Experiment – 1 b: TypeScript

Name of Student	**Shreyash Ganesh Dhekane**
Class Roll No	**16**
D.O.P.	**28-01-2025**
D.O.S.	**11-02-2025**
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?

TypeScript has basic types like `string`, `number`, `boolean`, `null`, `undefined`, `void`, `any`, `unknown`, `never`, `object`, arrays (`number[]` or `Array<number>`), tuples (`[string, number]`), and enums.

Type annotations explicitly define the type of a variable, function parameter, or return value. Example:

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

b. How do you compile TypeScript files?

Use the TypeScript compiler (tsc):
tsc filename.ts
This generates a JavaScript file (filename.js) from the TypeScript code.

c. What is the difference between JavaScript and TypeScript?

- JavaScript is dynamically typed; TypeScript is statically typed.
- TypeScript has optional type annotations, interfaces, and generics.
- TypeScript requires compilation to JavaScript before execution.
- TypeScript provides better tooling and error checking at compile time.

d. Compare how Javascript and Typescript implement Inheritance.

- JavaScript uses prototype-based inheritance:

```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = function () {
    console.log("Hello " + this.name);
};
```

- TypeScript uses class-based inheritance:

```
class Person {
    constructor(public name: string) {}
    greet() {
        console.log("Hello " + this.name);
    }
}
class Student extends Person {
    constructor(name: string, public grade: number) {
        super(name);
    }
}
```

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 writing reusable and type-safe functions/classes.
- They ensure the correct type is maintained across different uses.

```
Example:
function identity<T>(value: T): T {
    return value;
}
```

- Why generics are better?
    - any removes type safety, allowing unintended values.
    - Generics retain type information, ensuring better reliability.

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

- Classes define implementation and behavior; they can have constructors and methods.
- Interfaces define a structure (contract) without implementation.
- Where are interfaces used?
    - Used for defining object shapes, function types, and enforcing structure in TypeScript.

```
Example:
interface User {
    name: string;
    age: number;
}
function getUser(user: User) {
    console.log(user.name);
}
```

4. **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 extending Student
class GraduateStudent extends Student {
    constructor(
        name: string,
        studentId: number,
        grade: string,
        public thesisTopic: string
    ) {
        super(name, studentId, grade);
    }

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

// Independent class LibraryAccount (not inheriting from Student)
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) {}

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

// Creating instances
const student1 = new Student("Alice", 101, "A");
console.log(student1.getDetails());

const gradStudent1 = new GraduateStudent("Shreyash Dhekane", 16, "A+", "Machine
Learning");
```

```
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());

const libraryAcc1 = new LibraryAccount(5001, 3);
console.log(libraryAcc1.getLibraryInfo());

// Demonstrating composition over inheritance
const studentWithLibrary = new StudentWithLibrary(student1, libraryAcc1);
console.log(studentWithLibrary.getFullInfo());
```

```
PS C:\Users\shrey\Desktop\typescript> tsc exp3.ts
PS C:\Users\shrey\Desktop\typescript> node exp3.js
Student: Alice, ID: 101, Grade: A
Student: Shreyash Dhekane, ID: 16, Grade: A+, Thesis Topic: Machine Learning
Thesis Topic: Machine Learning
Library Account ID: 5001, Books Issued: 3
Student: Alice, ID: 101, Grade: A
Library Account ID: 5001, Books Issued: 3
PS C:\Users\shrey\Desktop\typescript>
```

```
// Employee interface
interface Employee {
    name: string;
    id: number;
    role: string;
    getDetails(): string;
}

// Manager class implementing Employee
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 implementing Employee
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}, Programming Languages:
${this.programmingLanguages.join(", ")}`;
    }
}

// Demonstrating the solution
const manager1 = new Manager("Alice", 101, "Manager", "Sales");
console.log(manager1.getDetails());

const developer1 = new Developer("Shreyash Dhekane", 16, "Developer", ["TypeScript",
"JavaScript", "Python"]);
console.log(developer1.getDetails());
```