Experiment – 1 b: TypeScript

Name of Student	Madhura Jangale
Class Roll No	D15A_20
D.O.P.	
D.O.S.	
Sign and Grade	

Aim: To study Basic constructs in TypeScript.

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:

a. What are the different data types in TypeScript? What are Type Annotations in Typescript?

Different Data Types in TypeScript TypeScript provides a variety of built-in data types, including:

1. Number: Represents both integer and floating-point values.
   Eg: let age: number = 25;

2. String: Represents textual data.
   Eg: let name: string = "John";

3. Boolean: Represents true or false values.
   Eg: let isStudent: boolean = true;

4. Array: Represents a collection of elements of the same type.
   Eg: let numbers: number[] = [1, 2, 3];

5. Tuple: Represents an array with fixed types and a specific length.
   Eg: let person: [string, number] = ["Alice", 30];

6. Enum: Represents a set of named constants.
   Eg: enum Color {Red, Green, Blue}

7. Any: Represents any data type, used when the type is unknown.
8. Void: Used for functions that do not return a value.
9. Null and Undefined: Represent null and undefined values.

10. Never: Represents values that never occur (e.g., functions that throw errors).

Type Annotations in TypeScript Type annotations explicitly specify the type of a variable, function parameter, or return value.

Eg:

```
let message: string = "Hello, TypeScript!";
function add(a: number, b: number): number {
  return a + b;
}
```

b. How do you compile TypeScript files?

Compiling TypeScript Files To compile TypeScript files, you need to use the TypeScript compiler (tsc). First, install TypeScript globally if not already installed. Then, compile a single file using the tsc command followed by the filename. This generates a corresponding JavaScript file. You can also compile multiple files at once, watch for changes with the --watch flag, or configure

project-wide compilation using a tsconfig.json file. The tsconfig.json file allows setting compiler options such as target JavaScript version, output directory, and strict type-checking rules.

c. What is the difference between JavaScript and TypeScript?

Difference Between TypeScript and JavaScript

Feature	JavaScript	TypeScript
Typing	Dynamically typed	Statically typed
Compilation	No compilation required	Needs compilation using tsc
Interfaces	Not supported	Supported
Generics	Not supported	Supported
Modularity	Uses ES6 modules	Stronger support for modules
Debugging	More runtime errors	Fewer runtime errors due to type safety
OOP Features	Limited class-based OOP	Fully supports OOP concepts like class interfaces, and access modifiers
Tooling Support	Basic support	Better support with IDEs due to static typing
Use Case	Suitable for small projects a quick scripting	Preferred for large-scale applications w maintainability in mind

d. Compare how Javascript and Typescript implement Inheritance.

Feature	Class	Interface
Definition	Blueprint for creating objects	Defines structure but has no implementation

Implementation	Can have methods a properties	Only defines properties/methods but does implement them
Inheritance	Can extend another class	Can extend multiple interfaces
Use Case	Used for object instantiation	Used for defining a contract

Interfaces are used to define object shapes, function signatures, and enforce structure in class implementations. They help maintain consistency in data structures, making the code more maintainable and scalable.

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 make the code flexible by allowing it to handle different data types without specifying a concrete type. Instead of writing separate methods or classes for different types, generics enable you to write reusable and type-safe code.

Why Use Generics Over Other Types?

2. Type Safety: Generics ensure that only the expected data type is used, reducing runtime errors caused by incorrect type casting.

3. Code Reusability: Instead of writing multiple versions of a method or class for different types, generics allow a single, reusable implementation.

4. Compile-Time Checking: Errors are caught at compile-time rather than at runtime, improving code reliability.

5. Eliminates Type Casting: Without generics, you would need explicit type casting, which increases the risk of ClassCastException. Generics eliminate the need for this casting.

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

Difference Between Classes and Interfaces in TypeScript

Feature	Class	Interface

Definition	Blueprint for creating objects w properties and methods.	Defines a contract for an object's structu but does not provide implementation.
Implementation	Can have constructors, properti and method implementations.	Only declares method signatures a properties; no implementation.
Instantiation	Can create objects using n keyword.	Cannot be instantiated directly.
Inheritance	Supports inheritance (extends).	Supports multiple inheritance (exten multiple interfaces).
Usage	Used to create reusable obje blueprints.	Used to enforce a structure for objec and classes.
Modifiers	Can have public, private, protect modifiers.	Cannot have access modifiers (pub private, etc.).
Performance	Generates JavaScript code.	Only used for type checking, does r exist in JavaScript output.

Output:

a.

```
// Base class Student
class Student {
  constructor(public name: string, public studentId: number, public grade: string) {}

  getDetails(): string {
    return `Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}`;
  }
}

// Subclass GraduateStudent
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}`;
    }
}

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

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

// Composition over inheritance
class StudentWithLibrary {
    constructor(public student: Student, public libraryAccount: LibraryAccount) {}

    getFullDetails(): string {
        return `${this.student.getDetails()}\n${this.libraryAccount.getLibraryInfo()}`;
    }
}

// Creating instances
const student = new Student("Madhura", 20, "O");
const gradStudent = new GraduateStudent("Madhura Jangale", 20, "A", "Machine Learning");
const libraryAccount = new LibraryAccount(5041, 3);
const studentWithLibrary = new StudentWithLibrary(student, libraryAccount);

// Displaying details
console.log(student.getDetails());
console.log(gradStudent.getDetails());
console.log(gradStudent.getThesisTopic());
console.log(libraryAccount.getLibraryInfo());
console.log(studentWithLibrary.getFullDetails());
```

Output:

```
Name: Madhura, ID: 20, Grade: O
Name: Madhura Jangale, ID: 20, Grade: A, Thesis Topic: Machine Learning
Thesis Topic: Machine Learning
Account ID: 5041, Books Issued: 3
Name: Madhura, ID: 20, Grade: O
Account ID: 5041, Books Issued: 3
```

b.

```
// Employee interface
interface Employee {
   name: string;
   id: number;
   role: string;
   getDetails(): string;
}

// Manager class implementing Employee interface
class Manager implements Employee {
   constructor(public name: string, public id: number, public role: string, public department:
string) {}

   getDetails(): string {
      return `Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
   }
}

// Developer class implementing Employee interface
class Developer implements Employee {
   constructor(public name: string, public id: number, public role: string, public
programmingLanguages: string[]) {}

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

// Creating instances
const manager = new Manager("Madhura", 20, "Manager", "IT");
const developer = new Developer("Madhura Jangale", 27, "Developer", [ "TypeScript", "Python"]);
```

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

Output:

```
Name: Madhura, ID: 20, Role: Manager, Department: IT
Name: Madhura Jangale, ID: 27, Role: Developer, Programming Languages: TypeScript, Python
```