Experiment – 1 b: TypeScript

Name of Student	**WADHWANI DEVANSH NARESH**
Class Roll No	**D15A_64**
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:**

1. Different Data Types in TypeScript

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

Object Types: `object`, `array`, `tuple`.

Special Types: `any`, `unknown`, `void`, `never`.

User-Defined Types: `enum`, `interface`, `type alias`, `class`, `function`.

2. Type Annotations in TypeScript

Used to specify the type of a variable, function parameter, or return value.

Example:

```
let age: number = 25;

function greet(name: string): string {
  return `Hello, ${name}`;
}
```

3. How to Compile TypeScript Files

Use the TypeScript compiler (`tsc`):

```
tsc filename.ts
```

Generates a JavaScript file (`filename.js`).

Can configure compilation settings in `tsconfig.json`.

4. Difference Between JavaScript and TypeScript

TypeScript is a superset of JavaScript with static typing.

JavaScript is dynamically typed and runs directly in browsers.

TypeScript offers **interfaces, generics, and compile-time error checking**, which JavaScript lacks.

5. Inheritance in JavaScript vs. TypeScript

JavaScript: Uses prototypal inheritance with prototype or ES6 class syntax.

TypeScript: Uses class-based inheritance with extends and supports interfaces for enforcing structure.

Example:

```
class Parent {
  greet() {
    console.log("Hello from Parent");
  }
}
class Child extends Parent {
  greetChild() {
    console.log("Hello from Child");
  }
}
```

6. Generics in TypeScript

Allow code reusability and type safety without sacrificing flexibility.

Avoids using `any`, which loses type information.

Example:

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

Why use generics in Lab Assignment 3?

Ensures type consistency while handling inputs.

Prevents runtime errors by enforcing types at compile time.

7. Difference Between Classes and Interfaces in TypeScript

Classes: Define a blueprint with implementation details (methods, properties).

Interfaces: Define a contract (structure) without implementation.

Interfaces Usage:

Used in type-checking, object shapes, and function signatures.

Example:

```
interface Person {
  name: string;
  age: number;
}
function printPerson(person: Person) {
  console.log(person.name, person.age);
}
```

4. **Output:**

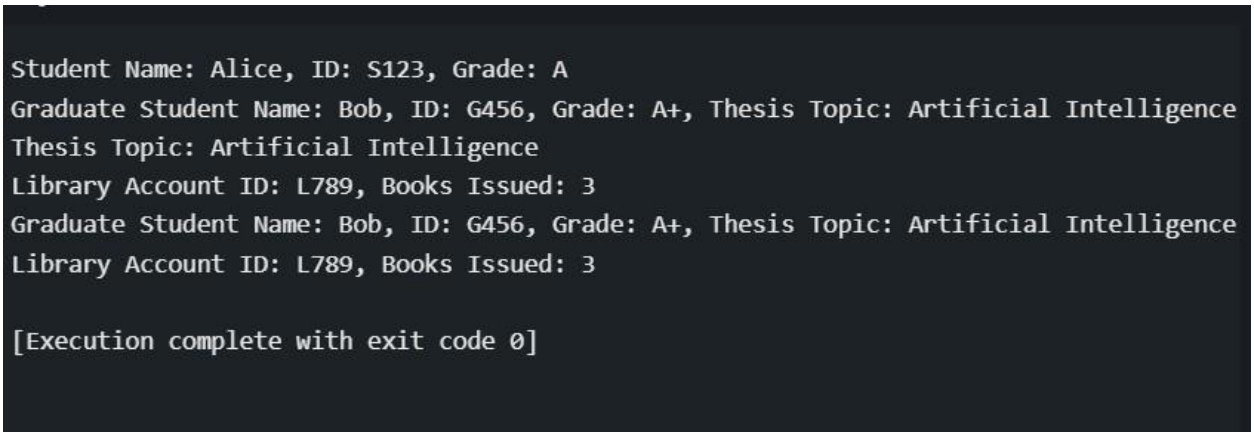
A.

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

```
    }
}
class GraduateStudent extends Student {
    thesisTopic: string;
    constructor(name: string, studentId: string, 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 Topic:
${this.thesisTopic}`;
    }
}
class LibraryAccount {
    accountId: string;
    booksIssued: number;
    constructor(accountId: string, booksIssued: number) {
        this.accountId = accountId;
        this.booksIssued = booksIssued;
    }
getLibraryInfo(): string {
        return `Library Account ID: ${this.accountId}, Books Issued: ${this.booksIssued}`;
    }
}
function associateStudentWithLibrary(student: Student, libraryAccount: LibraryAccount): string {
```

```
    return `${student.getDetails()}\n${libraryAccount.getLibraryInfo()}`;
}
const student1 = new Student("Alice", "S123", "A");
const gradStudent1 = new GraduateStudent("Bob", "G456", "A+", "Artificial Intelligence");
const libraryAccount1 = new LibraryAccount("L789", 3);
console.log(student1.getDetails());
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());
console.log(libraryAccount1.getLibraryInfo());
console.log(associateStudentWithLibrary(gradStudent1, libraryAccount1));
```

output:



B.

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

```
}

const manager = new Manager("Alice Johnson", 101, "IT");

const developer = new Developer("Bob Smith", 202, ["TypeScript", "JavaScript", "Python"]);

console.log(manager.getDetails());

console.log(developer.getDetails());
```

Output:

```
Name: Alice Johnson, ID: 101, Role: Manager, Department: IT
Name: Bob Smith, ID: 202, Role: Developer, Programming Languages: TypeScript, JavaScript, Python

[Execution complete with exit code 0]
```