Experiment – 1 b: TypeScript

Name of Student	Jai Talreja
Class Roll No	59
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?
    b. How do you compile TypeScript files?
    c. What is the difference between JavaScript and TypeScript?
    d. Compare how Javascript and Typescript implement Inheritance.
    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.
    f. What is the difference between Classes and Interfaces in Typescript? Where are interfaces used?

a. What are the different data types in TypeScript? What are Type Annotations in TypeScript?
Data Types in TypeScript:

TypeScript builds on JavaScript by adding static types. Here are the main data types:

Primitive Types:
number: Represents numbers (e.g., 1, 2.5).
string: Represents strings of characters (e.g., "hello", 'world').
boolean: Represents true or false values.
void: Represents the absence of any type (often used for functions that do not return a value).
null: Represents a null value.
undefined: Represents an undefined value.

Special Types:
any: A type that can hold any type of value. It's like opting out of type checking for that variable.
never: Represents values that will never occur, such as a function that throws an error.
unknown: Similar to any, but more type-safe. You need to perform some checks before operating on it.

Object Types:
Object: Represents any non-primitive type (not number, string, boolean, null, or undefined).
Array: Represents an array of values (e.g., number[] or Array<number>).
Tuple: Represents a fixed-size array with elements of different types (e.g., [string, number]).
Enum: A way to define named constant values (e.g., enum Color { Red, Green, Blue }).

Type Annotations in TypeScript:

Type annotations in TypeScript are used to explicitly define the type of a variable, function parameter, or function return value. This allows TypeScript to perform static type checking and catch potential errors during development.

Example:

```
let age: number = 30;
let name: string = "Alice";
```

Functions can also have type annotations:

```
function greet(name: string): string {
  return "Hello " + name;
}
```

b. How do you compile TypeScript files?

To compile TypeScript files, you need to follow these steps:

Install TypeScript: If TypeScript is not installed globally, you can install it using npm:

```
npm install -g typescript
```

Compile using tsc: After installing TypeScript, you can use the TypeScript compiler (tsc) to compile .ts files into JavaScript. For example, to compile a single file:

```
tsc filename.ts
```

This will generate a filename.js file.

Compile Multiple Files: You can compile multiple files by specifying them together:

```
tsc file1.ts file2.ts
```

Using tsconfig.json: For more complex projects, you can set up a tsconfig.json file to specify compiler options. Run:

```
tsc --init
```

Then, use tsc in the root directory to compile the project according to the settings in the tsconfig.json.

c. What is the difference between JavaScript and TypeScript?

JavaScript and TypeScript have several key differences:

Typing:
JavaScript is dynamically typed, meaning types are inferred during runtime.
TypeScript is statically typed, meaning types are checked at compile-time.

Syntax:
TypeScript extends JavaScript by adding type annotations and other features, but you can still use regular JavaScript syntax.

Type Safety:
TypeScript enforces type safety and provides features like interfaces, enums, and type annotations, helping avoid common runtime errors.

Tooling:
TypeScript offers better IDE support, including autocompletion, type inference, and refactoring capabilities, thanks to its static typing.

Compilation:
TypeScript needs to be compiled into JavaScript using the TypeScript compiler (tsc), whereas JavaScript can be executed directly by browsers or Node.js.

OOP Features:
TypeScript supports modern Object-Oriented Programming (OOP) features like classes, interfaces, and access modifiers, which are more limited or absent in JavaScript.

d. Compare how JavaScript and TypeScript implement Inheritance.

Both JavaScript and TypeScript support inheritance, but they implement it in slightly different ways.

JavaScript:
JavaScript uses prototypal inheritance, where objects inherit properties and methods from other objects. Classes in JavaScript were introduced in ECMAScript 6 (ES6), but behind the scenes, they still use prototypal inheritance.

Example (ES6 class):

```
class Animal {
  constructor(name) {
```

```
    this.name = name;
  }

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}
```

TypeScript:

TypeScript works similarly to JavaScript with regard to inheritance. However, TypeScript adds type checking and interfaces to enhance the structure of the code.

TypeScript supports both class-based and interface-based inheritance, which allows you to define types for classes and objects more strictly.

Example:

```
class Animal {
  constructor(public name: string) {}

  speak(): void {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak(): void {
    console.log(this.name + ' barks.');
  }
}
```

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

Generics allow you to write functions, classes, and interfaces that can work with any data type, while maintaining type safety. They enable flexibility and reusability without sacrificing the benefits of static typing.
Benefits of Generics:

Flexibility: You can write code that works with any data type (e.g., a function that works with both number and string).
Type Safety: Generics ensure that types are consistent throughout the code and catch potential errors at compile-time.

Why Generics over any:

The any type allows you to bypass type checking, which defeats the purpose of TypeScript's static type checking.
Generics enforce constraints on the type while still allowing flexibility, whereas any can lead to runtime errors due to lack of type safety.

Example of Generics vs. any:

```
// Using `any`
function getItem(value: any) {
  return value;
}

// Using Generics
function getItem<T>(value: T): T {
  return value;
}
```

With generics, you ensure that the returned type matches the input type, which is safer than using any.

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

Classes:

Classes are blueprints for creating objects with properties and methods. They can contain implementation details and are used to define the structure of objects.
Classes in TypeScript support access modifiers (public, private, protected), inheritance, and method implementations.

Example:

```
class Car {
  constructor(private model: string) {}

  getModel(): string {
    return this.model;
  }
}
```

Interfaces:

Interfaces define the structure of an object or class but do not contain implementation. They are used to define the shape of an object, specifying what properties and methods it should have.
Interfaces help ensure that a class or object adheres to a specific structure.

Example:

```
interface Vehicle {
  model: string;
  start(): void;
}

class Car implements Vehicle {
  constructor(public model: string) {}

  start(): void {
    console.log(`${this.model} is starting.`);
  }
}
```

Where are Interfaces used?

Type checking: They ensure that a class or object meets a specific structure.
Object Shape Definition: Interfaces are useful when you want to define the structure of objects or when creating contracts for classes to implement.
Function Signatures: Interfaces can define the expected structure of functions or methods.

4. **Output:**

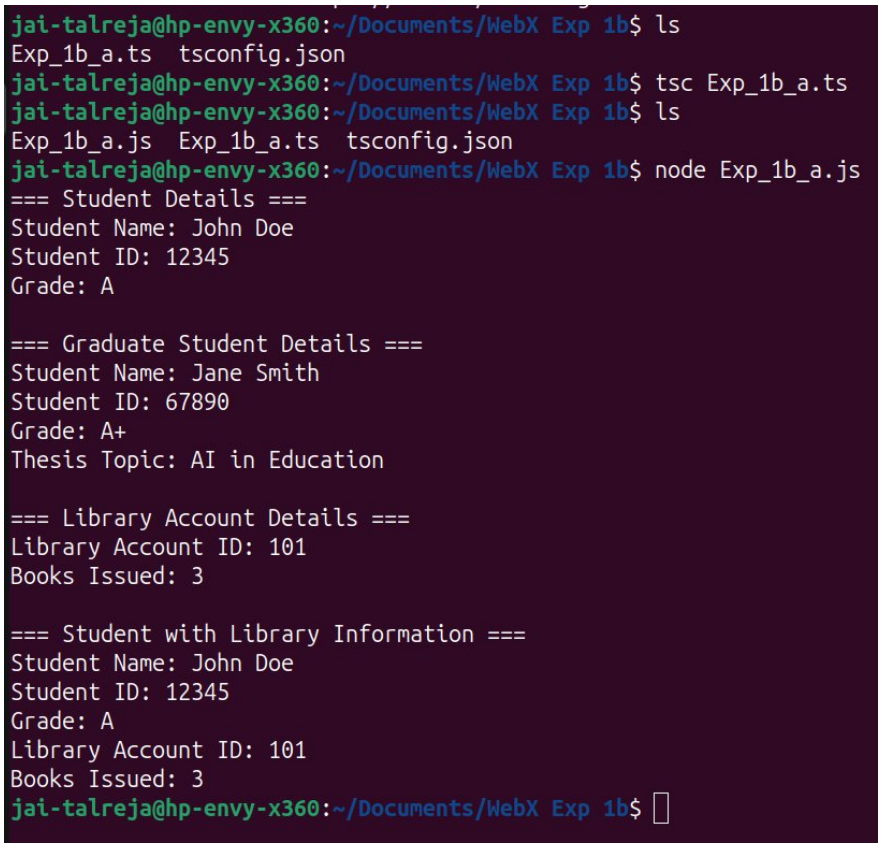
[a. Student, Graduate and Library]

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

    getDetails(): void {
      console.log(`Student Name: ${this.name}`);
      console.log(`Student ID: ${this.studentId}`);
      console.log(`Grade: ${this.grade}`);
    }
  }

  // Subclass GraduateStudent that extends Student
  class GraduateStudent extends Student {
    constructor(
      name: string,
      studentId: number,
      grade: string,
      public thesisTopic: string
    ) {
```

```
    super(name, studentId, grade); // Call the parent class constructor
  }

  // Override getDetails() method to display additional information
  getDetails(): void {
    super.getDetails(); // Call the base class method
    console.log(`Thesis Topic: ${this.thesisTopic}`);
  }

  // New method for GraduateStudent
  getThesisTopic(): string {
    return this.thesisTopic;
  }
}

// Non-subclass LibraryAccount
class LibraryAccount {
  constructor(
    public accountId: number,
    public booksIssued: number
  ) {}

  getLibraryInfo(): void {
```

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

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

  // Combined method to show student's details and library account info
  getFullInfo(): void {
    this.student.getDetails(); // Get student details
    this.libraryAccount.getLibraryInfo(); // Get library account info
  }
}

// Creating instances of each class
const student1 = new Student('John Doe', 12345, 'A');
const gradStudent1 = new GraduateStudent('Jane Smith', 67890, 'A+', 'AI in Education');
const libraryAccount1 = new LibraryAccount(101, 3);
```

```
// Creating a StudentWithLibrary instance demonstrating composition
const studentWithLibrary1 = new StudentWithLibrary(student1, libraryAccount1);

// Calling methods
console.log("=== Student Details ===");
student1.getDetails(); // Calls Student's getDetails()

console.log("\n=== Graduate Student Details ===");
gradStudent1.getDetails(); // Calls GraduateStudent's overridden getDetails()

console.log("\n=== Library Account Details ===");
libraryAccount1.getLibraryInfo(); // Calls LibraryAccount's getLibraryInfo()

console.log("\n=== Student with Library Information ===");
studentWithLibrary1.getFullInfo(); // Calls combined getFullInfo method

//Program ends here.
```



[b. Employee management system]

```
// Employee Interface
interface Employee {
  name: string;
  id: number;
  role: string;
  getDetails(): string;  // Method to get employee details
}

// Manager class implementing Employee interface
class Manager implements Employee {
```

```
  constructor(
    public name: string,
    public id: number,
    public role: string,
    public department: string
  ) {}

  // Override getDetails method to include department information
  getDetails(): string {
    return `${this.name} (ID: ${this.id}) is a ${this.role} in the ${this.department} department.`;
  }
}

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

  // Override getDetails method to include programming languages
```

```
  getDetails(): string {

   return `${this.name} (ID: ${this.id}) is a ${this.role} skilled in ${this.programmingLanguages.join(", ")}.`;

  }

}


// Creating instances of Manager and Developer

const manager1 = new Manager("Alice Johnson", 101, "Manager", "HR");

const developer1 = new Developer("Bob Smith", 102, "Developer", ["JavaScript", "TypeScript", "Node.js"]);


// Displaying the details using getDetails() method

console.log(manager1.getDetails());

console.log(developer1.getDetails());

//Program ends here.
```