Experiment – 1 b: TypeScript

Name of Student	**Ronak Katariya**
Class Roll No	**D15A 23**
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:**
   Data Types in TypeScript:

   a. number: For numeric values (e.g., 10, 3.14).

   b. string: For text (e.g., "hello", 'world').

   c. boolean: For true/false values.

   d. array: For collections of the same type (e.g., number[], string[]).

   e. tuple: For arrays with a fixed number of elements and known types (e.g., [string, number]).

   f. enum: For defining named constants.

   g. any: Disables type checking (use sparingly).

   h. void: For functions that don't return a value.

   i. null and undefined: Represent the absence of a value.

   j. object: Represents non-primitive type.

   k. symbol: Represents a unique identifier

4. Type Annotations:

   Type annotations are a way to explicitly specify the type of a variable, parameter, or return value in TypeScript. They use a colon (:) followed by the type after the variable name.

   typescript

```
let age: number = 30;
function greet(name: string): string {
   return "Hello, " + name;
}
```

5.
6. Compiling TypeScript Files:

   Use the tsc (TypeScript compiler) command in the terminal:

   bash

```
tsc yourFileName.ts
```

7.

8. This will compile the yourFileName.ts file into a yourFileName.js file. You might need to configure a tsconfig.json file for more complex projects.

9. Difference Between JavaScript and TypeScript:

    a. Typing: JavaScript is dynamically typed (types are checked at runtime), while TypeScript is statically typed (types are checked at compile time).

    b. Features: TypeScript offers features like classes, interfaces, enums, generics, and more, which are not natively available in JavaScript (though some can be polyfilled).

    c. Compilation: TypeScript code needs to be compiled into JavaScript code before it can be run in a browser or Node.js. JavaScript can be run directly.

    d. Error Detection: TypeScript catches type-related errors during development, while JavaScript only reveals them at runtime.

    e. Readability and Maintainability: TypeScript's type system and features generally lead to more readable and maintainable code, especially in large projects.

10. Inheritance in JavaScript vs. TypeScript:

    a. JavaScript (Classical Inheritance using Prototypes): JavaScript traditionally uses prototypal inheritance. This can be a bit more complex to understand and implement than classical inheritance.

javascript

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.name);
}

function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}
Dog.prototype = Object.create(Animal.prototype); // Set up the prototype chain
Dog.prototype.constructor = Dog; // Reset the constructor property
Dog.prototype.bark = function() {
  console.log("Woof!");
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.sayHello(); // Inherited from Animal
myDog.bark();
```

    b. TypeScript (Classical Inheritance): TypeScript provides a class keyword and extends keyword that makes inheritance much cleaner and easier to understand, similar to languages like Java or C++. It's essentially syntactic sugar over

JavaScript's prototypal inheritance, but it provides a more familiar and structured approach.

typescript

```
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHello() {
        console.log("Hello, I'm " + this.name);
    }
}

class Dog extends Animal {
    breed: string;
    constructor(name: string, breed: string) {
        super(name); // Call the parent constructor
        this.breed = breed;
    }
    bark() {
        console.log("Woof!");
    }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.sayHello(); // Inherited from Animal
myDog.bark();
```

c.

11. TypeScript's class and extends make inheritance more explicit and easier to work with, reducing the boilerplate code required in traditional JavaScript.

12. Generics:

    a. Flexibility: Generics allow you to write code that can work with a variety of types without sacrificing type safety. Instead of using any, you can define a type parameter (e.g., <T>) that represents the specific type the code will operate on.

    b. Type Safety: Generics preserve type information. The compiler knows the specific type being used, so it can perform type checking and provide better error messages.

    c. Why Use Generics Over any: Using any disables type checking, which defeats the purpose of using TypeScript. Generics provide the flexibility of working with multiple types while still maintaining type safety.

13. In lab assignment 3 (which you didn't provide, but I can still answer generally), if you're handling input of potentially different types, generics are more suitable than any because:

a. If you used any, you'd lose all type information about the input, making it difficult to work with the data safely and predictively.

b. Generics allow you to write a function or class that can accept different types of input, but within each specific usage, the type is known and enforced.

14. For example:

typescript

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

let myString: string = identity<string>("hello");  // Type of myString is string
let myNumber: number = identity<number>(42);     // Type of myNumber is number
```

15.

16. Classes vs. Interfaces:

a. Classes:

i. Define both the structure (properties) and behavior (methods) of an object.

ii. Can be instantiated (you can create objects from them using new).

iii. Can have constructors to initialize object state.

iv. Can include implementation details.

b. Interfaces:

i. Define a contract or a shape of an object. They specify what properties and methods an object should have, but not how they are implemented.

ii. Cannot be instantiated directly.

iii. Do not have constructors.

iv. Primarily used for type checking and defining relationships between objects.

17. Where Interfaces Are Used:

a. Defining Contracts: Ensuring that objects conform to a specific structure.

b. Implementing Polymorphism: Allowing different classes to be treated as the same type if they implement the same interface.

c. Decoupling Code: Reducing dependencies between different parts of your code.

d. Describing API Responses: Defining the structure of data returned from an API.

Output:

A.

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

// Subclass GraduateStudent extending Student
class GraduateStudent extends Student {
    thesisTopic: string;

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

    // Override getDetails() method to include thesis topic
    getDetails(): string {
        const baseDetails = super.getDetails();
        return `${baseDetails}, Thesis Topic: ${this.thesisTopic}`;
    }

    getThesisTopic(): string {
        return this.thesisTopic;
    }
}

// Non-subclass LibraryAccount class (does not inherit from Student)
class LibraryAccount {
    accountId: string;
    booksIssued: number;

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

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

```
    }
}

// Composition over inheritance: Associate LibraryAccount with Student
class StudentWithLibraryAccount {
    student: Student;
    libraryAccount: LibraryAccount;

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

    getStudentDetails(): string {
        return this.student.getDetails();
    }

    getLibraryInfo(): string {
        return this.libraryAccount.getLibraryInfo();
    }
}

// Create instances of Student, GraduateStudent, and LibraryAccount
const student = new Student("Ronak", "123", "A");
const graduateStudent = new GraduateStudent("Sannidhi", "G456", "B", "AI Research");
const libraryAccount = new LibraryAccount("L789", 5);

// Demonstrating composition
const studentWithAccount = new StudentWithLibraryAccount(student, libraryAccount);
console.log(studentWithAccount.getStudentDetails());  // Composition: Get student details
console.log(studentWithAccount.getLibraryInfo());    // Composition: Get library info
console.log(graduateStudent.getDetails());           // Inheritance: Get graduate student's details
```

Output:

```
Name: Ronak, Student ID: 123, Grade: A
Account ID: L789, Books Issued: 5
Name: Sannidhi, Student ID: G456, Grade: B, Thesis Topic: AI Research
```

B.

```
// Employee interface
interface Employee {
    name: string;
    id: string;
    role: string;
```

```
    getDetails(): string;
}

// Manager class implementing Employee interface
class Manager implements Employee {
    name: string;
    id: string;
    role: string;
    department: string;

    constructor(name: string, id: string, 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}`;
    }
}

// Developer class implementing Employee interface
class Developer implements Employee {
    name: string;
    id: string;
    role: string;
    programmingLanguages: string[];

    constructor(name: string, id: string, 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(", ")}`;
    }
}

// Creating instances of Manager and Developer and displaying their details
```

```
const manager = new Manager("Ronak", "M001", "Manager", "Finance");
const developer = new Developer("Ram", "D002", "Developer", ["JavaScript", "TypeScript",
"Python"]);

console.log(manager.getDetails());  // Manager details
console.log(developer.getDetails());  // Developer details
```

Output:

```
Name: Ronak, ID: M001, Role: Manager, Department: Finance
Name: Ram, ID: D002, Role: Developer, Programming Languages: JavaScript, TypeScript, Python
```