

Experiment – 1 a: TypeScript

Name of Student	Rohan Lalchandani
Class Roll No	25
D.O.P.	23/01/25
D.O.S.	30/01/25
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:

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

Different Data Types in TypeScript:

TypeScript provides several data types, including:

1. **Primitive Types:** `number`, `string`, `boolean`, `null`, `undefined`, `bigint`, `symbol`
2. **Object Types:** `array`, `tuple`, `enum`, `object`, `class`, `interface`
3. **Special Types:** `any`, `unknown`, `never`, `void`

Type annotations in TypeScript allow developers to explicitly specify the type of a variable, function parameter, or return type. This helps in catching errors at compile time.

Example:

```
let age: number = 25;
let name: string = "John";
function greet(name: string): string {
  return "Hello, " + name;
}
```

- B. How do you compile TypeScript files?

To compile a TypeScript file (`.ts`) into JavaScript (`.js`), use the TypeScript compiler (`tsc`).

1. Install TypeScript (if not installed):

```
npm install -g typescript
```

2. Compile a TypeScript file:

```
tsc filename.ts
```

This generates a `filename.js` file that can be run in a JavaScript environment.

C. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Definition	A dynamically typed scripting language used for web development.	A statically typed superset of JavaScript that adds type safety.
Compilation	Interpreted by browsers directly.	Needs to be compiled using the TypeScript compiler (<code>tsc</code>).
Type Safety	Weakly typed (dynamically typed).	Statically typed (type annotations and inference).
Syntax	Loose syntax, no need for type annotations.	Requires strict typing and follows JavaScript syntax with additional features.
Error Detection	Errors are detected at runtime.	Errors are detected at compile-time, reducing runtime issues.
OOP Support	Supports classes (ES6+), but mainly uses prototypes.	Fully supports OOP with classes, interfaces, and access modifiers.
Interfaces	Not supported.	Supported, allowing better code structure and reusability.
Generics	Not supported.	Supported, allowing flexible and reusable type-safe code.
Modules	Uses ES6 modules with <code>import</code> and <code>export</code> .	Supports both ES6 and its own module system.
Performance	Runs directly in the browser without compilation overhead.	Needs compilation, but compiled JavaScript runs at the same speed as JavaScript.
Tooling Support	Works well with most text editors.	Better tooling support with IntelliSense, auto-completion, and type checking in editors like VS Code.
Use Case	Best for quick scripting and small projects.	Best for large-scale applications where maintainability and type safety are essential.
Community Support	Large community due to long existence.	Growing community with enterprise adoption.

D. Compare how Javascript and Typescript implement Inheritance.

1. Inheritance in JavaScript

JavaScript follows **prototypal inheritance**, meaning objects inherit properties and methods from other objects via prototypes.

Example of Inheritance in JavaScript (ES5 using Constructor Functions and Prototypes):

```
// Parent class
function Animal(name) {
  this.name = name;
}

// Adding method using prototype
Animal.prototype.makeSound = function() {
  console.log(this.name + " makes a sound");
};

// Child class inheriting from Animal
function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}

// Inheriting prototype methods
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Adding a new method
Dog.prototype.bark = function() {
  console.log(this.name + " barks!");
};

// Creating an object
const dog1 = new Dog("Buddy", "Golden Retriever");
dog1.makeSound(); // Buddy makes a sound
dog1.bark(); // Buddy barks!
```

2. Inheritance in TypeScript

TypeScript follows **class-based inheritance**, making it more structured and easier to read and maintain.

Example of Inheritance in TypeScript (Using ES6 Class Syntax):

```
// Parent class
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  makeSound(): void {
    console.log(`${this.name} makes a sound`);
  }
}

// Child class inheriting from Animal
class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name); // Call parent constructor
    this.breed = breed;
  }

  bark(): void {
    console.log(`${this.name} barks!`);
  }
}

// Creating an object
const dog1 = new Dog("Buddy", "Golden Retriever");
dog1.makeSound(); // Buddy makes a sound
dog1.bark(); // Buddy barks!
```

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.

1. What Are Generics?

Generics in TypeScript allow functions, classes, and interfaces to work with **multiple types** while maintaining **type safety**. Instead of using a specific type, generics introduce **type parameters**, making code more reusable and flexible.

2. Benefits of Using Generics Over Other Types

Feature	any Type	Generics (<T>)
Type Safety	No type checking at compile time	Ensures type consistency
Reusability	Can accept any type but loses type association	Can work with multiple types while preserving type association
Performance	May cause runtime errors due to incorrect type usage	Catches errors at compile time, reducing runtime issues
Code Readability	Less readable, since the type is unknown	More readable and predictable code
Flexibility	Accepts all types but loses structure	Works with different types while maintaining structure

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

Difference Between Classes and Interfaces in TypeScript

Both **classes** and **interfaces** are used to define structures in TypeScript, but they serve different purposes.

Feature	Classes	Interfaces
Definition	A blueprint for creating objects with properties and methods.	A contract that defines the structure of an object but does not provide implementation.
Usage	Used to create instances (objects) with properties and behaviors.	Used to define object structure without instantiation.
Implementation	Provides implementation for methods.	Only declares method signatures, implementation is provided in classes.
Inheritance	Supports single and multiple inheritance using <code>extends</code> .	Supports multiple inheritance using <code>implements</code> .
Object Instantiation	Can be instantiated using <code>new</code> keyword.	Cannot be instantiated; used only as a type definition.
Access Modifiers	Supports <code>public</code> , <code>private</code> , <code>protected</code> for encapsulation.	Does not support access modifiers.
Methods & Properties	Can have method implementations.	Can only have method signatures.
Use Case	Used when creating objects with behavior.	Used when defining object structure and enforcing a contract.

Example: Class in TypeScript:

```
class Car {
  brand: string;

  constructor(brand: string) {
    this.brand = brand;
  }

  drive(): void {
    console.log(`${this.brand} is driving`);
  }
}
```

```
const myCar = new Car("Toyota");
myCar.drive(); // Output: Toyota is driving
```

- The **class** `Car` has a **constructor**, a **property** (`brand`), and a **method** (`drive()`).
- Objects can be created using `new Car("Toyota")`.

Example: Interface in TypeScript:

```
interface Vehicle {  
  brand: string;  
  drive(): void;  
}
```

```
let myVehicle: Vehicle = {  
  brand: "Honda",  
  drive() {  
    console.log(`${this.brand} is moving`);  
  }  
};
```

myVehicle.drive(); // Output: Honda is moving

- The **interface** `Vehicle` defines a **structure** that objects must follow.
- It cannot be instantiated but ensures that `myVehicle` has a `brand` and `drive()` method.

Where Are Interfaces Used?

1. Defining Object Structure

Interfaces define the expected structure of objects, ensuring consistency.

```
interface User {  
  name: string;  
  age: number;  
}
```

let user: User = { name: "John", age: 25 }; // Must follow User structure

2. Enforcing Class Structure

A class can **implement** an interface to ensure it follows a specific structure.

```
interface Animal {  
  makeSound(): void;  
}
```

```
class Dog implements Animal {  
  makeSound() {
```



```
        console.log("Bark!");
    }
}
```

The class **must** implement all methods from the interface.

3. Function Type Definition

Interfaces can define function signatures for consistency.

```
interface MathOperation {
    (a: number, b: number): number;
}
```

```
let add: MathOperation = (x, y) => x + y;
console.log(add(5, 3)); // Output: 8
```

- Ensures **add** follows the correct function signature.

4. Multiple Interface Implementation in Classes

Unlike classes that support **single inheritance**, interfaces allow **multiple implementations**.

```
interface Flyable {
    fly(): void;
}
```

```
interface Swimmable {
    swim(): void;
}
```

```
class Bird implements Flyable, Swimmable {
    fly() { console.log("Flying!"); }
    swim() { console.log("Swimming!"); }
}
```

- **Bird** implements both **Flyable** and **Swimmable**, which is not possible with classes.

GitHub Link:

https://github.com/Rohan-Lalchandani08/WebX_Lab/tree/main/WebX_Exp1a

Output:

(a) TypeScript Calculator

Output:

```
PS D:\Web x lab\ts-calculator> npm install readline-sync

added 1 package, and audited 3 packages in 552ms

found 0 vulnerabilities
PS D:\Web x lab\ts-calculator> npm install --save-dev @types/readline-sync
>>

added 1 package, and audited 4 packages in 683ms

found 0 vulnerabilities
PS D:\Web x lab\ts-calculator> npx tsc --init
>>

module: commonjs
strict: true
esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig
```

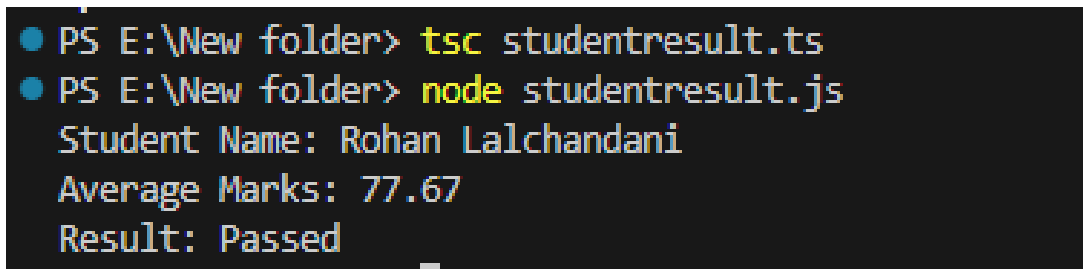
```
PS D:\Web x lab\ts-calculator> npx tsc
>>
PS D:\Web x lab\ts-calculator> node dist/calculator.js
>>
Welcome to the Simple Calculator!
Enter the first number: 2
Enter the second number: 3
Enter an operation (+, -, *, /): +
Result: 5
PS D:\Web x lab\ts-calculator> node dist/calculator.js
>>
Welcome to the Simple Calculator!
Enter the first number: 2
Enter the second number: 3
Enter an operation (+, -, *, /): *
Result: 6
PS D:\Web x lab\ts-calculator> node dist/calculator.js
>>
Welcome to the Simple Calculator!
Enter the first number: 2
Enter the second number: 0
Enter an operation (+, -, *, /): /
Result: Error: Division by zero is not allowed.
PS D:\Web x lab\ts-calculator> node dist/calculator.js
>>
Welcome to the Simple Calculator!
Enter the first number: 2
Enter the second number: 3
Enter an operation (+, -, *, /): %
Result: Error: Invalid operation.
PS D:\Web x lab\ts-calculator> █
```

This screenshot showcases the output of the TypeScript Calculator, which performs basic arithmetic operations such as addition, subtraction, multiplication, and division. The console displays:

- The results of valid operations (add, subtract, multiply, and divide).
- An error message when attempting division by zero.
- An error message for an invalid operation (e.g., modulus).

(b) Student Result Database Management System

Output:



```
PS E:\New folder> tsc studentresult.ts
PS E:\New folder> node studentresult.js
Student Name: Rohan Lalchandani
Average Marks: 77.67
Result: Passed
```

This screenshot displays the output of the Student Result Database Management System, which calculates and prints:

- The student's name.
- The average marks (formatted to two decimal places).
- The final result (either "Passed" or "Failed" based on a 40% passing criteria).

Conclusion:

This experiment successfully showcased the development of a calculator and a student result management system using TypeScript. The calculator efficiently performs arithmetic operations while ensuring proper error handling, such as managing invalid inputs and preventing division by zero. The student result management system effectively organizes student data, computes total and average marks, and determines pass/fail status using object-oriented programming principles.