Experiment – 1 b: TypeScript

Name of Student	Aryan Dangat
Class Roll No	D15A_12
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:**

1. Data Types in TypeScript:

TypeScript extends JavaScript's data types and adds some of its own. Here's a summary:

- **Basic Types:**
    - number: For all numeric values (integers, floating-point numbers, etc.).
    - string: For text.
    - boolean: trueor false.
    - null: Represents the intentional absence of a value.
    - undefined: Represents a variable that has been declared but not assigned a value.
    - symbol: Unique and immutable values (often used as keys for object properties).
    - bigint: For arbitrarily large integers.
- **Structural Types:**
    - object: Represents non-primitive values, such as objects, arrays, and functions.
    - array: Ordered collections of values.
    - tuple: Fixed-length arrays where each element can have a different type.
    - function: Represents functions, including their parameters and return types.
- **Special Types:**
    - any: Disables type checking (use sparingly!).
    - void: Represents the absence of a return value from a function.
    - never: Represents values that never occur (e.g., a function that always throws an exception).
    - unknown: Represents a value whose type is not known at compile time. It forces you to perform type checking before you can use the value.

- **Enum:** A way to define a set of named constants.

2. Type Annotations:

Type annotations explicitly specify the type of a variable, function parameter, or function return value. They are written after the variable or parameter name, followed by a colon and the type.

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

Type annotations are crucial for TypeScript's static type checking. The compiler uses them to catch type errors *before* runtime.

3. Compiling TypeScript Files:

TypeScript files (.ts) need to be compiled into JavaScript files (.js) that browsers or Node.js can understand. You use the TypeScript compiler (tsc) for this.

- **Command Line:** tsc filename.tscompiles filename.ts. tscwithout a filename compiles all .tsfiles in the current directory (or as configured in a tsconfig.jsonfile).
- **tsconfig.json:** A configuration file that lets you customize the compilation process (output directory, target JavaScript version, etc.). It's highly recommended to use a tsconfig.jsonfile.

4. JavaScript vs. TypeScript:

- **Type System:** JavaScript is dynamically typed (types are checked at runtime), while TypeScript is statically typed (types are checked at compile time). This is the biggest difference.
- **Compilation:** TypeScript needs to be compiled to JavaScript before it can be run. JavaScript can be run directly in browsers or Node.js.

- **Error Detection:** TypeScript catches type errors early during development, leading to more robust code. JavaScript errors related to types are only discovered at runtime.
- **Code Maintainability:** TypeScript's type system makes it easier to understand and maintain large codebases.
- **Features:** TypeScript often supports newer JavaScript features earlier than some browsers. It can also transpile down to older JavaScript versions for compatibility.

5. Inheritance in JavaScript vs. TypeScript:

- **JavaScript (Prototypal Inheritance):** JavaScript uses prototypes for inheritance. Objects inherit properties and methods from other objects (their prototypes). This is a dynamic mechanism. It can be more difficult to reason about in complex scenarios.
- **TypeScript (Class-based Inheritance):** TypeScript (and modern JavaScript) supports class-based inheritance, which is a more familiar pattern for developers coming from languages like Java or C++. TypeScript classes are compiled down to JavaScript prototype-based code under the hood. It provides a more structured and predictable approach to inheritance.

6. Generics:

Generics allow you to write reusable components that can work with a variety of types without sacrificing type safety. Instead of using any, which bypasses type checking, you use type parameters to specify the type at the time the component is used.

- **Flexibility:** Generics make your code adaptable to different data types. You write the code once, and it works with numbers, strings, custom objects, etc.
- **Type Safety:** Generics preserve type information. The compiler knows the specific type you're working with, so it can perform type checks and prevent errors.

Why Generics are better than any (and why they are suitable in the scenario you mentioned):

- **anydefeats the purpose of TypeScript.** It essentially turns off type checking, so you lose all the benefits of static typing. Errors that would be caught by the compiler are now only found at runtime.
- **Generics maintain type safety.** They allow you to work with different types *while* preserving type information. If you use any, you lose track of the actual type, making your code more error-prone.

In your lab assignment, using generics is likely more suitable because you want to handle input of various types (e.g., numbers, strings, or custom objects). Using any would make the code less type-safe and harder to maintain. Generics let you write a single function or component that can handle different input types while still ensuring type safety.

7. Classes vs. Interfaces:

- **Classes:** Define the blueprint for creating objects (instances). They can contain properties (data) and methods (functions). Classes can be instantiated using the new keyword. Classes can implement interfaces and also extend other classes.
- **Interfaces:** Define a *contract* or *shape* for an object. They specify the properties and methods that an object *must* have. Interfaces cannot be instantiated directly. They are used to enforce type compatibility. Classes *implement* interfaces.

Where Interfaces are Used:

- **Defining the shape of objects:** Interfaces are used to ensure that objects have the required properties and methods.
- **Enforcing contracts:** They specify the requirements that classes must meet.
- **Improving code reusability:** Interfaces can be used to create more generic functions and components.
- **Working with different types:** Interfaces can be used to define the shape of complex data structures.

4. **Output:**

a.

```
// Base Class: Student
class Student {
   name: string;
   studentId: string;
   grade: string;
   libraryAccount: LibraryAccount | null; // Composition: Student
has a LibraryAccount

   constructor(name: string, studentId: string, grade: string) {
      this.name = name;
      this.studentId = studentId;
      this.grade = grade;
      this.libraryAccount = null; // Initially no library account
   }

   getDetails(): void {
      console.log(`Student ID: ${this.studentId}`);
      console.log(`Name: ${this.name}`);
      console.log(`Grade: ${this.grade}`);
      if (this.libraryAccount) {
         console.log("Associated Library Account:");
         this.libraryAccount.getLibraryInfo();
      }
   }

   assignLibraryAccount(account: LibraryAccount): void {
      this.libraryAccount = account;
   }
}

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

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

```
    getThesisTopic(): void {
        console.log(`Thesis Topic: ${this.thesisTopic}`);
    }

    // Override getDetails()
    getDetails(): void {
        console.log("--- Graduate Student Details ---");
        super.getDetails(); // Call the parent's getDetails()
        this.getThesisTopic();
    }
}

// Non-Subclass: LibraryAccount
class LibraryAccount {
    accountId: string;
    booksIssued: number;

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

    getLibraryInfo(): void {
        console.log(`Account ID: ${this.accountId}`);
        console.log(`Books Issued: ${this.booksIssued}`);
    }
}

// Example Usage:

// Create a LibraryAccount
const libraryAccount1 = new LibraryAccount("LA001", 3);

// Create a Student
const student1 = new Student("Aryan", "S001", "10th");
student1.getDetails(); // Display student information
student1.assignLibraryAccount(libraryAccount1);
console.log("\nAfter assigning library account:");
student1.getDetails();

console.log("\n ----------------------------- \n");
```

```
// Create a GraduateStudent
const graduateStudent1 = new GraduateStudent("Niraj", "GS001", "Masters", "AI in Healthcare");
graduateStudent1.getDetails(); // Display graduate student information (overridden)

console.log("\n ----------------------------- \n");

// Create another LibraryAccount
const libraryAccount2 = new LibraryAccount("LA002", 5);

// Create another Student
const student2 = new Student("Vaishnal", "S002", "11th");
student2.assignLibraryAccount(libraryAccount2);
student2.getDetails();

console.log("\n ----------------------------- \n");

// Create another GraduateStudent
const graduateStudent2 = new GraduateStudent("Rahul", "GS002", "PhD", "Quantum Computing");
graduateStudent2.assignLibraryAccount(libraryAccount2);
graduateStudent2.getDetails();

console.log("\n ----------------------------- \n");
```

OUTPUT:-

```
PS C:\Users\aryan\WebX\Exp2> npx tsc student.ts
PS C:\Users\aryan\WebX\Exp2> node student.js
Student ID: S001
Name: Aryan
Grade: 10th

After assigning library account:
Student ID: S001
Name: Aryan
Grade: 10th
Associated Library Account:
Account ID: LA001
Books Issued: 3
```

b.

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

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

  constructor(name: string, id: string, department: string) {
    this.name = name;
    this.id = id;
    this.role = "Manager";
    this.department = department;
  }

  getDetails(): string {
    return `
    --- Manager Details ---
    ID: ${this.id}
    Name: ${this.name}
```

```
    Role: ${this.role}
    Department: ${this.department}
    `;
  }
}

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

  constructor(name: string, id: string, programmingLanguages: string[]) {
    this.name = name;
    this.id = id;
    this.role = "Developer";
    this.programmingLanguages = programmingLanguages;
  }

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

// Example Usage:

// Create a Manager instance
const manager1 = new Manager("Alice Smith", "M001", "Sales");
console.log(manager1.getDetails());

// Create a Developer instance
const developer1 = new Developer("Bob Johnson", "D001", ["TypeScript", "JavaScript", "React"]);
console.log(developer1.getDetails());

// Create another Manager instance
const manager2 = new Manager("Charlie Brown", "M002", "Marketing");
console.log(manager2.getDetails());

// Create another Developer instance
const developer2 = new Developer("David Lee", "D002", ["Java", "Spring Boot", "SQL"]);
console.log(developer2.getDetails());
```

OUTPUT:-

```
• PS C:\Users\aryan\WebX\Exp2> npx tsc employee.ts
• PS C:\Users\aryan\WebX\Exp2> node employee.js

--- Manager Details --- ID: M001
Name: Alice Smith

Role: Manager
Department: Sales


--- Developer Details --- ID: D001
Name: Bob Johnson Role: Developer
Programming Languages: TypeScript, JavaScript, React
```