represents a value whose type is not known at compile time. Safer than `any`.

- Type Literals: Allow specifying exact values as types (e.g., `let myStatus: "active" | "inactive";`).

- Union Types: Allow a variable to hold values of multiple types (e.g., `let id: number | string;`).

- Intersection Types: Combine multiple types into a single type (e.g., `interface A { a: string; } interface B { b: number; } type C = A & B;` - C has both `a` and `b`).

- Generics: Allow creating reusable components that can work with a variety of types.

2. Type Annotations in TypeScript:

Type annotations are a way to explicitly specify the type of a variable, function parameter, function return value, or property. They are written after a colon (`:`).

TypeScript

```typescript
3.  let name: string = "Alice"; // Type annotation for a variable
4.  function greet(person: string): string { // Type annotation for a
    parameter and return value
5.    return "Hello, " + person;
6.  }
7.
8.  interface Person {
9.    name: string; // Type annotation for a property
10.   age: number;
11. }
```

Type annotations are crucial for TypeScript's static type checking. The compiler uses them to catch type errors *before* runtime.

3. Compiling TypeScript Files:

TypeScript files (`.ts`) cannot be directly run by a browser or Node.js. They need to be *compiled* into JavaScript files (`.js`). The TypeScript compiler (`tsc`) does this.

- Command Line: `tsc myFile.ts` (compiles `myFile.ts` to `myFile.js`). You can also compile multiple files or use a configuration file (`tsconfig.json`).

- `tsconfig.json`: A configuration file that lets you customize the compilation process (e.g., output directory, target JavaScript version, strictness settings). A typical `tsconfig.json` might look like this:

JSON

```json
12. {
13.   "compilerOptions": {
14.     "target": "es5", // Target JavaScript version
15.     "outDir": "./dist", // Output directory for compiled files
16.     "strict": true // Enable strict type checking
17.   }
18. }
```

4. JavaScript vs. TypeScript:

- Type System: JavaScript is dynamically typed (types are checked at runtime), while TypeScript is statically typed (types are checked at compile time). This is the biggest difference.

- Compilation: TypeScript code needs to be compiled into JavaScript before it can be executed. JavaScript runs directly in the browser or Node.js.

- Features: TypeScript supports modern JavaScript features (like classes, interfaces, modules) and adds its own (generics, enums, type aliases).

- Error Detection: TypeScript's type system helps catch errors early during development, while JavaScript errors are often only discovered at runtime.

- Maintainability: TypeScript code is generally easier to maintain and refactor because the types provide a form of documentation and help prevent accidental errors.

5. Inheritance in JavaScript and TypeScript:

- JavaScript (Prototypal Inheritance): JavaScript uses prototypal inheritance, where objects inherit properties and methods directly from other objects (prototypes). It's often implemented using constructor functions and the `prototype` property.

- TypeScript (Class-based Inheritance): TypeScript, like many other object-oriented languages, uses class-based inheritance. Classes are blueprints for creating objects. Inheritance is achieved using the `extends` keyword. TypeScript's classes are ultimately compiled down to JavaScript that uses prototypes under the hood,

but the class syntax makes inheritance easier to work with.

TypeScript

```
19. // TypeScript Example
20. class Animal {
21.   name: string;
22.   constructor(name: string) { this.name = name; }
23.   makeSound() { console.log("Generic animal sound"); }
24. }
25.
26. class Dog extends Animal { // Dog inherits from Animal
27.   breed: string;
28.   constructor(name: string, breed: string) {
29.     super(name); // Call the parent class constructor
30.     this.breed = breed;
31.   }
32.   makeSound() { console.log("Woof!"); } // Override the makeSound
   method
33. }
34.
35. let myDog = new Dog("Buddy", "Golden Retriever");
36. myDog.makeSound(); // Output: Woof!
```

6. Generics:

Generics allow you to write reusable components (functions, classes, interfaces) that can work with a variety of types *without* sacrificing type safety. Instead of using `any` (which disables type checking), you use a type parameter (often `T`) as a placeholder for a specific type that will be determined later when the component is used.

- Flexibility: Generics make code more flexible because you can use the same component with different types.

- Type Safety: Generics preserve type safety. The compiler will check that you're using the correct types with the generic component.
- Improved Code Readability: Generics make code easier to understand because the types are clearly defined.

Why use generics over `any` in Lab 3 (or similar situations)?

Let's say you have a function that needs to process data. If you use `any`, you're telling TypeScript to ignore type checking for that data. This means:

- You lose the benefits of type checking. Errors related to incorrect data types might not be caught until runtime.
- Your code becomes harder to understand and maintain. It's not clear what types of data the function is supposed to work with.

If you use generics, you're saying, "This function can work with different types, but *I will specify the type when I use it*." This gives you:

- Type safety: The compiler will ensure that you're using the correct types.
- Code clarity: It's clear what types of data the function can handle.

7. Classes vs. Interfaces:

- Classes: Are blueprints for creating objects. They can contain properties, methods, and constructors. Classes can be instantiated (you can create objects from them). Classes can implement interfaces and extend other classes.

- Interfaces: Define a *contract* or a *shape* for an object. They specify the properties and methods that an object *must* have. Interfaces cannot be instantiated directly. Classes *implement* interfaces.

Where are interfaces used?

- Defining the shape of objects: Interfaces are commonly used to specify the structure of objects that are passed to functions or returned from functions.
- Enforcing contracts: Interfaces ensure that classes that implement them adhere to a certain set of properties and methods.
- Improving code readability: Interfaces make code easier to understand by clearly defining the structure of data.
- Working with different types: Interfaces can be used with generics to create flexible and type-safe components.

4.Output and Code

a.) Calculator

```typescript
// Define valid operations including string concatenation

type Operation = 'add' | 'subtract' | 'multiply' | 'divide' | 'concatenate';

function calculate(operation: Operation, value1: number | string, value2: number | string): number | string | never {

    if (typeof value1 !== typeof value2) {

        throw new Error('Both values must be of the same type (either numbers or strings).');

    }


    switch (operation) {

        case 'add':

            return ensureNumber(value1) + ensureNumber(value2);

        case 'subtract':
```

```typescript
        return ensureNumber(value1) - ensureNumber(value2);

    case 'multiply':

        return ensureNumber(value1) * ensureNumber(value2);

    case 'divide':

        if (ensureNumber(value2) === 0) {

            throw new Error('Error: Division by zero is not allowed.');

        }

        return ensureNumber(value1) / ensureNumber(value2);

    case 'concatenate':

        return ensureString(value1) + ensureString(value2);

    default:

        return handleInvalidOperation(operation);

    }

}

function ensureNumber(value: any): number {

    if (typeof value !== 'number') {

        throw new Error('Invalid input: Expected a number.');

    }

    return value;

}


function ensureString(value: any): string {

    if (typeof value !== 'string') {
```

```typescript
        throw new Error('Invalid input: Expected a string.');

    }

    return value;

}

function handleInvalidOperation(_operation: any): never {

     throw new Error('Error: Invalid operation. Use add, subtract, multiply,
divide, or concatenate.');

}

//console.log(calculate('divide', 8, 0));

// Example usage

try {

    console.log(calculate('add', 10, 5)); // Output: 15

    console.log(calculate('multiply', 3, 4)); // Output: 12

    console.log(calculate('divide', 8, 2)); // Output: 4

    console.log(calculate('subtract', 9, 3)); // Output: 6

     console.log(calculate('concatenate', 'Hello, ', 'World!')); // Output:
Hello, World!

    console.log(calculate('modulus' as any, 5, 2)); // Throws an error

    console.log(calculate('add', 'text', 5)); // Throws an error: Both values
must be of the same type

} catch (error: any) {

    console.error(error.message);}
```

OUTPUT:-

```
PS C:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js> ts-node "c:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js\webx_exp1a.ts"
15
12
4
6
Hello, World!
Error: Invalid operation. Use add, subtract, multiply, divide, or concatenate.
PS C:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js>
```

```
Error: Invalid operation. Use add, subtract, multiply, divide, or concatenate.
PS C:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js> ts-node "c:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js\webx_exp1a.ts"
c:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js\webx_exp1a.ts:19
                throw new Error('Error: Division by zero is not allowed.');
                      ^
Error: Error: Division by zero is not allowed.
    at calculate (c:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js\webx_exp1a.ts:19:23)
    at Object.<anonymous> (c:\Users\Sanket More\Desktop\Sanket Web code\sanket js\webx exp1a.ts:50:13)
```

b) Database

```typescript
interface Subject {

    name: string;

    marks: number;

}

interface Student {

    name: string;

    rollNumber: string;

    subjects: Subject[];

}

function calculateAverage(student: Student): number {

    if (student.subjects.length === 0) {

        return 0;

    }


    const totalMarks = student.subjects.reduce((sum, subject) => sum + subject.marks, 0);

    return totalMarks / student.subjects.length;

}



function isPassed(averageMarks: number, passingThreshold: number = 40): boolean {

    return averageMarks >= passingThreshold;
```

```typescript
  }

  function displayResult(student: Student, passingThreshold: number = 40):
void {

    const averageMarks = calculateAverage(student);

    const passed = isPassed(averageMarks, passingThreshold);


    console.log(`Student Name: ${student.name}`);

    console.log(`Roll Number: ${student.rollNumber}`);


    student.subjects.forEach(subject => {

      console.log(`${subject.name}: ${subject.marks}`);

    });


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

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

    console.log("--------------------");

  }

  const student1: Student = {

    name: "SANKET MORE",

    rollNumber: "29",

    subjects: [

      { name: "Math", marks: 45 },

      { name: "Science", marks: 38 },

      { name: "English", marks: 50 },
```

```typescript
  ],
};


const student2: Student = {

  name: "ATHARVA PRABHU",

  rollNumber: "42",

  subjects: [

    { name: "Math", marks: 75 },

    { name: "Science", marks: 82 },

    { name: "English", marks: 90 },

    { name: "History", marks: 65}, // Added an extra subject

  ],
};


const student3: Student = {

    name: "ARYAN PATANKAR",

    rollNumber: "33",

    subjects: [],

  };

// Displaying the results for each student

displayResult(student1);

displayResult(student2);

displayResult(student3);
```

```typescript
    const students: Student[] = [student1, student2, student3];

    students.forEach(student => displayResult(student));
```

OUTPUT:-

```
PS C:\Users\Sanket More\Desktop\Sanket_Web_code\sanket_js> ts-node "c:\Users\Sanket More\
Student Name: SANKET MORE
Roll Number: 29
Math: 45
Science: 38
English: 50
Average Marks: 44.33
Result: Passed
--------------------
Student Name: ATHARVA PRABHU
Roll Number: 42
Math: 75
Science: 82
English: 90
History: 65
Average Marks: 78.00
Result: Passed
--------------------
Student Name: ARYAN PATANKAR
Roll Number: 33
Average Marks: 0.00
Result: Failed
```