Experiment – 1 b: TypeScript

Name of Student	**<u>Vedang Wajge</u>**
Class Roll No	**<u>D15A 62</u>**
D.O.P.	
D.O.S.	
Sign and Grade	

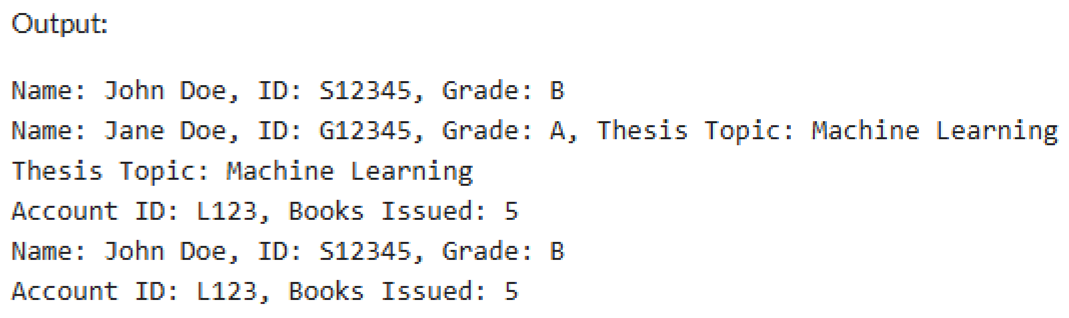
Aim: To study Basic constructs in TypeScript.

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.

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

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

```
// Subclass: GraduateStudent (extends Student)
class GraduateStudent extends Student {
constructor(
    name: string,
    studentId: string,
    grade: string,
    public thesisTopic: string
  ) {
    super(name, studentId, grade); // Call parent constructor
  }

  // Override the getDetails method
  getDetails(): void {
       console.log(`Name: ${this.name}, ID: ${this.studentId}, Grade: ${this.grade}, Thesis
Topic: ${this.thesisTopic}`);
  }

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

// Non-Subclass: LibraryAccount (Composition over Inheritance)
class LibraryAccount {
  constructor(
    public accountId: string,
    public booksIssued: number
  ) {}

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

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

  // Call the getDetails method from the student object
  displayStudentInfo(): void {
    this.student.getDetails();
  }

  // Call the getLibraryInfo method from the libraryAccount object
  displayLibraryInfo(): void {
    this.libraryAccount.getLibraryInfo();
  }
}

// Create instances of Student, GraduateStudent, and LibraryAccount
const student = new Student("John Doe", "S12345", "B");
const gradStudent = new GraduateStudent("Jane Doe", "G12345", "A",
"Machine Learning");
const libraryAccount = new LibraryAccount("L123", 5);

// Demonstrate behavior of classes
student.getDetails();
gradStudent.getDetails();
gradStudent.getThesisTopic();
libraryAccount.getLibraryInfo();

// Demonstrate Composition over Inheritance
const studentWithLibrary = new StudentWithLibraryAccount(student, libraryAccount);
studentWithLibrary.displayStudentInfo(); // From Student
studentWithLibrary.displayLibraryInfo(); // From LibraryAccount
```

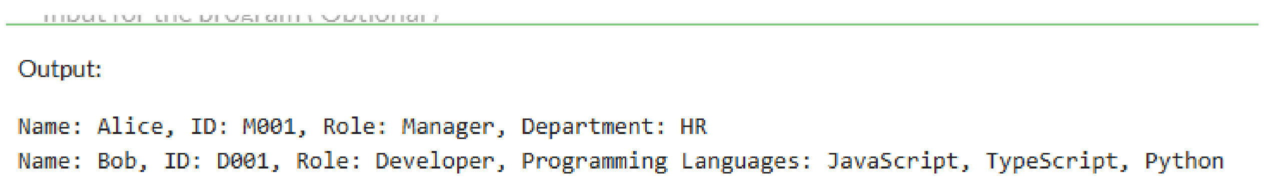


Output:

```
Name: John Doe, ID: S12345, Grade: B
Name: Jane Doe, ID: G12345, Grade: A, Thesis Topic: Machine Learning
Thesis Topic: Machine Learning
Account ID: L123, Books Issued: 5
Name: John Doe, ID: S12345, Grade: B
Account ID: L123, Books Issued: 5
```

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.

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

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

  // Override getDetails method to include department
  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: string,
    public role: string,
    public programmingLanguages: string[]
  ) {}

  // Override getDetails method to include programming languages
  getDetails(): string {
    return `Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Programming Languages:
${this.programmingLanguages.join(", ")}`;
  }
}

// Create instances of Manager and Developer
const manager = new Manager("Alice", "M001", "Manager", "HR");
const developer = new Developer("Bob", "D001", "Developer", ["JavaScript", "TypeScript",
"Python"]);

// Display details using getDetails() method
console.log(manager.getDetails());
console.log(developer.getDetails());
```



Output:

```
Name: Alice, ID: M001, Role: Manager, Department: HR
Name: Bob, ID: D001, Role: Developer, Programming Languages: JavaScript, TypeScript, Python
```

Conclusion

This experiment explored key TypeScript concepts like **inheritance, interfaces, and composition**. We demonstrated **inheritance** with `GraduateStudent`, **composition over inheritance** with `LibraryAccount`, and **interfaces** with `Employee`. By implementing **method overriding** and structured class design, we highlighted the benefits of **code reusability, flexibility, and maintainability** in TypeScript.