EXPERIMENT NO. 1B : TypeScript

Name of Student	**Komal Milind Deolekar**
Class Roll No	**14**
D.O.P.	**28-05-2025**
D.O.S.	**11-02-2025**
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.

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.

Github Link :

https://github.com/KomalDeolekar0607/Webx_Lab/tree/main/Webx_Lab_Exp_1b

Theory:

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

TypeScript provides a variety of data types categorized into **Primitive Types**, **Object Types**, and **Special Types**.

Primitive Data Types:

- string – Represents text (e.g., "Hello, World!")
- number – Represents integers and floating-point numbers (e.g., 10, 3.14)
- boolean – Represents true or false
- bigint – Represents large numbers (e.g., 9007199254740991n)
- symbol – Used to create unique identifiers

Object Data Types:

- Array<T> – Represents a list of elements (e.g., let nums: number[] = [1, 2, 3];)
- Tuple – Fixed-length array with specific types (e.g., [string, number])
- Enum – Named constant values (e.g., enum Color { Red, Green, Blue })
- Class – Object-oriented representation (e.g., class Student {})

Special Types:

- any – Can hold any type (not recommended for strict type safety)
- unknown – Similar to any, but safer as it requires type checking
- void – Used for functions that do not return anything
- null and undefined – Represent absence of values

- never – Represents a function that never returns (e.g., errors, infinite loops)

What are Type Annotations in TypeScript?

Type annotations explicitly specify the data type of a variable, parameter, or return value.

Example:

```
let username: string = "Komal";
let age: number = 22;
function add(a: number, b: number): number {
  return a + b;
}
```

This ensures type safety and prevents unintended type mismatches.

b. How do you compile TypeScript files?

TypeScript files (.ts) need to be compiled into JavaScript (.js) using the TypeScript compiler (tsc).

Steps to Compile:

- Install TypeScript (if not installed)

  ```
  npm install -g typescript
  ```

- Compile a Single File:

  ```
  tsc filename.ts
  ```

- Compile and Watch for Changes:

  ```
  tsc --watch filename.ts
  ```

- Compile All TypeScript Files in a Project:

  ```
  tsc
  ```

By default, the compiled JavaScript file is placed in the same directory. This can be configured using a tsconfig.json file.

c. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Type System	Dynamic (weakly typed)	Static (strongly typed)
Compilation	Interpreted	Compiled to JavaScript
Interfaces	Not supported	Supported
Generics	Not available	Available
Classes & Modules	Available (ES6+)	More advanced features
Error Handling	Runtime errors	Compile-time error checking
Code Readability	Less structured	More structured and scalable

TypeScript helps catch errors at **compile-time**, making development safer and more efficient.

d. Compare how Javascript and Typescript implement Inheritance.

Inheritance allows a class to acquire properties and methods from another class.

JavaScript Inheritance (ES6+)

JavaScript uses the class keyword for inheritance.

```
class Person {
  constructor(name) {
    this.name = name;
  }
```

```
  getDetails() {
    return `Name: ${this.name}`;
  }
}

class Employee extends Person {
  constructor(name, role) {
    super(name);
    this.role = role;
  }
  getDetails() {
    return `${super.getDetails()}, Role: ${this.role}`;
  }
}

const emp = new Employee("John", "Developer");
console.log(emp.getDetails());
```

TypeScript Inheritance

TypeScript provides additional type safety.

```
class Person {
  name: string;
  constructor(name: string) {
```

```
    this.name = name;
  }
  getDetails(): string {
    return `Name: ${this.name}`;
  }
}

class Employee extends Person {
  role: string;
  constructor(name: string, role: string) {
    super(name);
    this.role = role;
  }
  getDetails(): string {
    return `${super.getDetails()}, Role: ${this.role}`;
  }
}

const emp = new Employee("John", "Developer");
console.log(emp.getDetails());
```

In TypeScript, we explicitly define **data types** and ensure type safety, reducing runtime errors.

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.

Generics provide **flexibility**, **type safety**, and **code reusability**, making them a better alternative to using any.

Why Not Use any?

Using any removes type restrictions, making the function flexible but also unsafe. It can lead to unexpected behavior at runtime:

```
function processData(data: any): any {
  return data;
}

let result = processData(10);  // Allowed
result = processData("Hello"); // Also Allowed
```

Here, TypeScript does not prevent result from being reassigned to different data types, which can lead to bugs.

Why Use Generics?

Generics allow **type inference** and **ensure consistency** while keeping the function flexible:

```
function processData<T>(data: T): T {
  return data;
}

let num = processData<number>(10); // Ensures num is of type number
let str = processData<string>("Hello"); // Ensures str is of type string
```

Now, TypeScript ensures that num can only be a number and str can only be a string.

Real-World Example of Generics

Consider a function that filters an array:

```
function filterArray<T>(arr: T[], condition: (item: T) => boolean): T[] {
  return arr.filter(condition);
}

const numbers = [1, 2, 3, 4, 5];
const evenNumbers = filterArray(numbers, (num) => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

Using generics, the function can work with **any data type** while maintaining **type safety**.

Why Generics Are More Suitable for Lab Assignment 3?

In **Lab Assignment 3**, where we deal with student data (e.g., student names, marks, and IDs), generics ensure that:

- We **do not lose type information** (unlike any).
- We **maintain type consistency** (e.g., student names should always be string).
- We **avoid unnecessary type conversions**.

For example, a function handling student data can use generics:

```
class Student<T> {
  constructor(public id: number, public name: T) {}
}

const student1 = new Student<string>(1, "John");
```

const student2 = new Student<number>(2, 123); // Type error prevented

Thus, generics **increase reliability** and **prevent incorrect data assignments**

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

Feature	**Class**	**Interface**
Purpose	Defines objects and their behavior	Defines object structure witho implementation
Instantiation	Can be instantiated using new	Cannot be instantiated directly
Implementation	Includes method definitions	Only describes method signatures
Usage	Used to create reusable object models	Used to enforce a structure in other classes or objects

When to Use Classes?

- When you need a **blueprint for creating objects**.
- When objects have **state (properties)** and **behavior (methods)**.
- When you want to implement **object-oriented principles** like **inheritance**.

Example of a Class

```
class Employee {
  constructor(public name: string, public id: number) {}

  getDetails(): string {
    return `Employee ID: ${this.id}, Name: ${this.name}`;
  }
```

```
}

const emp = new Employee("Alice", 101);

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

When to Use Interfaces?

- When you need to **define a contract** that other objects must follow.
- When working with **API responses** or **third-party integrations**.
- When enforcing **consistent object structures**.

Example of an Interface

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

const emp: IEmployee = {
  name: "Bob",
  id: 102,
  getDetails: () => "Employee ID: 102, Name: Bob"
};

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

Here, the interface **enforces** that Employee objects must have name, id, and a getDetails method.

Interfaces in Real-World Use

Interfaces are commonly used in **API design**:

```
interface ApiResponse {
  status: string;
  data: object;
}

function fetchData(): ApiResponse {
  return {
    status: "success",
    data: { message: "Data retrieved" }
  };
}
```

This ensures that any API response follows a **consistent structure**.

Code :

libraryStudent.ts

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

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

```
    }
}

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

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

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

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

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

```
    }
}

// Creating instances
const student = new Student("Alice", 101, "A");
console.log(student.getDetails());
const gradStudent = new GraduateStudent("Bob", 102, "A+", "Machine Learning");
console.log(gradStudent.getDetails());
console.log(gradStudent.getThesisTopic());
const libraryAccount = new LibraryAccount(2001, 5);
console.log(libraryAccount.getLibraryInfo());

// Associating LibraryAccount with Student
const studentLibraryMapping = { student, libraryAccount };
console.log(studentLibraryMapping.student.getDetails());
console.log(studentLibraryMapping.libraryAccount.getLibraryInfo());
```

Output :

```
D:\Users\Komal\OneDrive\Desktop\sem 6\webx_lab\typescript_lab_1b>node libraryStudent.js
Student Name: Alice, ID: 101, Grade: A
Graduate Student Name: Bob, ID: 102, Grade: A+, Thesis: Machine Learning
Thesis Topic: Machine Learning
Library Account ID: 2001, Books Issued: 5
Student Name: Alice, ID: 101, Grade: A
Library Account ID: 2001, Books Issued: 5
```

Code :

employeeManagement.ts

```
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 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: number, public role: string, public programmingLanguages: string[]) {}

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

```
    }
}
// Creating instances
const manager = new Manager("Charlie", 201, "Manager", "Software Development");
console.log(manager.getDetails());
const developer = new Developer("David", 202, "Developer", ["TypeScript", "JavaScript", "Python"]);
console.log(developer.getDetails());
```

Output :

```
D:\Users\Komal\OneDrive\Desktop\sem 6\webx_lab\typescript_lab_1b>node employeeManagement.js
Manager Name: Charlie, ID: 201, Role: Manager, Department: Software Development
Developer Name: David, ID: 202, Role: Developer, Languages: TypeScript, JavaScript, Python
```

Conclusion :

This study of basic constructs in TypeScript provides a solid foundation for understanding the language's core features, such as variables, data types, operators, functions, and control structures. By utilizing TypeScript's strongly-typed system, we can write code that is more reliable, maintainable, and less error-prone compared to JavaScript.The exploration of TypeScript's static typing, interfaces, and type inference ensures better code quality and debugging capabilities. This knowledge serves as a stepping stone for developing scalable and efficient applications, making TypeScript a preferred choice for modern web development.