

Experiment – 1 b: TypeScript

Name of Student	Arnav Sawant
Class Roll No	D15A , 52
D.O.P.	13/02/2025
D.O.S.	20/02/2025
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?
- 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:

<Include Code and Screenshot of Output>

Q1)

// Base Student Class

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

```
}
```

```
}
```

```
// GraduateStudent Subclass
```

```
class GraduateStudent extends Student {
```

```
    thesisTopic: string;
```

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

```
        super(name, studentId, grade);
```

```
        this.thesisTopic = thesisTopic;
```

```
}
```

```
getDetails(): string {
```

```
    return `Graduate Student: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade},  
    Thesis: ${this.thesisTopic}`;
```

```
}
```

```
getThesisTopic(): string {
```

```
    return `Thesis Topic: ${this.thesisTopic}`;
```

```
}
```

```
}
```

// Independent LibraryAccount Class

```
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}`;
```

```
    }
```

```
}
```

// Composition: Associating a LibraryAccount with a Student

```
class StudentWithLibrary {
```

```
    student: Student;
```

```
    libraryAccount: LibraryAccount;
```

```
    constructor(student: Student, libraryAccount: LibraryAccount) {
```

```
    this.student = student;

    this.libraryAccount = libraryAccount;
}

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

// Creating Instances

```
const student1 = new Student("Alice", "S123", "A");
const gradStudent1 = new GraduateStudent("Bob", "G456", "A+", "AI Research");
const libraryAcc1 = new LibraryAccount("L789", 3);
```

// Composition Example

```
const studentWithLibrary = new StudentWithLibrary(gradStudent1, libraryAcc1);
```

// Outputs

```
console.log(student1.getDetails()); // Normal Student
console.log(gradStudent1.getDetails()); // Graduate Student with Thesis
console.log(libraryAcc1.getLibraryInfo()); // Library Account Info
console.log(studentWithLibrary.getFullInfo()); // Composition Demonstration
```

```
● PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> npx tsc question1.ts
● PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> node question1.js
Student: Alice, ID: S123, Grade: A
Graduate Student: Bob, ID: G456, Grade: A+, Thesis: AI Research
Library Account ID: L789, Books Issued: 3
Graduate Student: Bob, ID: G456, Grade: A+, Thesis: AI Research | Library Account ID: L789, Books Issued: 3
○ PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> █
```

Q2)

// Employee Interface

```
interface Employee {

    name: string;

    id: number;

    role: string;

    getDetails(): string;

}
```

// Manager Class

```
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: ${this.name}, ID: ${this.id}, Department: ${this.department}`;
    }
}
```

// Developer Class

```
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:  ${this.name},   ID:  ${this.id},   Languages:
        ${this.programmingLanguages.join(", ")}`;

    }

}

```

// Creating Instances

```

const manager1 = new Manager("John Doe", 101, "Engineering");

const developer1 = new Developer("Jane Smith", 202, ["TypeScript", "React", "Node.js"]);

```

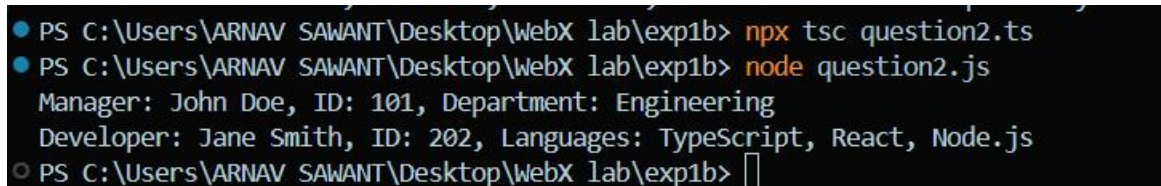
// Display Outputs

```

console.log(manager1.getDetails());

console.log(developer1.getDetails());

```



```

● PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> npx tsc question2.ts
● PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> node question2.js
  Manager: John Doe, ID: 101, Department: Engineering
  Developer: Jane Smith, ID: 202, Languages: TypeScript, React, Node.js
○ PS C:\Users\ARNAV SAWANT\Desktop\WebX lab\exp1b> 

```

THEORY ANSWERS :-

Ans Q1)

Data types -

Primitive Types

- number: let age: number = 25;
- string: let name: string = "D15A";

- boolean: `let isStudent: boolean = true;`
- null: `let emptyValue: null = null;`
- undefined: `let notDefined: undefined = undefined;`

Special Types

- any: `let randomValue: any = "Hello";`
- unknown: `let someValue: unknown = "World";`
- void (for functions without return):

```
function logMessage(): void {
    console.log("No return value");
}
```

Complex Types

- array: `let numbers: number[] = [1, 2, 3];`
- tuple: `let person: [string, number] = ["John", 25];`
- enum:

```
enum Status { Success, Failure, Pending }
```

```
let currentStatus: Status = Status.Success;
```

- object:

```
let student: { name: string; age: number } = { name: "Alice", age: 21 };
```

Type Annotations:-

Type annotations ensure type safety.

1. Variable Annotation

```
let username: string = "Arnav";
```

```
let count: number = 10;
```

2. Function Parameter & Return Type

```
function add(a: number, b: number): number {  
    return a + b;  
}
```

3. Array Annotation

```
let scores: number[] = [85, 90, 78];
```

4. Object Annotation

```
let person: { name: string; age: number } = { name: "John", age: 25 };
```

5. Void Function

```
function greet(name: string): void {  
    console.log(`Hello, ${name}`);  
}
```

Ans Q2) To compile typescript files we have to follow the following steps:-

Step-1:- Install typescript globally in your system

```
npm i -g typescript
```

Step-2:- Once you write the code for a typescript file, then you compile it by writing..

```
npx tsc filename.ts
```

Step-3:- A filename.js file would be generated just simply run that file

```
node filename.js
```

Ans Q3)

Feature	JavaScript	TypeScript
Type Safety	No static typing	Supports static typing
Compilation	No compilation needed	Needs compilation (<code>tsc</code>)
OOP Features	Limited (prototypes)	Supports classes, interfaces, generics
Readability	Can have runtime errors	Catches errors at compile time
Performance	Interpreted directly by browsers	Needs transpilation to JS

Ans Q4)

Both JavaScript and TypeScript support **class-based inheritance**, but TypeScript enforces **strong type checking**.

JavaScript Inheritance

Uses class with extends:

```

class Parent {
    constructor(name) {
        this.name = name;
    }
    greet() {
        console.log(`Hello, ${this.name}`);
    }
}

class Child extends Parent {
    constructor(name, age) {

```

```
        super(name);

        this.age = age;
    }
}

let obj = new Child("John", 21);

obj.greet(); // Hello, John
```

TypeScript Inheritance

Same syntax but with **type safety**:

```
class Parent {

    name: string;

    constructor(name: string) {

        this.name = name;
    }

    greet(): void {

        console.log(`Hello, ${this.name}`);
    }
}

class Child extends Parent {

    age: number;

    constructor(name: string, age: number) {

        super(name);

        this.age = age;
    }
}
```

```

    }

    let obj = new Child("John", 21);

    obj.greet(); // Hello, John

```

Key Difference:

- JavaScript allows any type of values.
- TypeScript enforces strict type checking.

Ans Q5)

How Generics Improve Code Flexibility

Generics allow functions and classes to work with **multiple data types** while maintaining **type safety**.

Example with Generics:

```

function identity<T>(value: T): T {

    return value;

}

console.log(identity<number>(10)); // 10

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

```

Feature	Type	Generics
Type safety	No	Yes
Code reusability	Limited	High
Performance	Slightly lower	Optimized
Predictability	Can cause unexpected errors	Ensures correct data type

Ans Q6)

Feature	Classes	Interfaces
Purpose	Blueprint for objects	Defines object structure
Implementation	Can have methods and properties	Only defines structure
Usage	Used to create instances	Used for type checking
Inheritance	Supports inheritance	Supports multiple inheritances

Where Are Interfaces Used?

- Defining object shapes

```
interface Person {  
    name: string;  
    age: number;  
}  
  
let user: Person = { name: "Arnav", age: 22 };
```

- Enforcing structure in classes

```
interface Animal {  
    makeSound(): void;  
}  
  
class Dog implements Animal {  
    makeSound() {  
        console.log("Bark!");  
    }  
}
```

```
}
```

- Ensuring API response types

```
interface ApiResponse {
```

```
    data: string;
```

```
    status: number;
```

```
}
```

Conclusion:

- Use classes when you need to create instances.
- Use interfaces when defining expected structures.