Experiment – 1 b: TypeScript

Name of Student	**Rujuta Medhi**
Class Roll No	**D15A_27**
D.O.P.	**30/01/2025**
D.O.S.	**06/02/2025**
Sign and Grade	

1. **Aim:** To study Basic constructs in TypeScript.
2. **Problem Statement:**

    a. Create a base class **Student** with properties like name, studentId, grade, and a method getDetails() to display student information.
    Create a subclass **GraduateStudent** that extends Student with additional properties like thesisTopic and a method getThesisTopic().
        - Override the getDetails() method in GraduateStudent to display specific information.

    Create a non-subclass **LibraryAccount** (which does not inherit from Student) with properties like accountId, booksIssued, and a method getLibraryInfo().
    Demonstrate composition over inheritance by associating a LibraryAccount object with a Student object instead of inheriting from Student.
    Create instances of Student, GraduateStudent, and LibraryAccount, call their methods, and observe the behavior of inheritance versus independent class structures.

    b. Design an employee management system using TypeScript. Create an Employee interface with properties for name, id, and role, and a method getDetails() that returns employee details. Then, create two classes, Manager and Developer, that implement the Employee interface. The Manager class should include a department property and override the getDetails() method to include the department. The Developer class should include a programmingLanguages array property and override the getDetails() method to include the programming languages. Finally, demonstrate the solution by creating

instances of both Manager and Developer classes and displaying their details using the getDetails() method.

3. **Theory:**

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

Primitive Types: number, string, boolean, null, undefined, bigint, symbol.

Object Types: arrays, tuples, enums, interfaces, classes, functions.

Special Types: any (disables type checking), unknown (safer alternative to any), void (for functions that return nothing), and never (for unreachable code).

Type annotations

Type annotations explicitly define variable types, making the code more predictable and reducing runtime errors.

Example:

```
let age: number = 25;
let isStudent: boolean = true;
let names: string[] = ["Alice", "Bob"];
```

b. How do you compile TypeScript files?

To convert TypeScript (.ts) files into JavaScript (.js), use the TypeScript compiler

```
tsc filename.ts
```

This generates a JavaScript file that can be executed in a browser or Node.js.

c. What is the difference between JavaScript and TypeScript?

Feature	JavaScript (JS)	TypeScript (TS)
Typing	Dynamic (no type checking)	Static (explicit types)
Compilation	No compilation needed	Compiles to JavaScript
Error Detection	Errors found at runtime	Errors caught during development
Object-Oriented Features	Uses prototypes	Supports classes, interfaces, generics
Code Maintainability	Harder to scale	Easier for large projects
ES6+ Support	Supports modern JS features	Includes ES6+ and ext ures
Browser Support	Directly supported	Needs compilation to JS

d. Compare how Javascript and Typescript implement Inheritance.

JavaScript: Uses prototype-based inheritance. Objects inherit from other objects via prototypes.

TypeScript: Uses class-based inheritance, similar to OOP languages like Java and C#. Supports access modifiers (public, private, protected).

Example in TypeScript:

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
class Dog extends Animal {
  bark() {
    console.log("Woof!");
  }
}
```

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 you to **write reusable and type-safe** code without sacrificing flexibility. They let you define **placeholder types** that can be used with different data types while maintaining strict type safety.

Key Benefits of Generics

Reusability → Write one function or class that works with multiple data types.
Type Safety → Prevents runtime type errors by ensuring type correctness at compile time.
Better Code Maintainability → Reduces code duplication and improves readability.

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

Feature	Class	Interface
Definition	Blueprint for creating objects with methods & properties	Defines a structure for objects without implementation
Usage	Used to create instances (objects)	Used for type-checking and enforcing structure
Implementation	Can have both properties & methods with implementations	Only has property and method signatures (no implementation)
Inheritance	Supports inheritance using `extends`	Supports multiple inheritance using `extends` or `implements`
Compiles to	JavaScript functions and constructors	No JS output, only used for type-checking
Accessibility Modifiers	Supports `public`, `private`, `protected`	Doesn't support access modifiers

Interfaces are mainly used for:

- **Defining Object Shapes** (e.g., API responses, function parameters)
- **Ensuring Class Structure** (via implements)
- **Enforcing Type Safety** in function arguments and return types

Output:

a.

```
// Base class Student
class Student {
    constructor(public name: string, public studentId: number, public grade: string) {}

    getDetails(): string {
        return `Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}`;
    }
}

// Subclass GraduateStudent
class GraduateStudent extends Student {
        constructor(name: string, studentId: number, grade: string, public thesisTopic:
string) {
        super(name, studentId, grade);
    }

    getDetails(): string {
        return `${super.getDetails()}, Thesis Topic: ${this.thesisTopic}`;
    }

    getThesisTopic(): string {
        return `Thesis Topic: ${this.thesisTopic}`;
    }
}

// Independent class LibraryAccount
class LibraryAccount {
    constructor(public accountId: number, public booksIssued: number) {}

    getLibraryInfo(): string {
        return `Account ID: ${this.accountId}, Books Issued: ${this.booksIssued}`;
    }
}

// Composition over inheritance
class StudentWithLibrary {
```

```
    constructor(public student: Student, public libraryAccount: LibraryAccount) {}

    getFullDetails(): string {
        return `${this.student.getDetails()}\n${this.libraryAccount.getLibraryInfo()}`;
    }
}

// Creating instances
const student = new Student("Rujuta", 27, "O");
const gradStudent = new GraduateStudent("Rujuta Medhi", 27, "A", "Machine Learning");
const libraryAccount = new LibraryAccount(5001, 3);
const studentWithLibrary = new StudentWithLibrary(student, libraryAccount);

// Displaying details
console.log(student.getDetails());
console.log(gradStudent.getDetails());
console.log(gradStudent.getThesisTopic());
console.log(libraryAccount.getLibraryInfo());
console.log(studentWithLibrary.getFullDetails());
```

Output

```
Name: Rujuta, ID: 27, Grade: O
Name: Rujuta Medhi, ID: 27, Grade: A, Thesis Topic: Machine Learning
Thesis Topic: Machine Learning
Account ID: 5001, Books Issued: 3
Name: Rujuta, ID: 27, Grade: O
Account ID: 5001, Books Issued: 3

[Execution complete with exit code 0]
```

b.

```
// Employee interface
interface Employee {
    name: string;
    id: number;
```

```
    role: string;
    getDetails(): string;
}

// Manager class implementing Employee interface
class Manager implements Employee {
        constructor(public name: string, public id: number, public role: string, public department: string) {}

    getDetails(): string {
            return `Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
    }
}

// Developer class implementing Employee interface
class Developer implements Employee {
        constructor(public name: string, public id: number, public role: string, public programmingLanguages: string[]) {}

    getDetails(): string {
            return `Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Programming Languages: ${this.programmingLanguages.join(", ")}`;
    }
}

// Creating instances
const manager = new Manager("Rujuta", 27, "Manager", "IT");
const developer = new Developer("Rujuta Medhi", 27, "Developer", [ "JavaScript", "Python"]);

// Displaying details
console.log(manager.getDetails());
console.log(developer.getDetails());
```

Output

```
Name: Rujuta, ID: 27, Role: Manager, Department: IT
Name: Rujuta Medhi, ID: 27, Role: Developer, Programming Languages: JavaScript, Python

[Execution complete with exit code 0]
```