Experiment – 1 b: TypeScript

Name of Student	MAHVISH SIDDIQUI
Class Roll No	56
D.O.P.	06/02/2025
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.

Theory:

1. What are the different data types in TypeScript?

- **Primitive Types**: number, string, boolean, null, undefined, symbol, bigint
- **Array**: number[], Array<number>
- **Tuple**: [string, number]
- **Enum**: enum Color {Red, Green, Blue}
- **Any**: any
- **Void**: No return type for functions
- **Never**: Represents values that never occur

What are Type Annotations in TypeScript?

Type annotations define the type of variables, function parameters, or return values to ensure type safety and avoid errors at compile time.

typescript

let x: number = 5;

function greet(name: string): string { return "Hello " + name; }

2. How do you compile TypeScript files?

Use the TypeScript compiler (tsc) to compile .ts files into .js files.

sc filename.ts

3. What is the difference between JavaScript and TypeScript?

- **JavaScript**: Dynamically typed, interpreted language.
- **TypeScript**: Statically typed superset of JavaScript that is compiled into JavaScript.

4. How do JavaScript and TypeScript implement Inheritance?

- **JavaScript**: Uses prototype-based inheritance.
- **TypeScript**: Uses class-based inheritance with the extends keyword.

Example:

JavaScript:
javascript
Copy

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

Animal.prototype.speak = function() { console.log(this.name); };
```

TypeScript:

```
class Animal {

  constructor(public name: string) {}

  speak() { console.log(this.name); }

}

class Dog extends Animal { }
```

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

Generics allow functions, classes, and interfaces to work with any data type while maintaining type safety. Using generics avoids the risks associated with any, which doesn't provide type checking, preventing potential runtime errors.

In the lab assignment 3, using generics ensures type safety while handling various input types, unlike any, which skips type checking.

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

- **Classes**: Define objects with both implementation (methods and properties).
- **Interfaces**: Define the structure (shape) of an object without implementation.

Interfaces Usage: Used to enforce structure in objects, function signatures, and class contracts.

3. **Output:**

A.

CODE:

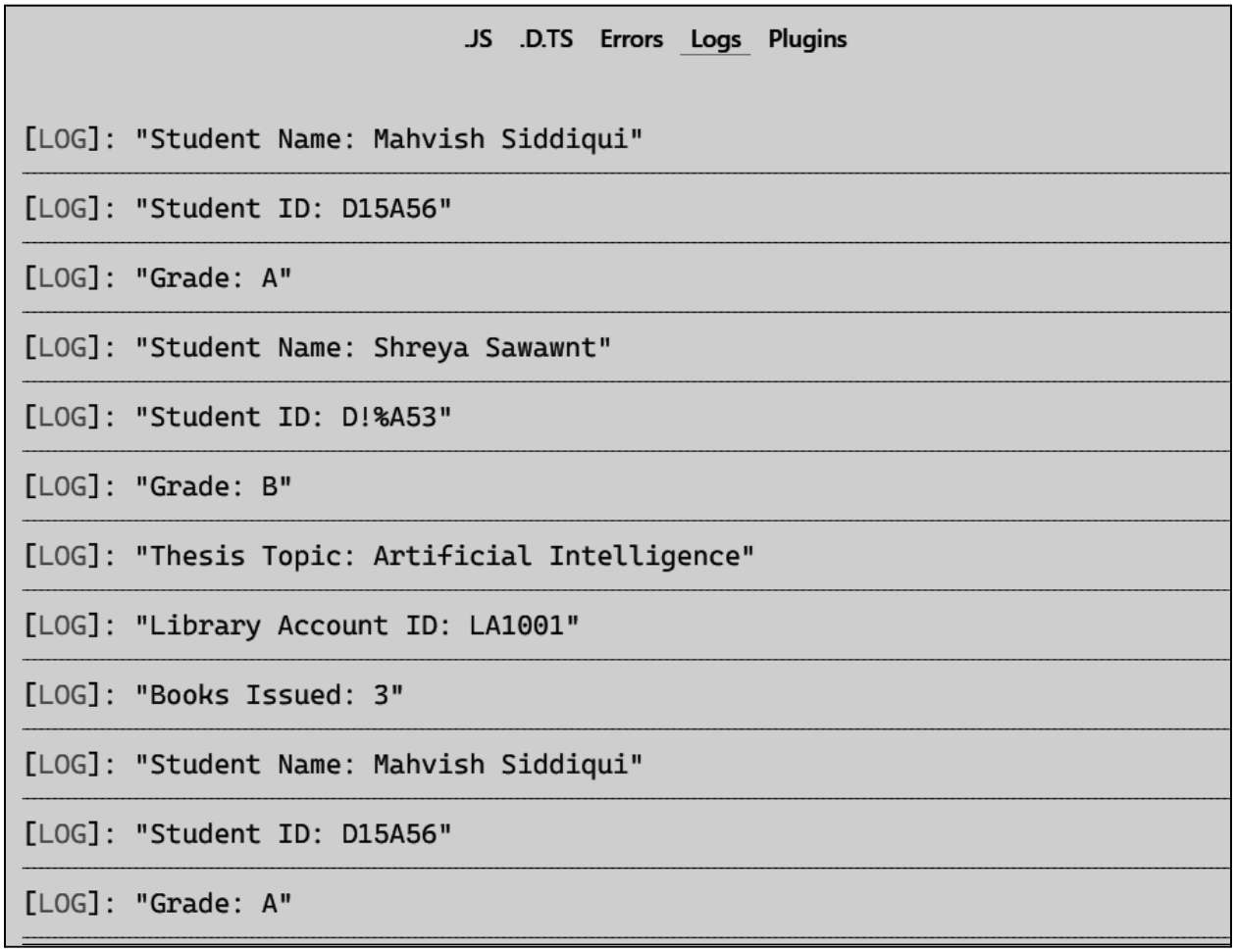
```
class Student {

  name: string;

  studentId: string;

  grade: string;



  constructor(name: string, studentId: string, grade: string) {

    this.name = name;

    this.studentId = studentId;

    this.grade = grade;

  }

  getDetails(): void {

    console.log(`Student Name: ${this.name}`);

    console.log(`Student ID: ${this.studentId}`);

    console.log(`Grade: ${this.grade}`);

  }
```

```
}

class GraduateStudent extends Student {
  thesisTopic: string;
  constructor(name: string, studentId: string, grade: string, thesisTopic: string) {
    super(name, studentId, grade); // Call the parent constructor
    this.thesisTopic = thesisTopic;
  }

  getDetails(): void {
    super.getDetails();
    console.log(`Thesis Topic: ${this.thesisTopic}`);
  }

  getThesisTopic(): void {
    console.log(`Thesis Topic: ${this.thesisTopic}`);
  }
}

class LibraryAccount {
  accountId: string;
  booksIssued: number;
```

```
  constructor(accountId: string, booksIssued: number) {
    this.accountId = accountId;
    this.booksIssued = booksIssued;
  }
  getLibraryInfo(): void {
    console.log(`Library Account ID: ${this.accountId}`);
    console.log(`Books Issued: ${this.booksIssued}`);
  }
}
class StudentWithLibrary {
  student: Student;
  libraryAccount: LibraryAccount;

  constructor(student: Student, libraryAccount: LibraryAccount) {
    this.student = student;
    this.libraryAccount = libraryAccount;
  }

  displayAllDetails(): void {
    this.student.getDetails();
    this.libraryAccount.getLibraryInfo();
```

```
  }
}


const student1 = new Student("Mahvish Siddiqui", "D15A56", "A");

student1.getDetails();


const gradStudent1 = new GraduateStudent("Shreya Sawawnt", "D!%A53", "B", "Artificial Intelligence");

gradStudent1.getDetails();


const libraryAccount1 = new LibraryAccount("LA1001", 3);

libraryAccount1.getLibraryInfo();


const studentWithLibrary = new StudentWithLibrary(student1, libraryAccount1);

studentWithLibrary.displayAllDetails();
```

OUTPUT:



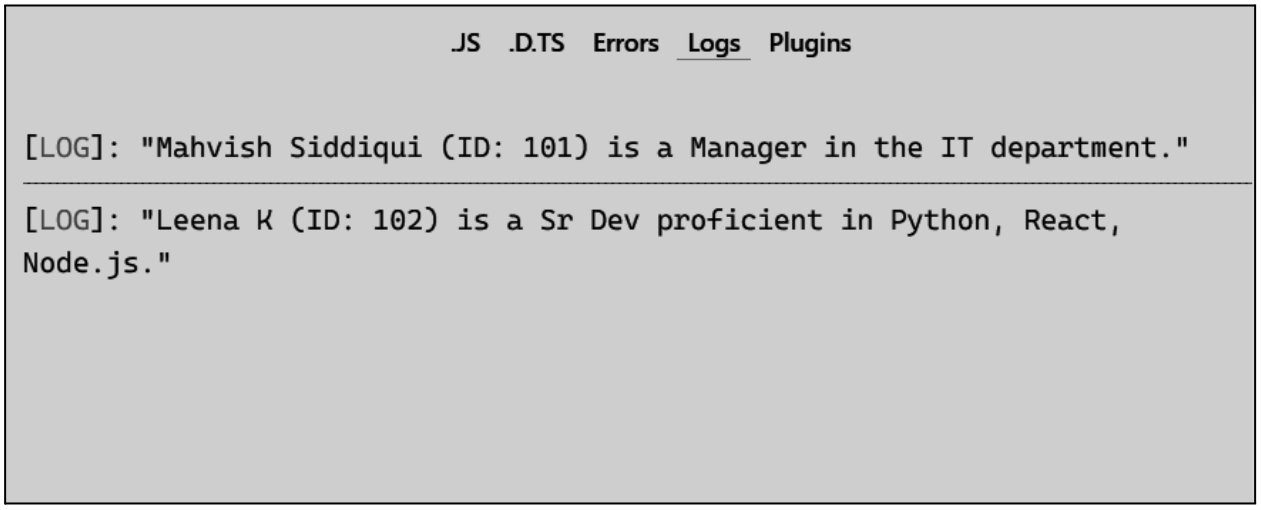
B.

CODE:

```
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, role: string, department: string) {
    this.name = name;
    this.id = id;
    this.role = role;
    this.department = department;
  }
  getDetails(): string {
    return `${this.name} (ID: ${this.id}) is a ${this.role} in the ${this.department} department.`;
  }
}
class Developer implements Employee {
  name: string;
  id: number;
  role: string;
  programmingLanguages: string[];
  constructor(name: string, id: number, role: string, programmingLanguages: string[]) {
    this.name = name;
    this.id = id;
```

```
    this.role = role;
    this.programmingLanguages = programmingLanguages;
  }
  getDetails(): string {
                return `${this.name} (ID: ${this.id}) is a ${this.role} proficient in ${this.programmingLanguages.join(", ")}.`;
  }
}
const manager1 = new Manager("Mahvish Siddiqui", 101, "Manager", "IT");
const developer1 = new Developer("Leena K", 102, "Sr Dev", ["Python", "React", "Node.js"]);
console.log(manager1.getDetails());
console.log(developer1.getDetails());
```



CONCLUSION:

In this experiment, we explored basic constructs in TypeScript, focusing on object-oriented programming concepts such as inheritance, composition, and interfaces. We created a `Student` class with properties and methods, then extended it with a `GraduateStudent` subclass, demonstrating inheritance and method overriding. Additionally, we used composition by associating a `LibraryAccount` with a `Student` object, illustrating the

benefits of composition over inheritance. We also designed an employee management system with an `Employee` interface and implemented it with `Manager` and `Developer` classes, showcasing polymorphism through method overriding. This exercise deepened our understanding of TypeScript's object-oriented features and design patterns.