Experiment – 1 b: TypeScript

Name of Student	Pranav Pramod Titambe
Class Roll No	60
D.O.P.	06.02.2024
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?

Different Data Types in TypeScript & Type Annotations

TypeScript supports various data types:

1. Primitive Types: number, string, boolean, null, undefined, symbol, bigint.
2. Object Types: arrays, tuples, enums, classes, interfaces.
3. Special Types: any, unknown, void, never.

Type Annotations in TypeScript allow specifying the expected data type of a variable, ensuring type safety. Example:

```
let age: number = 25;
let name: string = "Alice";
let isStudent: boolean = true;
```

b.What is the difference between JavaScript and TypeScript?

1. **Type Safety**:
   - **JavaScript**: Dynamically typed (no strict types).
   - **TypeScript**: Statically typed (types are explicitly defined).
2. **Compilation**:
   - **JavaScript**: Interpreted directly by browsers.
   - **TypeScript**: Compiled to JavaScript before execution.

3. **Code Maintainability**:
   - **JavaScript**: Can be difficult to maintain in large projects

- .○ **TypeScript**: Easier to manage due to strong typing and error checking.

4. **Object-Oriented Features**:
   - **JavaScript**: Uses prototypes for inheritance.
   - **TypeScript**: Supports classes, interfaces, and generics.
5. **Error Detection**:
   - **JavaScript**: Errors occur at runtime.
   - **TypeScript**: Errors are caught at compile-time.

c. Compare how Javascript and Typescript implement Inheritance.

JavaScript: Uses prototype-based inheritance.

```
function Person(name) {
        this.name = name;
}
Person.prototype.greet = function() {
        console.log(`Hello, my name is ${this.name}`);
};
let john = new Person("John");
john.greet();
```

TypeScript: Uses class-based inheritance.

```
class Person {
        constructor(public name: string) {}
        greet() {
        console.log(`Hello, my name is ${this.name}`);
        }
}
class Student extends Person {
        constructor(name: string, public grade: number) {
        super(name);
        }
}
let student = new Student("Alice", 10);
student.greet();
```

d. 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 reusable and type-safe functions/classes without losing type information.

Without Generics (any type):

```
function identity(value: any): any {
        return value;
}
let result = identity("Hello"); // No type enforcement
```

With Generics:

```
function identity<T>(value: T): T {
        return value;
}
let result = identity<string>("Hello"); // Ensures type safety
```

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

Definition:

- **Class**: Blueprint for creating objects with methods and properties.
- **Interface**: Defines a structure without implementation.

Implementation:

- **Class**: Can be instantiated.
- **Interface**: Only defines properties and method signatures.

Usage:

- **Classes**: Used for creating objects and defining behavior.
- **Interfaces**: Used for enforcing object structure in applications.

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

```
    makeSound() { console.log("Woof!"); }
}
```

f. How do you compile TypeScript files?

To compile a TypeScript file (file.ts) into JavaScript:

Install TypeScript (if not installed):

```
nginx
npm install -g typescript
```

1. Compile the file:
   ```
   nginx
   tsc file.ts
   ```
2. This generates a file.js which can be run in browsers or Node.js.

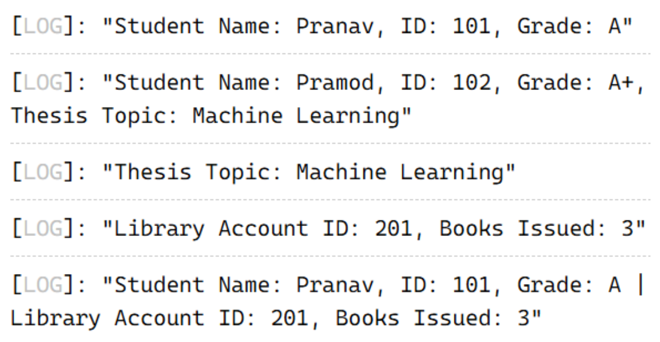
4. Output:

a.

```
class Student {
    constructor(public name: string, public studentId: number, public grade: string) {}
    getDetails(): string {
    return `Student Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}`;
    }
}
class GraduateStudent extends Student {
    constructor(name: string, studentId: number, grade: string, public thesisTopic: string) {
    super(name, studentId, grade);
    }

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


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

}

class LibraryAccount {
    constructor(public accountId: number, public booksIssued: number) {}
```

```
        getLibraryInfo(): string {
        return `Library Account ID: ${this.accountId}, Books Issued: ${this.booksIssued}`;
        }
}
class StudentWithLibrary {
        constructor(public student: Student, public libraryAccount: LibraryAccount) {}
        getStudentLibraryInfo(): string {
        return `${this.student.getDetails()} | ${this.libraryAccount.getLibraryInfo()}`;
        }
}

const student1 = new Student("Pranav", 101, "A");
const gradStudent1 = new GraduateStudent("Pramod", 102, "A+", "Machine Learning");
const libraryAcc1 = new LibraryAccount(201, 3);
const studentWithLibrary = new StudentWithLibrary(student1, libraryAcc1);
console.log(student1.getDetails());
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());
console.log(libraryAcc1.getLibraryInfo());
console.log(studentWithLibrary.getStudentLibraryInfo());
```

OUTPUT:



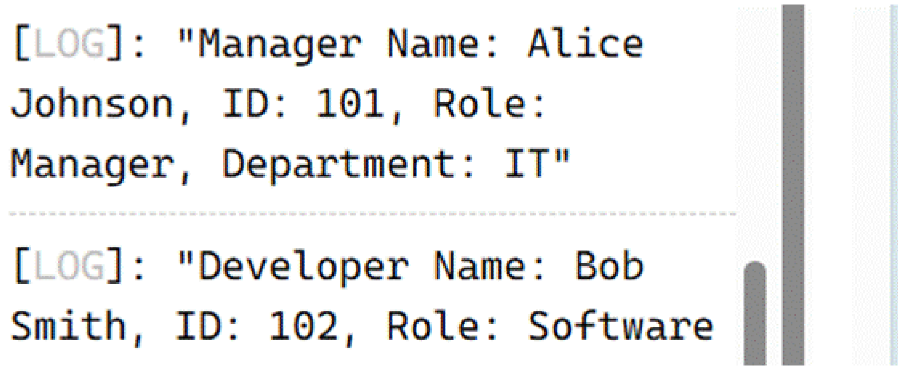
```
[LOG]: "Student Name: Pranav, ID: 101, Grade: A"
[LOG]: "Student Name: Pramod, ID: 102, Grade: A+,
Thesis Topic: Machine Learning"
[LOG]: "Thesis Topic: Machine Learning"
[LOG]: "Library Account ID: 201, Books Issued: 3"
[LOG]: "Student Name: Pranav, ID: 101, Grade: A |
Library Account ID: 201, Books Issued: 3"
```

b.

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

```
    role: string;
    getDetails(): string;
}
class Manager implements Employee {
    constructor(public name: string, public id: number, public role: string, public department: string) {}

    getDetails(): string {
        return `Manager Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
    }
}
class Developer implements Employee {
    constructor(public name: string, public id: number, public role: string, public programmingLanguages: string[]) {}

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

// Creating instances
const manager1 = new Manager("Alice Johnson", 101, "Manager", "IT");
const developer1 = new Developer("Bob Smith", 102, "Software Developer", ["TypeScript", "JavaScript", "Python"]);

// Displaying details
console.log(manager1.getDetails());
console.log(developer1.getDetails());
```

OUTPUT:



```
[LOG]: "Manager Name: Alice
Johnson, ID: 101, Role:
Manager, Department: IT"

[LOG]: "Developer Name: Bob
Smith, ID: 102, Role: Software
```