Experiment – 1 b: TypeScript

Name of Student	**MANAV PUNJABI**
Class Roll No	**D15A/44**
D.O.P.	
D.O.S.	
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.

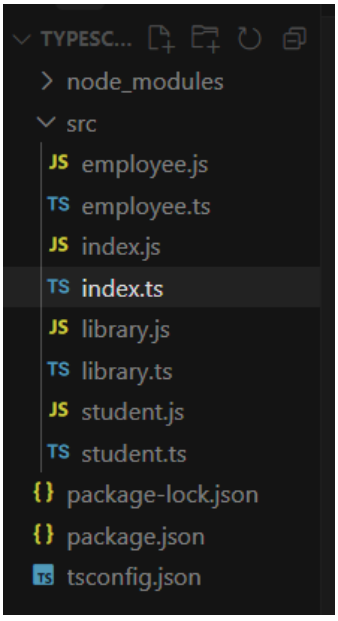
   b. Design an employee management system using TypeScript. Create an Employee interface with properties for name, id, and role, and a method getDetails() that returns employee details. Then, create two classes, Manager and Developer, that implement the Employee interface. The Manager class should include 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?
    b. How do you compile TypeScript files?
    c. What is the difference between JavaScript and TypeScript?
    d. Compare how Javascript and Typescript implement Inheritance.
    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.
    f. What is the difference between Classes and Interfaces in Typescript? Where are interfaces used?

4. **Output:**

FILE STRUCTURE :



Employee.js

```
"use strict";
Object.defineProperty(exports, "_esModule", { value: true });
exports.Developer = exports.Manager = void 0;
class Manager {
    constructor(name, id, department) {
        this.name = name;
        this.id = id;
        this.role = "Manager";
        this.department = department;
    }
    getDetails() {
        return `Manager Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
    }
}
exports.Manager = Manager;
class Developer {
    constructor(name, id, programmingLanguages) {
        this.name = name;
        this.id = id;
        this.role = "Developer";
        this.programmingLanguages = programmingLanguages;
    }
    getDetails() {
        return `Developer Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Languages: ${this.programmingLanguages.join(",
")}`;
    }
}
exports.Developer = Developer;
```

employee.ts

```
export interface Employee {
    name: string;
    id: number;
    role: string;

    getDetails(): string;
}

export class Manager implements Employee {
    name: string;
    id: number;
    role: string;
    department: string;

    constructor(name: string, id: number, department: string) {
        this.name = name;
        this.id = id;
        this.role = "Manager";
        this.department = department;
    }

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

```
export class Developer implements Employee {
    name: string;
    id: number;
    role: string;
    programmingLanguages: string[];

    constructor(name: string, id: number, programmingLanguages: string[]) {
        this.name = name;
        this.id = id;
        this.role = "Developer";
        this.programmingLanguages = programmingLanguages;
    }

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

Index.ts

```
"use strict";
Object.defineProperty(exports, "_esModule", { value: true });
const student_1 = require("./student");
const library_1 = require("./library");
const employee_1 = require("./employee");
// Student Instances
const student1 = new student_1.Student("Amit", 101, "A");
console.log(student1.getDetails());
const gradStudent1 = new student_1.GraduateStudent("Sita", 102, "A+", "Machine Learning");
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());
// LibraryAccount Instance
const libraryAccount1 = new library_1.LibraryAccount(5001, 3);
console.log(libraryAccount1.getLibraryInfo());
// Employee Instances
const manager1 = new employee_1.Manager("Rahul", 201, "Sales");
console.log(manager1.getDetails());
const developer1 = new employee_1.Developer("Neha", 202, ["TypeScript", "JavaScript", "Python"]);
console.log(developer1.getDetails());
```

index.js

```
"use strict";
Object.defineProperty(exports, "_esModule", { value: true });
exports.LibraryAccount = void 0;
class LibraryAccount {
    constructor(accountId, booksIssued) {
        this.accountId = accountId;
        this.booksIssued = booksIssued;
    }
    getLibraryInfo() {
        return `Library Account ID: ${this.accountId}, Books Issued: ${this.booksIssued}`;
    }
}
exports.LibraryAccount = LibraryAccount;
```

library.ts

```
export class LibraryAccount {
```

```
    accountId: number;
    booksIssued: number;

    constructor(accountId: number, booksIssued: number) {
        this.accountId = accountId;
        this.booksIssued = booksIssued;
    }

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

Student.js

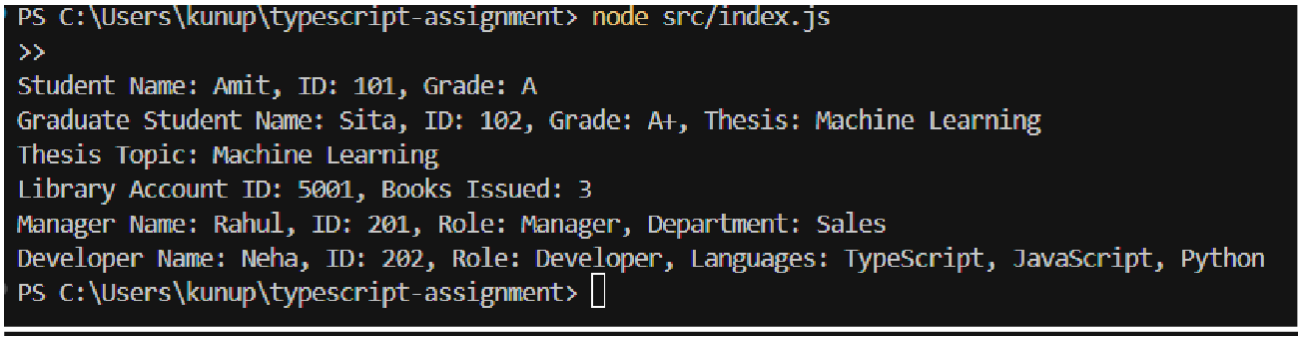
```
"use strict";
Object.defineProperty(exports, "_esModule", { value: true });
exports.GraduateStudent = exports.Student = void 0;
class Student {
    constructor(name, studentId, grade) {
        this.name = name;
        this.studentId = studentId;
        this.grade = grade;
    }
    getDetails() {
        return `Student Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}`;
    }
}
exports.Student = Student;
class GraduateStudent extends Student {
    constructor(name, studentId, grade, thesisTopic) {
        super(name, studentId, grade);
        this.thesisTopic = thesisTopic;
    }
    getThesisTopic() {
        return `Thesis Topic: ${this.thesisTopic}`;
    }
    getDetails() {
        return `Graduate Student Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}, Thesis: ${this.thesisTopic}`;
    }
}
exports.GraduateStudent = GraduateStudent;
```

student.ts

```
export class Student {
    name: string;
    studentId: number;
    grade: string;

    constructor(name: string, studentId: number, grade: string) {
        this.name = name;
        this.studentId = studentId;
        this.grade = grade;
    }

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

```
}

export class GraduateStudent extends Student {
    thesisTopic: string;

    constructor(name: string, studentId: number, grade: string, thesisTopic: string) {
        super(name, studentId, grade);
        this.thesisTopic = thesisTopic;
    }

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

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

OUTPUT :-



THEORY:

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

1. Data Types in TypeScript

TypeScript extends JavaScript by adding **static typing**. It supports the following data types:

- **1. Primitive Types**
  - number – e.g., let age: number = 25;
  - string – e.g., let name: string = "Anandi";
  - boolean – e.g., let isStudent: boolean = true;
  - null – e.g., let x: null = null;
  - undefined – e.g., let y: undefined = undefined;
- **2. Special Types**
  - any – Allows any type (not recommended)
  - unknown – Similar to any but with restricted usage
  - never – Represents values that never occur (like functions that throw errors)
- **3. Object Types**
  - Array<T> – e.g., let numbers: number[] = [1, 2, 3];
  - Tuple – e.g., let user: [string, number] = ["Alice", 30];
  - Enum –

e.g., typescript
enum Color { Red, Green, Blue };
let c: Color = Color.Green;

  - Object – e.g., let person: { name: string; age: number } = { name: "john", age: 22 };
- **4. Function Types**
  - void – No return value (e.g., function greet(): void { console.log("Hello"); })
  - never – Functions that never return (e.g., throwing an error)

2. Type Annotations in TypeScript

Type annotations help specify the expected data type explicitly:
typescript

let age: number = 25; // `age` must be a number
let name: string = "john"; // `name` must be a
string

b. How do you compile TypeScript files?

TypeScript code needs to be **compiled** into JavaScript before execution.

Steps to Compile a TypeScript File

1. Install TypeScript (if not already installed): npm install -g typescript

2. Create a TypeScript file (index.ts): let message: string = "Hello, TypeScript!"; console.log(message);

3. Compile using tsc (TypeScript Compiler):
This generates index.js file.

3. Run the JavaScript file:
For **automatic compilation**, use:
tsc --watch

c. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Type System	Dynamic (no static typing)	Static typing with type annotations
Compilation	No compilation needed	Requires compilation (tsc)
Error Detection	Errors appear at runtime	Errors detected during development
OOP Features	Limited (no interfaces, access modifiers)	Supports interfaces, generics, access modifiers
Support for ES6+	Fully supported	Supports ES6+ with additional features

d. Compare how JavaScript and TypeScript implement Inheritance. Inheritance in JavaScript

JavaScript uses **prototypes** for inheritance.

```
function Person(name) {
   this.name = name;
}

Person.prototype.getName = function () {
   return this.name;
};
```

```
function Student(name, id) {
    Person.call(this, name);
    this.id = id;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

let student1 = new Student("john", 101);
console.log(student1.getName());
```

Inheritance in TypeScript

TypeScript uses **class-based inheritance** (like Java & C++).

```
class Person {
    constructor(public name: string) {}

    getName(): string {
        return this.name;
    }
}

class Student extends Person {
    constructor(name: string, public id: number) {
        super(name);
    }
}

let student1 = new Student("john", 101);
console.log(student1.getName()); // "john"
```

TypeScript provides better readability, maintainability, and type safety over JavaScript.

---

e. How generics make the code flexible and why we should use generics over other types?

What are Generics?

Generics allow a function or class to work with **multiple data types**, increasing flexibility.

Without Generics (any type - Not Recommended)

typescript

```
function identity(arg: any): any {
    return arg;
}
```

Problem: It removes type safety, leading to potential runtime errors.

With Generics

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

console.log(identity<number>(10));    // 10
console.log(identity<string>("Hello")); // "Hello"
```

Advantages of Generics:

- Type Safety (prevents runtime errors)
- Code Reusability
- Better Performance
-

---

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

feature	Classes	Interfaces
Definition	Blueprint for creating objects	Defines structure without implementation
Can contain	Properties & Methods	Only method signatures & properties
Implementation	Can be instantiated	Cannot be instantiated
Extends/Implements	Can extend other classes	Can be implemented by classes

Example

Class:

```
class Animal {
    constructor(public name: string) {}
    makeSound() {
        console.log("Some sound");
    }
}
```

Interface:

```
interface Animal {
    name: string;
    makeSound(): void;
}
class Dog implements Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    makeSound() {
```

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

Conclusion:

In this experiment, we explored key concepts in TypeScript, focusing on object-oriented programming constructs like classes, inheritance, and interfaces.

We implemented inheritance through base and derived classes, demonstrated composition over inheritance by associating
non-subclass objects, and utilized interfaces to define flexible structures.

Through practical examples, we saw how TypeScript’s static typing enhances error detection and improves code maintainability.

Additionally, we compared TypeScript's features to JavaScript and emphasized the importance of generics for flexibility and type safety. This hands-on approach reinforced our understanding of TypeScript's power in building robust and scalable applications