Experiment – 1 b: TypeScript

Name of Student	**Prajyot Shinde**
Class Roll No	**55**
D.O.P.	**28-01-2025**
D.O.S.	**11-01-2025**
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?

Data Types in TypeScript:

TypeScript offers a rich set of data types, including:

- Primitive types like string, number, boolean, null, undefined, symbol, and bigint.
- Special types like any (for any value), unknown (safer than any), void (for functions that don't return a value), and never (for values that never occur, such as errors).
- Complex types like object, arrays, and tuple (a fixed-size collection of elements with different types).

Type Annotations: Type annotations allow developers to specify types for variables, function parameters, and return values. This enables TypeScript's static type checking, improving code safety and readability. Example: let age: number = 30;.

b. How do you compile TypeScript files?

1. Install TypeScript via npm: npm install -g typescript.
2. Compile: Use tsc filename.ts to compile TypeScript to JavaScript.
3. Watch mode: Run tsc --watch for automatic compilation upon changes.
4. Using tsconfig.json: Create a configuration file to manage compilation settings.

c. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Typing	Dynamically typed (no static types)	Statically typed with optional type annotations
Compilation	Interpreted directly by the browser	Needs to be compiled to JavaScript using tsc
Type Checking	No type checking at compile time	Type checking happens at compile time
Support for ES Features	Supports ES5 features (older versions)	Supports ES6/ES7 and beyond (Including features like async/await, decorators, etc.)
Error Handling	Errors are only found at runtime	Errors can be caught at compile time
Object-Oriented Support	Limited (using prototypes)	Full support (classes, interfac inheritance)
Interoperability	Runs directly in the browser or Node.	Needs to be compiled to JavaScript firs

d. Compare how Javascript and Typescript implement Inheritance.

JavaScript uses prototype-based inheritance, where objects can inherit properties and methods from other objects.
TypeScript supports class-based inheritance and offers features like access modifiers (public, private), and type safety. It can also implement interfaces for stricter contracts between classes.

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 functions or classes that can work with any data type while maintaining type safety. This makes code flexible and reusable, as it can handle multiple types without sacrificing type safety. Generics avoid the use of any, which bypasses type checking and could lead to runtime errors.

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

Classes are blueprints for creating objects with properties and methods. They can have constructors, implement interfaces, and inherit from other classes.
Interfaces define the structure of an object, specifying which properties and methods must exist, without providing any implementation. Interfaces are used to ensure that classes or objects adhere to a certain contract. Interfaces are typically used for defining the shape of objects, for function signatures, or for establishing contracts in class-based architecture. They ensure type consistency across various parts of the application.

4. **Output:**

PS A)

```
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 `Name: ${this.name}, Student ID: ${this.studentId}, Grade: ${this.grade}`;
    }
}

class GraduateStudent extends Student {
    thesisTopic: string;

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

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

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

class LibraryAccount {
    accountId: number;
    booksIssued: number;

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

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

const student1 = new Student("Prajyot Shinde", 101, "A");
const gradStudent1 = new GraduateStudent(
    "Prajyot Shinde",
    102,
    "B+",
    "AI in Healthcare",
);
const libraryAccount1 = new LibraryAccount(201, 3);
```

```
console.log(student1.getDetails());
console.log(gradStudent1.getDetails());
console.log(libraryAccount1.getLibraryInfo());
```

Output: -

```
Name: Prajyot Shinde, Student ID: 101, Grade: A
Name: Prajyot Shinde, Student ID: 102, Grade: B+, Thesis Topic:
    AI in Healthcare
Account ID: 201, Books Issued: 3
```

PS 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, role: string, department: string) {
      this.name = name;
      this.id = id;
      this.role = role;
      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,
      role: string,
```

```
        programmingLanguages: string[],
    ) {
        this.name = name;
        this.id = id;
        this.role = role;
        this.programmingLanguages = programmingLanguages;
    }

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

const manager1 = new Manager("Prajyot", 201, "Manager", "Sales");
const developer1 = new Developer("Soham", 202, "Developer", [
    "JavaScript",
    "TypeScript",
    "Python",
]);

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

Output: -

```
Name: Prajyot, ID: 201, Role: Manager, Department: Sales
Name: Soham, ID: 202, Role: Developer, Programming Languages: JavaScript,
    TypeScript, Python

=== Code Execution Successful ===
```

Conclusion: -

The TypeScript practicals showcase inheritance, method overriding, interfaces, and composition. GraduateStudent extends Student and overrides getDetails(), while LibraryAccount uses composition instead of inheritance for better modularity.

The Employee interface ensures a common structure for Manager and Developer classes, allowing specific properties like department and programming languages. These concepts improve code reusability, flexibility, and maintainability.