Experiment – 1 b: TypeScript

Name of Student	Spandan Deb
Class Roll No	13
D.O.P.	
D.O.S.	
Sign and Grade	

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

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.

a. 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.

Theory:

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

Typescript provides strict typing to improve code quality and catch errors early. The data types in TypeScript include:

1. Primitive Types
   - number, string, boolean, null, undefined, bigint, symbol
2. Special Types
   - any (disables type checking), unknown (requires type checking), void (for functions returning nothing), never (for unreachable code)
3. Object Types
   - object, interfaces, and custom types.
4. Array Types
   - Typed arrays (number[], string[]) and generic arrays (Array<T>).
5. Tuple Types
   - Fixed-length arrays with specified types.
6. Enum Types
   - Named constants (enum).
7. Union and Intersection Types
   - | (Union: allows multiple types) and & (Intersection: combines types).

Type annotations in TypeScript specify the expected type of variables, function parameters, and return values. Its purpose is to help catch errors at compile time, improves code readability and maintainability and enables better auto-completion And IntelliSense support

b) How do you compile TypeScript files?

Compile the Typescript file using this command tsc <filename.ts>. When this command is executed, it creates a copy of the typescript file to javascript file, now run the command node <filename.js>

c)What is the difference between JavaScript and TypeScript?

Feature	JavaScript (JS)	TypeScript (TS)
Typing	Dynamically typed	Statically typed (type checking at compile-time)
Compilation	Runs directly in browsers	Needs to be compiled to JavaScript (`tsc`)
Error Detection	Errors detected at runtime	Errors detected at compile-time, reducing bugs
OOP Support	Limited OOP features	Strong support for OOP with interfaces and generics
Code Maintainability	Harder to manage in large projects	Easier to maintain due to type safety and better tooling

d)Compare how Javascript and Typescript implement Inheritance.

In JavaScript, inheritance is implemented using the class and extends keywords. A subclass can inherit properties and methods from a parent class, and the super() function is used to call the parent class constructor. However, JavaScript does not have built-in access modifiers like private or protected, making encapsulation weaker.

In TypeScript, inheritance works similarly to JavaScript but with additional features like static typing and access modifiers (public, private, protected). TypeScript also allows the implementation of interfaces, which JavaScript does not support. This ensures better structure, code maintainability, and type safety.

While both JavaScript and TypeScript use the same fundamental inheritance model, TypeScript provides stricter rules and better encapsulation, making it more suitable for large-scale applications.

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 functions, classes, and interfaces to work with any data type while maintaining type safety. Instead of writing separate versions of a function for different types, a single generic function can handle multiple types dynamically.
Generics is suitable than any due to the following reasons

- Type Safety**:** Generics preserve the original type, preventing runtime errors. `any` disables type checking.

- Reusability**:** A single generic function/class can work with different types.

• Better Code Readability: Code remains clear and understandable without losing type information.

• Performance: TypeScript optimizes generics, while `any` may lead to unnecessary type conversions.

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

Feature	Class	Interface
Definition	A blueprint for creating objects with properties and methods.	A contract that defines the structure of an object without implementation.
Implementation	Provides both properties and method implementations.	Only defines properties and method signatures, without implementation.
Instance Creation	Can be instantiated using `new`.	Cannot be instantiated; used for type checking.
Inheritance	Uses `extends` to inherit from another class.	Uses `implements` to enforce structure in a class.
Usage	Used to create objects with specific behaviors.	Used to define a structure for objects, enforcing type safety.

Output:

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

    getDetails(): string {
        return `Student: ${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 `Graduate Student: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade},
Thesis: ${this.thesisTopic}`;
    }

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

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

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

// Composition: Associating LibraryAccount with Student
class StudentWithLibrary {
    constructor(public student: Student, public libraryAccount: LibraryAccount) {}

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

// Creating instances
const student1 = new Student("Spandan", 13, "A");
const gradStudent1 = new GraduateStudent("Sagar", 102, "A+", "Machine Learning");
const libraryAccount1 = new LibraryAccount(5001, 3);

// Associating LibraryAccount with Student using composition
const studentWithLibrary = new StudentWithLibrary(student1, libraryAccount1);

// Output
console.log(student1.getDetails());
```

```
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());
console.log(libraryAccount1.getLibraryInfo());
console.log(studentWithLibrary.getStudentAndLibraryInfo());
```

```
$ node student.js
Student: Spandan, ID: 13, Grade: A
Graduate Student: Sagar, ID: 102, Grade: A+, Thesis: Machine Learning
Thesis Topic: Machine Learning
Library Account ID: 5001, Books Issued: 3
Student: Spandan, ID: 13, Grade: A
Library Account ID: 5001, Books Issued: 3
```

```
// Employee Interface

interface Employee {

    name: string;

    id: number;

    role: string;

    getDetails(): string;

}


class Manager implements Employee {

    constructor(public name: string, public id: number, public role: string, public department: string) {}


    getDetails(): string {

        return `Manager: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
```

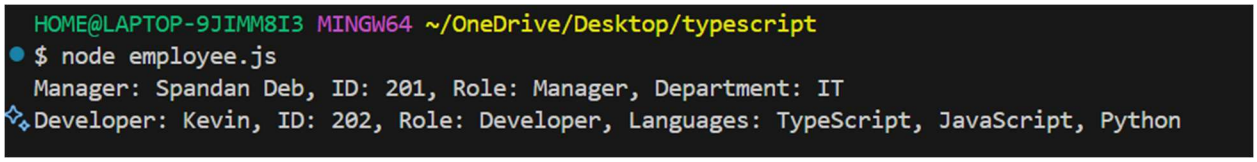
```
    }
}

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

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

// Creating instances
const manager1 = new Manager("Spandan Deb", 201, "Manager", "IT");
const developer1 = new Developer("Kevin", 202, "Developer", ["TypeScript", "JavaScript", "Python"]);

// Output
console.log(manager1.getDetails());
console.log(developer1.getDetails());
```


HOME@LAPTOP-9JIMM8I3 MINGW64 ~/OneDrive/Desktop/typescript
$ node employee.js
Manager: Spandan Deb, ID: 201, Role: Manager, Department: IT
Developer: Kevin, ID: 202, Role: Developer, Languages: TypeScript, JavaScript, Python