Experiment – 1 b: TypeScript

Name of Student	**Snehal Anilkumar Patil**
Class Roll No	**D15A / 38**
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.

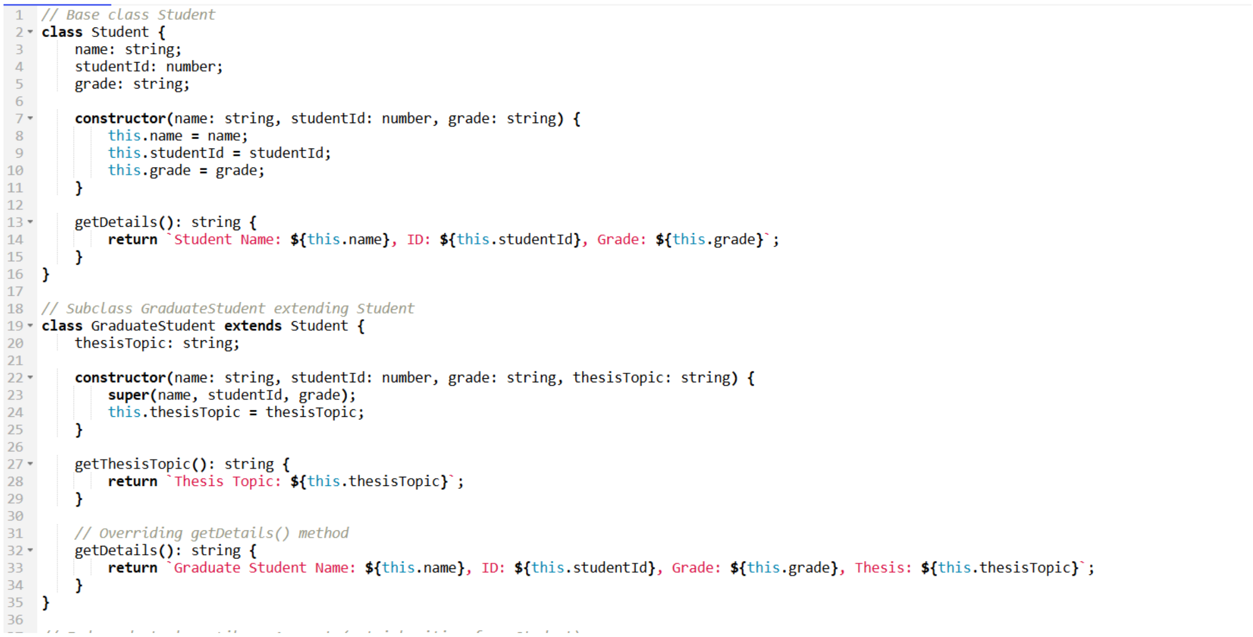
   Create a non-subclass **LibraryAccount** (which does not inherit from Student) with properties like accountId, booksIssued, and a method getLibraryInfo().
   Demonstrate composition over inheritance by associating a LibraryAccount object with a Student object instead of inheriting from Student.
   Create 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.

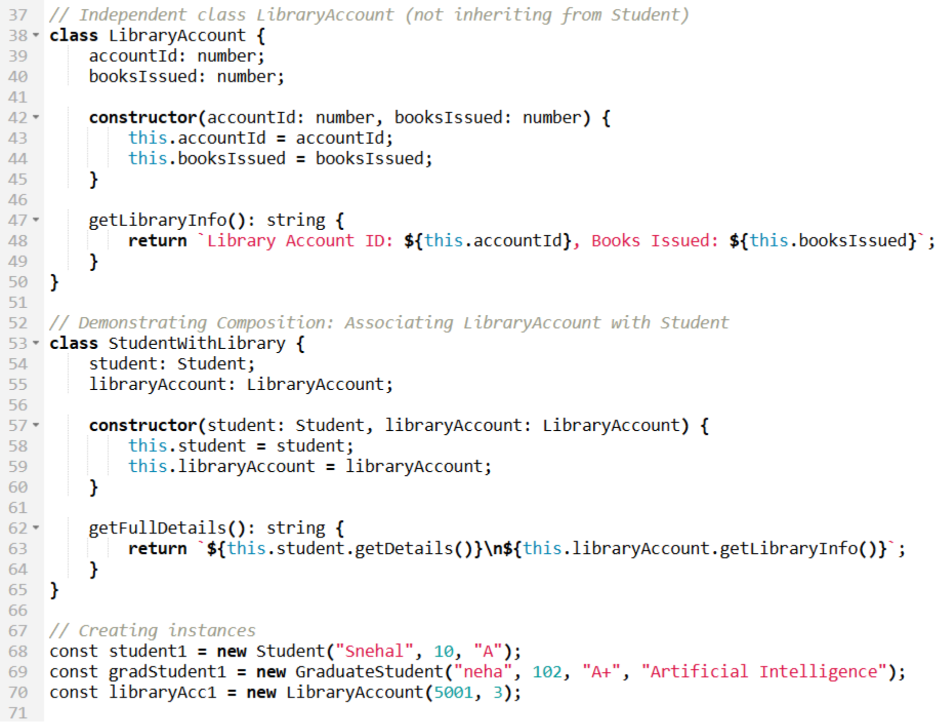
3. **Theory:**
    a. What are the different data types in TypeScript? What are Type Annotations in Typescript?
    b. How do you compile TypeScript files?
    c. What is the difference between JavaScript and TypeScript?
    d. Compare how Javascript and Typescript implement Inheritance.
    e. How generics 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?

4. **Output:**

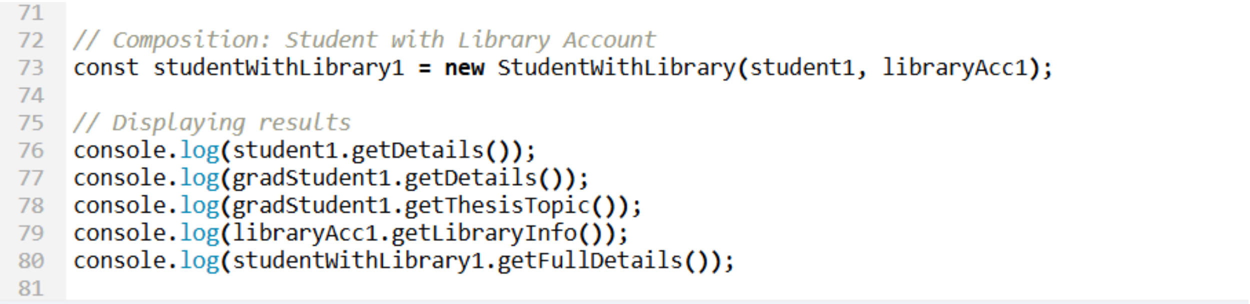
1.



```
// Base class Student
class Student {
    name: string;
    studentId: number;
    grade: string;

    constructor(name: string, studentId: number, grade: string) {
        this.name = name;
        this.studentId = studentId;
        this.grade = grade;
    }

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

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

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

    getThesisTopic(): string {
        return `Thesis Topic: ${this.thesisTopic}`;
    }

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

```

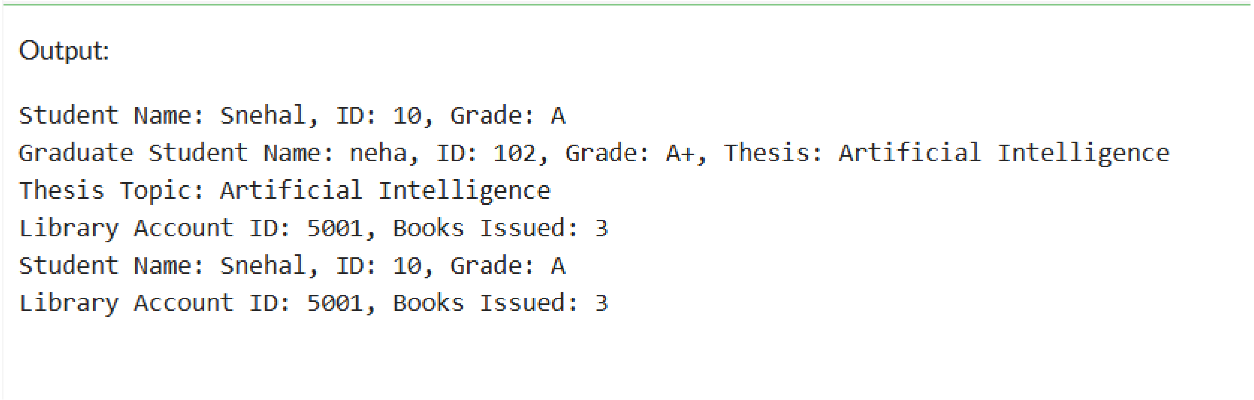


```typescript
// Independent class LibraryAccount (not inheriting from Student)
class LibraryAccount {
    accountId: number;
    booksIssued: number;

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

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

// Demonstrating Composition: Associating LibraryAccount with Student
class StudentWithLibrary {
    student: Student;
    libraryAccount: LibraryAccount;

    constructor(student: Student, libraryAccount: LibraryAccount) {
        this.student = student;
        this.libraryAccount = libraryAccount;
    }

    getFullDetails(): string {
        return `${this.student.getDetails()}\n${this.libraryAccount.getLibraryInfo()}`;
    }
}

// Creating instances
const student1 = new Student("Snehal", 10, "A");
const gradStudent1 = new GraduateStudent("neha", 102, "A+", "Artificial Intelligence");
const libraryAcc1 = new LibraryAccount(5001, 3);

```



```typescript

// Composition: Student with Library Account
const studentWithLibrary1 = new StudentWithLibrary(student1, libraryAcc1);

// Displaying results
console.log(student1.getDetails());
console.log(gradStudent1.getDetails());
console.log(gradStudent1.getThesisTopic());
console.log(libraryAcc1.getLibraryInfo());
console.log(studentWithLibrary1.getFullDetails());

```

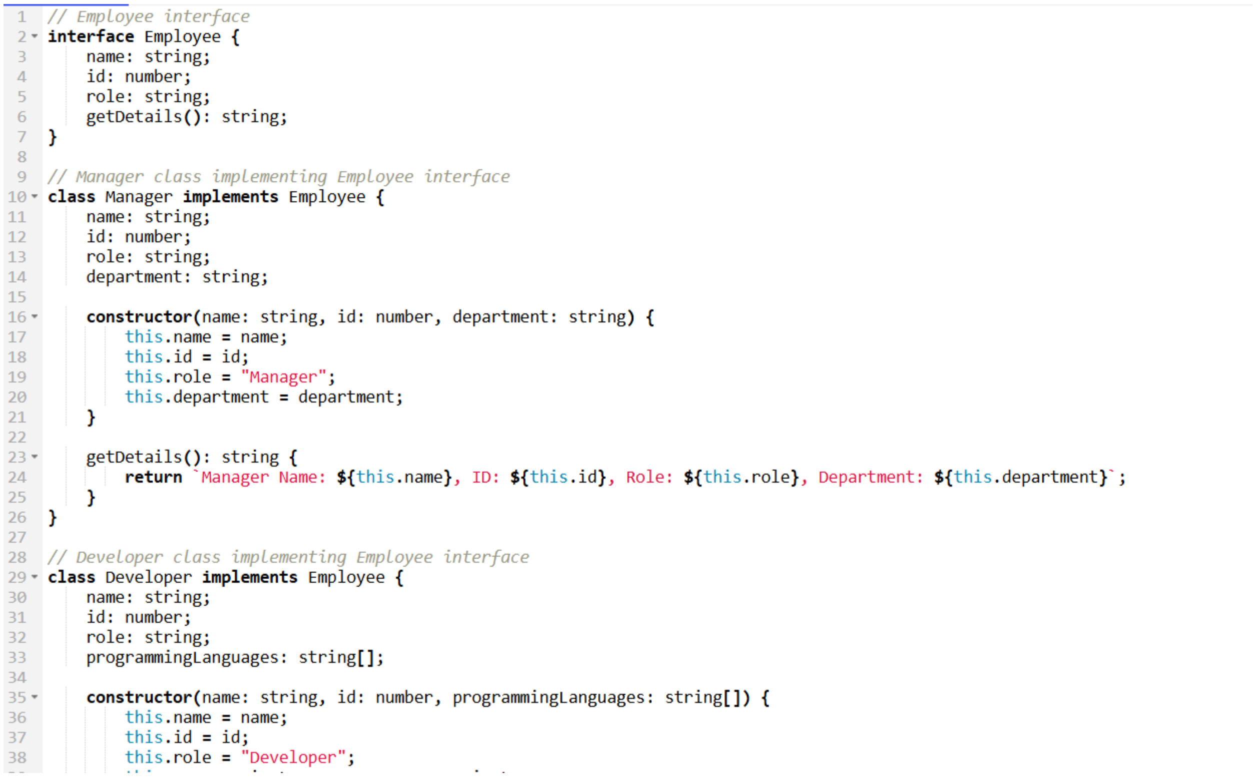
OUTPUT:



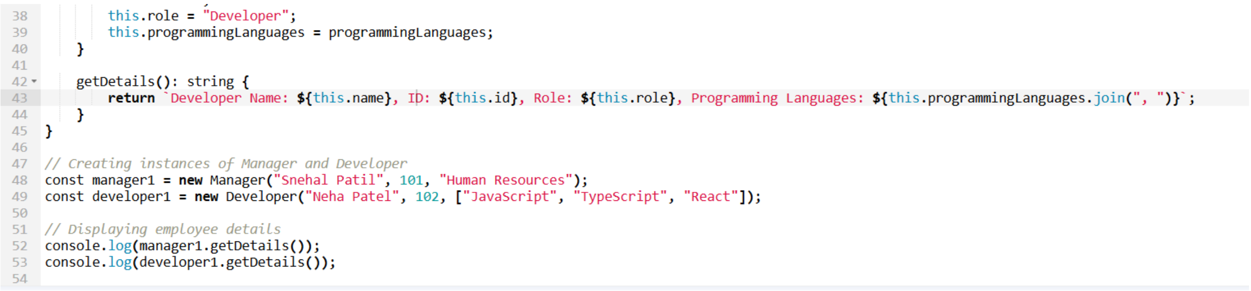
Output:

```
Student Name: Snehal, ID: 10, Grade: A
Graduate Student Name: neha, ID: 102, Grade: A+, Thesis: Artificial Intelligence
Thesis Topic: Artificial Intelligence
Library Account ID: 5001, Books Issued: 3
Student Name: Snehal, ID: 10, Grade: A
Library Account ID: 5001, Books Issued: 3
```

2.

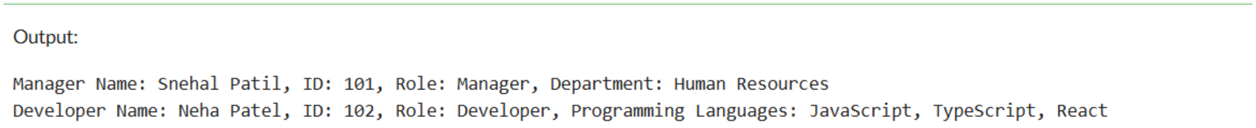


```typescript
// Employee interface
interface Employee {
    name: string;
    id: number;
    role: string;
    getDetails(): string;
}

// Manager class implementing Employee interface
class Manager implements Employee {
    name: string;
    id: number;
    role: string;
    department: string;

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

    getDetails(): string {
        return `Manager Name: ${this.name}, ID: ${this.id}, Role: ${this.role}, Department: ${this.department}`;
    }
}

// Developer class implementing Employee interface
class Developer implements Employee {
    name: string;
    id: number;
    role: string;
    programmingLanguages: string[];

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



```typescript
        this.role = "Developer";
        this.programmingLanguages = programmingLanguages;
    }

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

// Creating instances of Manager and Developer
const manager1 = new Manager("Snehal Patil", 101, "Human Resources");
const developer1 = new Developer("Neha Patel", 102, ["JavaScript", "TypeScript", "React"]);

// Displaying employee details
console.log(manager1.getDetails());
console.log(developer1.getDetails());
```

OUTPUT:



Output:

```
Manager Name: Snehal Patil, ID: 101, Role: Manager, Department: Human Resources
Developer Name: Neha Patel, ID: 102, Role: Developer, Programming Languages: JavaScript, TypeScript, React
```

3. **Theory:**

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

Ans:

Different Data Types in TypeScript

TypeScript supports several data types, including:

- **Primitive Types:** number, string, boolean, bigint, symbol, null, undefined
- **User-defined Types:** interface, class, enum, type
- **Advanced Types:** any, unknown, never, tuple, union, intersection, void

2. Type Annotations in TypeScript

Type annotations allow you to specify the type of a variable explicitly, such as:

1. let num: number = 10;

2. let username: string = "Alice";

3. let isActive: boolean = true;

b. How do you compile TypeScript files?

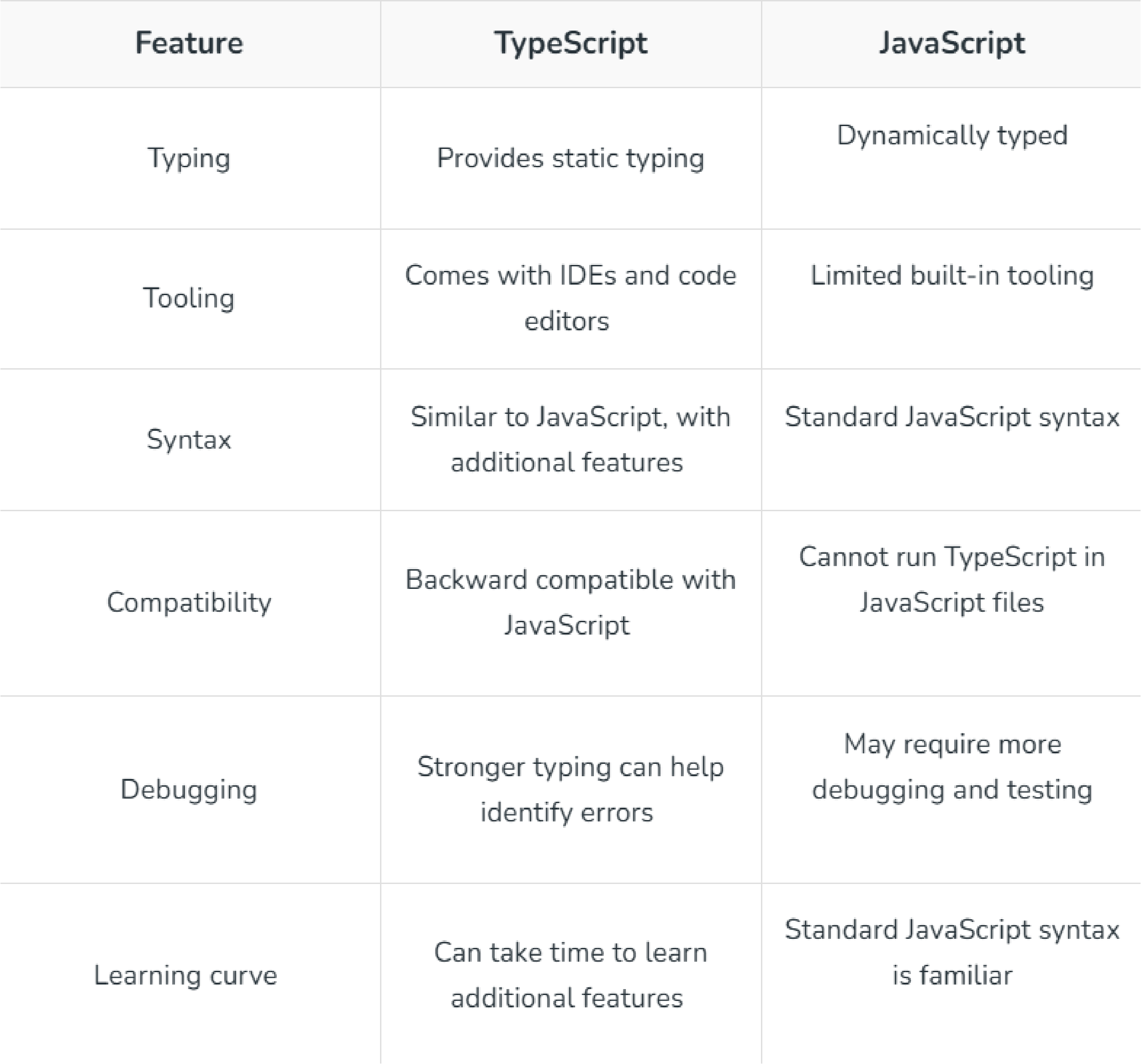
Ans:

To compile a TypeScript file (.ts) into JavaScript (.js), use:

tsc filename.ts

This generates a filename.js file, which can be run in any JavaScript environment.

c. What is the difference between JavaScript and TypeScript?



Feature	TypeScript	JavaScript
Typing	Provides static typing	Dynamically typed
Tooling	Comes with IDEs and code editors	Limited built-in tooling
Syntax	Similar to JavaScript, with additional features	Standard JavaScript syntax
Compatibility	Backward compatible with JavaScript	Cannot run TypeScript in JavaScript files
Debugging	Stronger typing can help identify errors	May require more debugging and testing
Learning curve	Can take time to learn additional features	Standard JavaScript syntax is familiar

d. Compare how Javascript and Typescript implement Inheritance.

Ans:

Inheritance in JavaScript vs. TypeScript

- JavaScript uses **prototypal inheritance**, where objects inherit directly from other objects.

- TypeScript supports **class-based inheritance**, similar to Java, using extends.

Example in TypeScript:

Typescript:

```
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}
class Student extends Person {
    rollNo: number;
    constructor(name: string, rollNo: number) {
        super(name);
        this.rollNo = rollNo;
    }
}
```

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.

Ans:

- Generics make the code reusable and type-safe. Instead of using any, generics ensure that the function or class works with multiple types without losing type safety. Example:

Typescript:

```
function identity<T>(arg: T): T {
   return arg;
}
console.log(identity<number>(10));
console.log(identity<string>("Hello"));
```

Generics are preferred over any because they preserve type information.

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

Ans:

Difference Between Classes and Interfaces

- **Class**: A blueprint for creating objects, supports inheritance.
- **Interface**: Defines a structure but does not provide implementation.

Example:

typescript

```
interface Person {
   name: string;
```

```
    age: number;
}

class Student implements Person {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
}
```

Interfaces are used for type checking and defining the structure of an object without implementing it.