

Experiment – 1 a: TypeScript

Name of Student	Niraj Kothawade
Class Roll No	D15A_24
D.O.P.	23/01/2025
D.O.S.	30/01/2025
Sign and Grade	

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 (`:`).

```

let name: string = "Alice"; // Type annotation for a variable
function greet(person: string): string { // Type annotation for a
    parameter and return value
    return "Hello, " + person;
}

interface Person {
    name: string; // Type annotation for a property
    age: number;
}

```

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:

```

{
    "compilerOptions": {
        "target": "es5", // Target JavaScript version
        "outDir": "./dist", // Output directory for compiled files
        "strict": true // Enable strict type checking
    }
}

```

- **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 Example
class Animal {
  name: string;
  constructor(name: string) { this.name = name; }
  makeSound() { console.log("Generic animal sound"); }
}

class Dog extends Animal { // Dog inherits from Animal
  breed: string;
  constructor(name: string, breed: string) {
    super(name); // Call the parent class constructor
    this.breed = breed;
  }
  makeSound() { console.log("Woof!"); } // Override the makeSound
  method
}

let myDog = new Dog("Buddy", "Golden Retriever");
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

```
// Calculator in TypeScript with basic operations and error handling

/**
 * Performs basic arithmetic operations.
 *
 * @param {number} num1 - The first number.
 * @param {number} num2 - The second number.
 * @param {string} operation - The operation to perform (+, -, *, /).
 * @returns {number | string} The result of the operation or an error
message.
 */

function calculate(num1, num2, operation) {

    // Input validation: Check if the operation is valid

    if (!["+ ", "- ", " * ", " / "].includes(operation)) {

        return "Invalid operation. Please use +, -, *, or /.";
    }
}
```



```

    }

    // Perform the operation based on the operator

    switch (operation) {

        case "+":

            return num1 + num2;

        case "-":

            return num1 - num2;

        case "*":

            return num1 * num2;

        case "/":

            // Check for division by zero

            if (num2 === 0) {

                return "Division by zero error!";

            }

            return num1 / num2;

        default:

            return "An unexpected error occurred."; // Should not reach here due
to validation

    }

}

// Example Usage:

// Valid operations

console.log("10 + 5 =", calculate(10, 5, "+")); // Output: 10 + 5 = 15

console.log("10 - 5 =", calculate(10, 5, "-")); // Output: 10 - 5 = 5

```

```
console.log("10 * 5 =", calculate(10, 5, "*")); // Output: 10 * 5 = 50

console.log("10 / 5 =", calculate(10, 5, "/")); // Output: 10 / 5 = 2

// Division by zero

console.log("10 / 0 =", calculate(10, 0, "/")); // Output: 10 / 0 = Division
by zero error!

// Invalid operation

console.log("10 % 5 =", calculate(10, 5, "%")); // Output: 10 % 5 = Invalid
operation. Please use +, -, *, or /.

// More examples

console.log("25 + 12 =", calculate(25, 12, "+")); // Output: 25 + 12 = 37

console.log("100 - 33 =", calculate(100, 33, "-")); // Output: 100 - 33 = 67

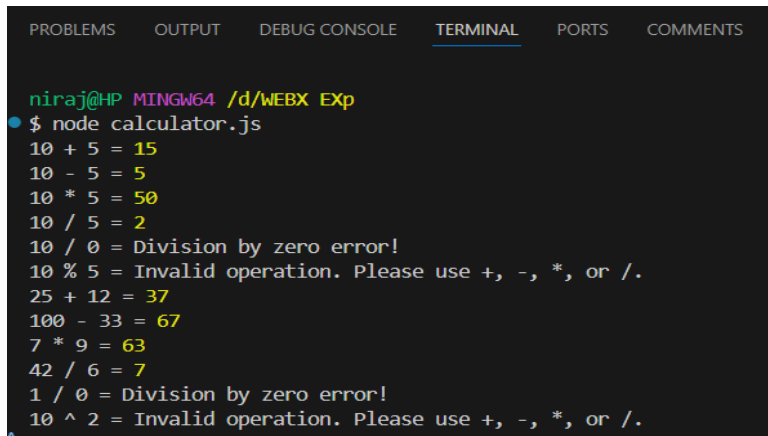
console.log("7 * 9 =", calculate(7, 9, "*")); // Output: 7 * 9 = 63

console.log("42 / 6 =", calculate(42, 6, "/")); // Output: 42 / 6 = 7

console.log("1 / 0 =", calculate(1, 0, "/")); // Output: 1 / 0 = Division by
zero error!

console.log("10 ^ 2 =", calculate(10, 2, "^")); // Output: 10 ^ 2 = Invalid
operation. Please use +, -, *, or /.
```

OUTPUT:-



The screenshot shows a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected), 'PORTS', and 'COMMENTS'. Below the tabs, the terminal shows the command prompt 'niraj@HP MINGW64 /d/WEBX Exp' followed by '\$ node calculator.js'. The output of the program is as follows:

```
niraj@HP MINGW64 /d/WEBX Exp
$ node calculator.js
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
10 / 0 = Division by zero error!
10 % 5 = Invalid operation. Please use +, -, *, or /.
25 + 12 = 37
100 - 33 = 67
7 * 9 = 63
42 / 6 = 7
1 / 0 = Division by zero error!
10 ^ 2 = Invalid operation. Please use +, -, *, or /.

```

b) Database

```
// Student Result Database (Simplified)

interface Student {
  id: string;
  name: string;
}

interface Result {
  studentId: string;
  subject: string;
  marks: number;
}

class StudentResultDatabase {
  private students: Student[] = [];
  private results: Result[] = [];

  addStudent(student: Student): void {
    this.students.push(student);
    console.log(`Student ${student.name} added.`);
  }

  addResult(result: Result): void {
    this.results.push(result);
    console.log(`Result for ${result.studentId} in ${result.subject} added.`);
  }

  getResultsByStudent(studentId: string): Result[] {
    return this.results.filter(r => r.studentId === studentId);
  }

  getAllStudents(): Student[] {
    return this.students;
  }
}
```

```
// Example Usage:
const db = new StudentResultDatabase();

db.addStudent({ id: "S001", name: "Alice" });
db.addStudent({ id: "S002", name: "Bob" });

db.addResult({ studentId: "S001", subject: "Math", marks: 85 });
db.addResult({ studentId: "S001", subject: "Science", marks: 92 });
db.addResult({ studentId: "S002", subject: "Math", marks: 78 });

console.log("\nAlice's Results:", db.getResultsByStudent("S001"));
console.log("\nAll Students:", db.getAllStudents())
```

OUTPUT:-

```
niraj@HP MINGW64 /d/WEBX Exp
● $ node student_db.js
Student Alice added.
Student Bob added.
Result for S001 in Math added.
Result for S001 in Science added.
Result for S002 in Math added.

Alice's Results: [
  { studentId: 'S001', subject: 'Math', marks: 85 },
  { studentId: 'S001', subject: 'Science', marks: 92 }
]

All Students: [ { id: 'S001', name: 'Alice' }, { id: 'S002', name: 'Bob' } ]
```