Experiment – 1 b: TypeScript

Name of Student	**Gunjan Chandnani**
Class Roll No	**06**
D.O.P.	**28/01/2025**
D.O.S.	**4/02/2025**
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 and floating-point).
- **string**: For text.
- **boolean**: true or false.
- **null**: Represents the intentional absence of a value.
- **undefined**: Represents a variable that has not been assigned a value.
- **symbol**: Unique and immutable values (often used as keys in objects).
- **bigint**: For arbitrarily large integers.

● **Structural Types:**

- **object**: Represents non-primitive values, like 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 with caution!).
- **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. Safer than any.

● **Type Literals:**

Allow specifying exact values as types (e.g., `let myStatus: "active" | "inactive";`).

- **Union Types:**

Allow a variable to hold values of multiple types (e.g., `let id: number | string;`).

- **Intersection Types:**

Combine multiple types into a single type:

typescript

CopyEdit

```typescript
interface A { a: string; }

interface B { b: number; }

type C = A & B; // C has both a and b
```

- **Generics:**

Allow creating reusable components that can work with a variety of types.

---

2. Type Annotations in TypeScript:

Type annotations are a way to explicitly specify the type of a variable, function parameter, function return value, or property. They are written after a colon `:`.

typescript

CopyEdit

```typescript
let name: string = "Alice"; // Type annotation for a variable


function greet(person: string): string { // Type annotation
for parameter and return value

    return "Hello, " + person;
}


interface Person {
    name: string; // Type annotation for a property
    age: number;
}
```

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`) cannot be directly run by a browser or Node.js. They need to be compiled into JavaScript files (`.js`). The TypeScript compiler (`tsc`) does this.

- **Command Line:**

`tsc myFile.ts` (compiles `myFile.ts` to `myFile.js`).

You can also compile multiple files or use a configuration file (`tsconfig.json`).

- **tsconfig.json:**

A configuration file that lets you customize the compilation process (e.g., output directory, target JavaScript version, strictness settings).
A typical `tsconfig.json` might look like this:

json

CopyEdit

```
{
  "compilerOptions": {
    "target": "es5", // Target JavaScript version
    "outDir": "./dist", // Output directory for compiled files
    "strict": true // Enable strict type checking
  }
}
```

---

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).
- **Compilation**: TypeScript code needs to be compiled into JavaScript before it can be executed. JavaScript runs directly in the browser or Node.js.
- **Features**: TypeScript supports modern JavaScript features (like classes, interfaces, modules) and adds its own (generics, enums, type aliases).
- **Error Detection**: TypeScript's type system helps catch errors early during development, while JavaScript errors are often only discovered at runtime.
- **Maintainability**: TypeScript code is generally easier to maintain and refactor because the types provide a form of documentation and help prevent accidental errors.

---

5. Inheritance in JavaScript and TypeScript:

- **JavaScript (Prototypal Inheritance)**: JavaScript uses prototypal inheritance, where objects inherit properties and methods directly from other objects (prototypes). It's often implemented using constructor functions and the prototype property.

- **TypeScript (Class-based Inheritance)**: TypeScript, like many other object-oriented languages, uses class-based inheritance. Classes are blueprints for creating objects. Inheritance is achieved using the `extends` keyword.

  TypeScript's classes are compiled down to JavaScript that uses prototypes under the hood, but the class syntax makes inheritance easier to work with.

typescript

CopyEdit

```
class Animal {

    name: string;

    constructor(name: string) {

        this.name = name;

    }

    makeSound() {

        console.log("Generic animal sound");

    }

}


class Dog extends Animal { // Dog inherits from Animal

    breed: string;

    constructor(name: string, breed: string) {

        super(name); // Call the parent class constructor

        this.breed = breed;

    }

    makeSound() {

        console.log("Woof!"); // Override the makeSound method
    }
}

let myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Output: Woof!
```

---

6. Generics:

Generics allow you to write reusable components (functions, classes, interfaces) that can work with a variety of types without sacrificing type safety. Instead of using `any` (which disables type checking), you use a type parameter (often `T`) as a placeholder for a specific type that will be determined later when the component is used.

Benefits:

- **Flexibility**: Generics make code more flexible because you can use the same component with different types.
- **Type Safety**: Generics preserve type safety. The compiler will check that you're using the correct types with the generic component.
- **Improved Code Readability**: Generics make code easier to understand because the types are clearly defined.

Why use generics over any in Lab 3 (or similar situations)?

If you use `any`, you're telling TypeScript to ignore type checking for that data, which means:

- You lose the benefits of type checking. Errors related to incorrect data types might not be caught until runtime.
- Your code becomes harder to understand and maintain. It's not clear what types of data the function is supposed to work with.

If you use **generics**, you're saying, "This function can work with different types, but I will specify the type when I use it." This gives you:

- **Type safety**: The compiler will ensure that you're using the correct types.
- **Code clarity**: It's clear what types of data the function can handle.

---

7. Classes vs. Interfaces:

- **Classes**:
    - Are blueprints for creating objects.
    - Can contain properties, methods, and constructors.
    - Can be instantiated (you can create objects from them).
    - Can implement interfaces and extend other classes.
- **Interfaces**:
    - Define a contract or a shape for an object.
    - Specify the properties and methods that an object must have.
    - Cannot be instantiated directly.
    - Classes implement interfaces.

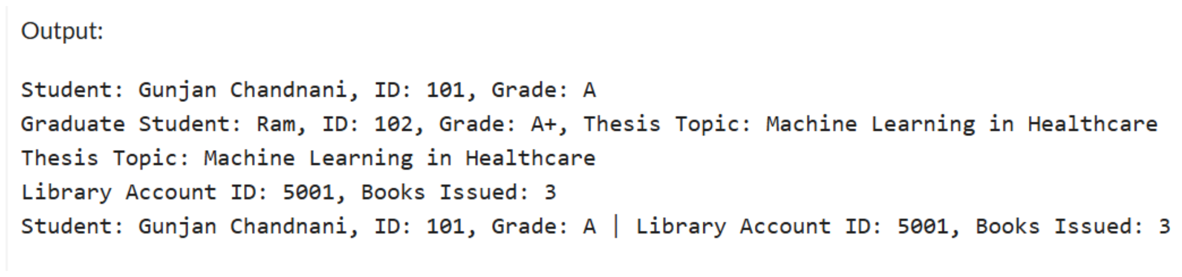
Where are interfaces used?

- **Defining the shape of objects**: Interfaces are commonly used to specify the structure of objects that are passed to functions or returned from functions.
- **Enforcing contracts**: Interfaces ensure that classes that implement them adhere to a certain set of properties and methods.
- **Improving code readability**: Interfaces make code easier to understand by clearly defining the structure of data.
- **Working with different types**: Interfaces can be used with generics to create flexible and type-safe components.

Output:

```
// Base class: Student

class Student {

  constructor(

    public name: string,

    public studentId: number,
        public grade: string
    ) {}

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

// Subclass GraduateStudent extending Student
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 Topic: ${this.thesisTopic}`;
    }
    getThesisTopic(): string {
        return `Thesis Topic: ${this.thesisTopic}`;
    }
}

// Independent class LibraryAccount
class LibraryAccount {
    constructor(
        public accountId: number,
        public booksIssued: number
    ) {}

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

// Demonstrating composition: Student + LibraryAccount
class StudentWithLibrary {
    constructor(
        public student: Student,
        public libraryAccount: LibraryAccount
    ) {}
    getFullDetails(): string {

        return `${this.student.getDetails()} | ${this.libraryAccount.getLibraryInfo()}`;

    }

}


// Creating instances

const student1 = new Student("Gunjan Chandnani", 101, "A");

const gradStudent1 = new GraduateStudent("Ram", 102, "A+", "Machine Learning in
Healthcare");

const libraryAccount1 = new LibraryAccount(5001, 3);

const studentWithLibrary1 = new StudentWithLibrary(student1, libraryAccount1);


// Output results

console.log(student1.getDetails());

console.log(gradStudent1.getDetails());

console.log(gradStudent1.getThesisTopic());

console.log(libraryAccount1.getLibraryInfo());

console.log(studentWithLibrary1.getFullDetails());
```

OutPut:


Output:

```
Student: Gunjan Chandnani, ID: 101, Grade: A
Graduate Student: Ram, ID: 102, Grade: A+, Thesis Topic: Machine Learning in Healthcare
Thesis Topic: Machine Learning in Healthcare
Library Account ID: 5001, Books Issued: 3
Student: Gunjan Chandnani, ID: 101, Grade: A | Library Account ID: 5001, Books Issued: 3
```

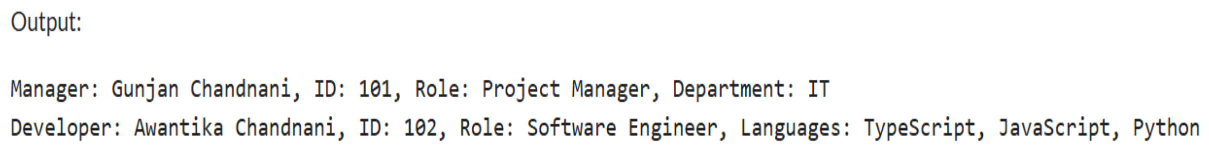
b)

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


// Manager class implementing Employee interface
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}`;
```

```
    }

}


// Developer class implementing Employee interface

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(", ")}`;

    }

}


// Creating instances

const manager1 = new Manager("Gunjan Chandnani", 101, "Project Manager", "IT");

const developer1 = new Developer("Awantika Chandnani", 102, "Software
Engineer", ["TypeScript", "JavaScript", "Python"]);


// Output results

console.log(manager1.getDetails());
```

console.log(developer1.getDetails());

OutPut:



Output:

```
Manager: Gunjan Chandnani, ID: 101, Role: Project Manager, Department: IT
Developer: Awantika Chandnani, ID: 102, Role: Software Engineer, Languages: TypeScript, JavaScript, Python
```