**Experiment – 1 b: TypeScript**

| | |
|---|---|
| **Name of Student** | **Prajjwal Pandey** |
| **Class Roll No** | **32** |
| **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:**

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

TypeScript supports the following data types:

- **Primitive Types:** number, string, boolean, null, undefined, bigint, symbol
- **Object Types:** object, array, tuple
- **Special Types:** any, unknown, never, void
- **Custom Types:** enum, union, intersection, type alias

**Type Annotations** in TypeScript are used to explicitly define the type of a variable, function parameter, or return value. Example:

```
let age: number = 25;

function greet(name: string): string {
    return `Hello, ${name}`;
}
```

---

**b. How do you compile TypeScript files?**

To compile a TypeScript file (.ts), use the TypeScript compiler (tsc).

```
tsc filename.ts
```

This generates a JavaScript file (filename.js). You can then run it using Node.js:

```
node filename.js
```

---

**c. What is the difference between JavaScript and TypeScript?**

| Feature | JavaScript | TypeScript |
| --- | --- | --- |
| Type System | Dynamic (no type checking) | Static (type checking at compile-time) |
| Compilation | Interpreted | Compiled to JavaScript |
| Error Detection | Runtime errors | Errors detected at compile-time |
| Object-Oriented Features | Limited | Supports interfaces, generics, and access modifiers |
| Usage | Used in browsers and Node.js | Primarily used for large-scale applications |

---

**d. Compare how JavaScript and TypeScript implement Inheritance.**

Both JavaScript and TypeScript use class-based inheritance with the extends keyword.

**JavaScript Inheritance:**

```
class Parent {
    constructor(name) {
        this.name = name;
    }
    greet() {
        console.log(`Hello, ${this.name}`);
    }
```

```
}

class Child extends Parent {}

const child = new Child("Alice");

child.greet();
```

**TypeScript Inheritance (With Type Safety):**

```
class Parent {
    constructor(public name: string) {}
    greet(): void {
        console.log(`Hello, ${this.name}`);
    }
}

class Child extends Parent {}

const child = new Child("Alice");

child.greet();
```

The difference is that TypeScript enforces type safety, ensuring proper usage of class members.

---

**e. How generics make the code flexible and why we should use generics over other types.**

Generics allow writing **reusable and type-safe** code. Instead of using any (which removes type safety), generics allow defining a placeholder type that can be replaced when calling the function or class.

**Example Without Generics:**

```
function identity(arg: any): any {
    return arg;
```

```
}
```

This loses type safety since arg can be anything.

**Example With Generics:**

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

console.log(identity<number>(10));

console.log(identity<string>("Hello"));
```

Generics ensure that the input and output maintain the same type, preventing runtime errors.

**Why Use Generics in Lab Assignment 3?** In Lab Assignment 3, generics are more suitable because they ensure type consistency while handling inputs dynamically, rather than using any, which could lead to unexpected errors.

---

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

| Feature | Classes | Interfaces |
| --- | --- | --- |
| Definition | Blueprint for creating objects | Defines a contract for object shape |
| Implementation | Can have methods and properties | Only defines structure, no implementation |
| Instantiation | Can be instantiated using new | Cannot be instantiated |

| Feature | Classes | Interfaces |
| --- | --- | --- |
| Supports Methods | Yes, with implementations | Only method signatures, no body |
| Extends | Can extend other classes | Can extend other interfaces |

**Example of Class:**

```
class Animal {
    constructor(public name: string) {}
    speak(): void {
        console.log("Animal sound");
    }
}
```

**Example of Interface:**

```
interface Person {
    name: string;
    age: number;
}
let student: Person = { name: "John", age: 22 };
```

**Where Are Interfaces Used?**

- Defining function signatures
- Enforcing object structure
- Implementing multiple types in a class
- Making code flexible and scalable

4. **Output:**

    a. Source code :

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

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

// Demonstration
const student = new Student("Alice", 101, "A");
const gradStudent = new GraduateStudent("Bob", 102, "A+", "Artificial Intelligence");
const libraryAccount = new LibraryAccount(5001, 3);
```

```
39.
40. console.log(student.getDetails());
41. console.log(gradStudent.getDetails());
42. console.log(gradStudent.getThesisTopic());
43. console.log(libraryAccount.getLibraryInfo());
```

output:

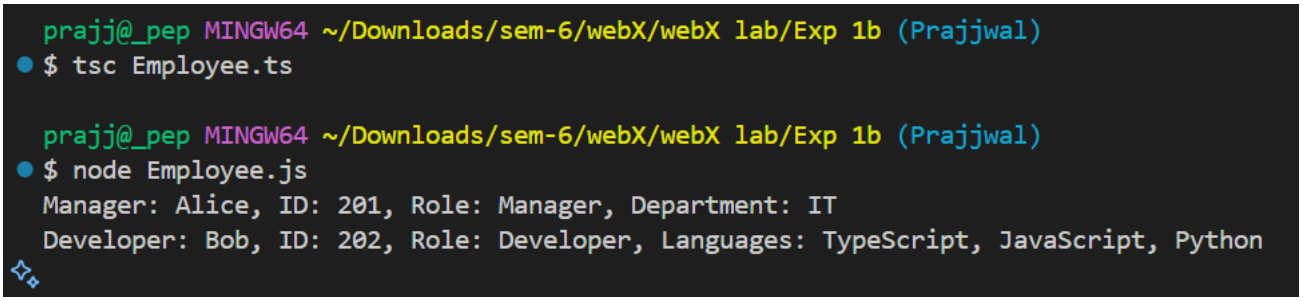
```
prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1b (Prajjwal)
$ tsc Student.ts

prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1b (Prajjwal)
$ node Student.js
Student: Alice, ID: 101, Grade: A
Graduate Student: Bob, ID: 102, Grade: A+, Thesis: Artificial Intelligence
Thesis Topic: Artificial Intelligence
Library Account ID: 5001, Books Issued: 3
```

b. source code:

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

// Demonstration
const manager = new Manager("Alice", 201, "Manager", "IT");
const developer = new Developer("Bob", 202, "Developer", ["TypeScript", "JavaScript", "Python"]);

console.log(manager.getDetails());
console.log(developer.getDetails());
```

output:


prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1b (Prajjwal)
$ tsc Employee.ts
prajj@_pep MINGW64 ~/Downloads/sem-6/webX/webX lab/Exp 1b (Prajjwal)
$ node Employee.js
Manager: Alice, ID: 201, Role: Manager, Department: IT
Developer: Bob, ID: 202, Role: Developer, Languages: TypeScript, JavaScript, Python