

## Experiment – 1 b: TypeScript

Name of Student	Prathamesh Palve
Class Roll No	D15A_31
D.O.P.	<u>30/01/2025</u>
D.O.S.	<u>06/02/2025</u>
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, floating-point numbers, etc.).
- `string`: For text.
- `boolean`: `true` or `false`.
- `null`: Represents the intentional absence of a value.
- `undefined`: Represents a variable that has been declared but not assigned a value.
- `symbol`: Unique and immutable values (often used as keys for object properties).
- `bigint`: For arbitrarily large integers.

- **Structural Types:**

- `object`: Represents non-primitive values, such as 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 sparingly!).
- `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. It forces you to perform type checking before you can use the value.

- **Enum:** A way to define a set of named constants.

## 2. Type Annotations:

Type annotations explicitly specify the type of a variable, function parameter, or function return value. They are written after the variable or parameter name, followed by a colon and the type.

TypeScript

- a. `let age: number = 30;`
- b. `let name: string =`  
`"Alice";` c.
- d. `function greet(person: string): string {`
- e. `return "Hello, " + person;`
- f. `}`

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`) need to be compiled into JavaScript files (`.js`) that browsers or Node.js can understand. You use the TypeScript compiler (`tsc`) for this.

- **Command Line:** `tsc filename.ts` compiles `filename.ts`. `tsc` without a filename compiles all `.ts` files in the current directory (or as configured in a `tsconfig.json` file).
- **tsconfig.json:** A configuration file that lets you customize the compilation process (output directory, target JavaScript version, etc.). It's highly recommended to use a `tsconfig.json` file.

## 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). This is the biggest difference.
- **Compilation:** TypeScript needs to be compiled to JavaScript before it can be run. JavaScript can be run directly in browsers or Node.js.

- **Error Detection:** TypeScript catches type errors early during development, leading to more robust code. JavaScript errors related to types are only discovered at runtime.
- **Code Maintainability:** TypeScript's type system makes it easier to understand and maintain large codebases.
- **Features:** TypeScript often supports newer JavaScript features earlier than some browsers. It can also transpile down to older JavaScript versions for compatibility.

## 5. Inheritance in JavaScript vs. TypeScript:

- **JavaScript (Prototypal Inheritance):** JavaScript uses prototypes for inheritance. Objects inherit properties and methods from other objects (their prototypes). This is a dynamic mechanism. It can be more difficult to reason about in complex scenarios.
- **TypeScript (Class-based Inheritance):** TypeScript (and modern JavaScript) supports class-based inheritance, which is a more familiar pattern for developers coming from languages like Java or C++. TypeScript classes are compiled down to JavaScript prototype-based code under the hood. It provides a more structured and predictable approach to inheritance.

## 6. Generics:

Generics allow you to write reusable components that can work with a variety of types without sacrificing type safety. Instead of using `any`, which bypasses type checking, you use type parameters to specify the type at the time the component is used.

- **Flexibility:** Generics make your code adaptable to different data types. You write the code once, and it works with numbers, strings, custom objects, etc.
- **Type Safety:** Generics preserve type information. The compiler knows the specific type you're working with, so it can perform type checks and prevent errors.

**Why Generics are better than `any` (and why they are suitable in the scenario you mentioned):**

- **`any` defeats the purpose of TypeScript.** It essentially turns off type checking, so you lose all the benefits of static typing. Errors that would be caught by the compiler are now only found at runtime.
- **Generics maintain type safety.** They allow you to work with different types *while* preserving type information. If you use `any`, you lose track of the actual type, making your code more error-prone.

In your lab assignment, using generics is likely more suitable because you want to handle input of various types (e.g., numbers, strings, or custom objects). Using `any` would make the code less type-safe and harder to maintain. Generics let you write a single function or component that can handle different input types while still ensuring type safety.

## 7. Classes vs. Interfaces:

- **Classes:** Define the blueprint for creating objects (instances). They can contain properties (data) and methods (functions). Classes can be instantiated using the `new` keyword. Classes can implement interfaces and also extend other classes.
- **Interfaces:** Define a *contract* or *shape* for an object. They specify the properties and methods that an object *must* have. Interfaces cannot be instantiated directly. They are used to enforce type compatibility. Classes *implement* interfaces.

### Where Interfaces are Used:

- **Defining the shape of objects:** Interfaces are used to ensure that objects have the required properties and methods.
- **Enforcing contracts:** They specify the requirements that classes must meet.
- **Improving code reusability:** Interfaces can be used to create more generic functions and components.
- **Working with different types:** Interfaces can be used to define the shape of complex data structures.

#### 4. Output:

a.

```
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
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
class StudentWithLibrary {
    constructor(public student: Student, public libraryAccount: LibraryAccount)
{}

    getFullDetails(): string {
```

```

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

// Creating instances
const student1 = new Student("Sanket", 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:-

```

prath@prathmzzPC MINGW64 ~/Desktop/Hackathons/temp/temp4
$ node temp.js
Student: Prathamesh, 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: Prathamesh, 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("Sanket",
101, "Project Manager", "IT");
const developer1 = new
Developer("Ram", 102, "Software
Engineer", ["TypeScript",
"JavaScript", "Python"]);

// Output results
console.log(manager1.getDetails());
console.log(developer1.getDetails());

```

## OUTPUT:-

```

prath@prathmzzPC MINGW64 ~/Desktop/Hackathons/temp/temp4
$ tsc temp.ts

prath@prathmzzPC MINGW64 ~/Desktop/Hackathons/temp/temp4
$ node temp.js
Manager: Prathamesh, ID: 101, Role: Project Manager, Department: IT
Developer: Ram, ID: 102, Role: Software Engineer, Languages: TypeScript, JavaScript, Python

```