**Experiment – 1.b Typescript**

|  |  |
| --- | --- |
| **Name of Student** | **Vanshika Ambwani** |
| **Class Roll No** | **D15A 2** |
| **D.O.P.** |  |
| **D.O.S.** |  |
| **Sign and Grade** |  |

**Experiment – 2: TypeScript**

1. **Aim:** To study Basic constructs in TypeScript.
2. **Problem Statement:** 
   1. 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.

* 1. 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.

1. **Theory:**
2. [What are the different data types in TypeScript?](https://www.naukri.com/learning/articles/typescript-interview-questions/?#What-are-the-different-data-types-in-TypeScript) What are Type Annotations in Typescript?

TypeScript has several data types, including Primitive Types (string, number, boolean, null, undefined, symbol, bigint) and Object Types (arrays, tuples, enums, interfaces, and classes). It also supports any, unknown, void, never, and union types, providing flexibility and type safety. Type Annotations in TypeScript explicitly define variable types, ensuring type safety and better code maintainability. For example: let age: number = 25; let name: string = "John"; This helps prevent type-related errors, improves code readability, and enhances debugging. TypeScript also uses type inference when annotations are omitted

1. How do you compile TypeScript files?

Step 1: Ensure you have Node.js installed on your system. You can download it from the official website if it's not already installed.

Step 2: Install TypeScript globally using the following command: npm install -g typescript

Step 3: Create a .ts file (e.g., example.ts) in your project directory that contains your TypeScript code.

Step 4: Compile your TypeScript file by running the following command in the terminal: tsc This will generate a JavaScript file () in the same directory.

Step 5: If you want TypeScript to automatically recompile when changes are made to your files, use watch mode: tsc --watch

1. [What is the difference between JavaScript and TypeScript?](https://www.naukri.com/learning/articles/typescript-interview-questions/?#Q10.%20What%20is%20the%20difference%20between%20JavaScript%20and%20TypeScript)

JavaScript is a dynamic, interpreted language commonly used for client-side web development. It is loosely typed, meaning types are determined at runtime, which can lead to runtime errors and unexpected behavior. TypeScript is a superset of JavaScript that introduces static typing, allowing developers to define variable types explicitly. It offers advanced features like interfaces, classes, and type annotations, which enhance code reliability, error-checking, and maintainability. TypeScript code is compiled into JavaScript, ensuring it works in all JavaScript environments. In essence, TypeScript provides a more structured, predictable, and type-safe development experience compared to JavaScript.

1. Compare how Javascript and Typescript implement Inheritance.

In JavaScript, inheritance is prototype-based, where objects inherit properties and methods through their prototype chain. With ES6, JavaScript introduced the class syntax, which simplifies inheritance but still relies on prototypes under the hood. In TypeScript, inheritance follows the same prototype-based model as JavaScript but with added features like static typing, access modifiers (public, private, protected), and support for interfaces and abstract classes. TypeScript enforces type safety at compile-time, preventing errors before runtime, and allows more structured inheritance through type definitions and stricter class contracts. Thus, while both use class syntax for inheritance, TypeScript offers better type safety and more features for enforcing class structures.

1. 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 in TypeScript provide flexibility by enabling functions, classes, or interfaces to handle various data types while maintaining strict type safety. Unlike using any, which disables type checking, generics ensure that types are consistent, reducing the likelihood of runtime errors and enhancing code reliability. They also improve code readability, reusability, and IDE support, making development smoother and more efficient. In Lab Assignment 3, using generics is advantageous over any as it allows combining both numbers and strings safely. This ensures predictable behavior, prevents type-related issues, and enhances maintainability, making the codebase more robust.

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

In TypeScript, classes and interfaces serve different purposes. A class defines a blueprint for creating objects, including properties and methods. It supports features like inheritance, access modifiers (public, private, protected), and method implementations. On the other hand, an interface defines the structure of an object but does not provide implementations. It acts as a contract, ensuring that objects follow a specific shape. Interfaces are commonly used to define object types, enforce consistency in function parameters, and enable multiple inheritance. They help in building scalable and maintainable applications by ensuring a structured and predictable data format

4. **Output:**

**a.**

class Student { name: string; studentId: number; grade:

string; libraryAccount:

LibraryAccount;

constructor(name: string, studentId: number, grade: string, libraryAccount: LibraryAccount)

{

this.name = name; this.studentId = studentId; this.grade = grade; this.libraryAccount = libraryAccount;

}

getDetails(): string { return `Student Name: ${this.name}, ID:

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

}

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

}

}

class GraduateStudent extends Student {

thesisTopic: string;

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

}

getThesisTopic(): string { return

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

override getDetails(): string { return `${super.getDetails()},

Thesis Topic: ${this.thesisTopic}`;

}

}

class LibraryAccount { accountId: number; booksIssued: number;

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

}

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

}

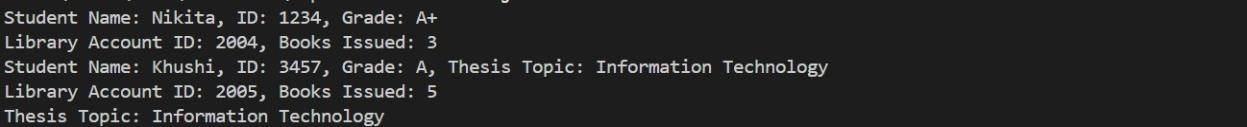
}

const studentLibraryAccount1 = new LibraryAccount(2004, 3); const studentLibraryAccount2 = new LibraryAccount(2005, 5); const student1 = new Student("Nikita", 1234, "A+", studentLibraryAccount1); console.log(student1.getDetails()); console.log(student1.getLibraryDetails());

const student2 = new GraduateStudent("Khushi", 3457, "A", studentLibraryAccount2,

"Information Technology"); console.log(student2.getDetails()); console.log(student2.getLibraryDetails()); console.log(student2.getThesisTopic());

**Output:**



**b.**

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

getDetails(): string;

}

class Manager implements Employee { name: string; id: number; role: string = "Manager"; department:

string;

constructor(name: string, id: number, department: string) {

this.name = name; this.id = id; 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

= "Developer"; programmingLanguages: string[];

constructor(name: string, id: number, programmingLanguages: string[]) {

this.name = name; this.id = id;

this.programmingLanguages = programmingLanguages;

}

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

}

}

const manager1 = new Manager("Karan Singh", 6743, "Sales");

console.log(manager1.getDetails());

const developer1 = new Developer("Khushi Kapoor", 8458, ["JavaScript", "TypeScript",

"Python"]); console.log(developer1.getDetails());

**Output:**

