

WEBX EXP1B

Aim: To study Basic constructs in TypeScript.

Theory:

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

TypeScript provides several built-in data types, including primitive types like string, number, boolean, null, undefined, symbol, and bigint. It also includes object types such as Array, Tuple, Enum, Class, and Interface. Additionally, TypeScript has special types like any, unknown, void, and never.

Type annotations in TypeScript allow developers to explicitly define the type of a variable, function parameter, or return value. For example, `let age: number = 25;` ensures that age can only hold numeric values, reducing runtime errors and improving code clarity.

2.How do you compile TypeScript files?

To compile a TypeScript file, use the TypeScript compiler (tsc) by running the command `tsc filename.ts` in the terminal. This command translates the TypeScript code into JavaScript, generating a `filename.js` file that can be executed in a browser or Node.js environment.

3.What is the difference between JavaScript and TypeScript?

JavaScript is a dynamically typed language, whereas TypeScript is statically typed, which helps catch errors at compile time. JavaScript does not support interfaces or strong type checking, while TypeScript allows defining interfaces for better structure. Additionally, JavaScript is interpreted, whereas TypeScript must be compiled into JavaScript before execution. TypeScript also provides modern ES features along with additional enhancements like generics and type annotations, making it more robust for large-scale applications.

4.Compare how Javascript and Typescript implement Inheritance.

JavaScript uses prototype-based inheritance, where objects inherit properties and methods from other objects through the prototype chain. TypeScript, on the other hand, supports class-based inheritance using the `class` and `extends` keywords, making it more structured and similar to object-oriented programming in languages like Java or C#. This makes TypeScript more readable and maintainable compared to JavaScript's traditional prototype-based approach.

5.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 type safety by ensuring that functions and data structures work with a specific type rather than accepting any type, as the any keyword does. This prevents runtime errors and improves code reusability while maintaining strict type rules. For example, a function using generics like `function identity<T>(value: T): T { return value; }` ensures that it always returns the same type that it receives as input. In Lab Assignment 3, using generics is more suitable than using any because it ensures that the input data type remains consistent while still allowing flexibility.

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

Classes in TypeScript define both the structure and implementation of an object, meaning they can have methods and properties, and they support instantiation. Interfaces, however, only define the structure without providing any implementation and cannot be instantiated. Interfaces are primarily used to define contracts for objects, ensuring that they have specific properties and methods. For example, an interface `Person { name: string; age: number; }` can be implemented by multiple classes to ensure they follow the same structure.

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.

Code:

```
// 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 extending Student
```

```
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}`;  
    }  
  
    override getDetails(): string {  
        return `${super.getDetails()}, 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: Associating LibraryAccount with Student
```

```
class StudentWithLibrary {  
    constructor(public student: Student, public libraryAccount: LibraryAccount) {}  
  
    getFullInfo(): string {  
        return `${this.student.getDetails()}\n${this.libraryAccount.getLibraryInfo()}`;  
    }  
}
```

```
// Creating instances
```

```
const student1 = new Student("Sneha", 101, "A");  
const gradStudent1 = new GraduateStudent("Bob", 102, "A+", "AI Research");  
const libraryAcc1 = new LibraryAccount(5001, 3);
```

```
// Associating Student with LibraryAccount
```

```
const studentWithLibrary = new StudentWithLibrary(student1, libraryAcc1);
```

```
// Calling methods and observing behavior

console.log(student1.getDetails()); // Student details

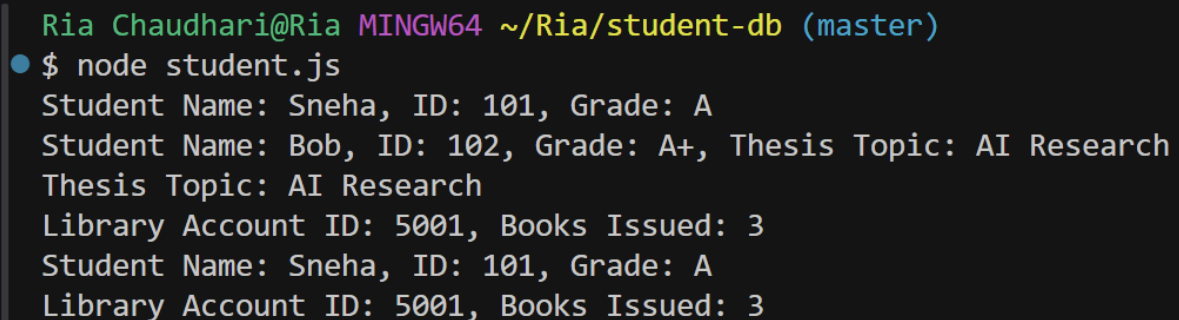
console.log(gradStudent1.getDetails()); // GraduateStudent details with thesis

console.log(gradStudent1.getThesisTopic()); // Specific thesis topic

console.log(libraryAcc1.getLibraryInfo()); // Library account details

console.log(studentWithLibrary.getFullInfo()); // Composition example
```

Output:



```
Ria Chaudhari@Ria MINGW64 ~/Ria/student-db (master)
● $ node student.js
Student Name: Sneha, ID: 101, Grade: A
Student Name: Bob, ID: 102, Grade: A+, Thesis Topic: AI Research
Thesis Topic: AI Research
Library Account ID: 5001, Books Issued: 3
Student Name: Sneha, ID: 101, Grade: A
Library Account ID: 5001, Books Issued: 3
```

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.

Code:

```
// Employee interface

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},
        Programming Languages: ${this.programmingLanguages.join(", ")}`;

    }
}

// Creating instances

```

```
const manager1 = new Manager("John", 101, "Manager", "Sales");

const developer1 = new Developer("Ria", 102, "Developer", ["TypeScript", "JavaScript",
"Python"]);

// Displaying details

console.log(manager1.getDetails());

console.log(developer1.getDetails());
```

Output:

```
Ria Chaudhari@Ria MINGW64 ~/Ria/student-db (master)
● $ node student.js
Manager Name: John, ID: 101, Role: Manager, Department: Sales
Developer Name: Ria, ID: 102, Role: Developer, Programming Languages: TypeScript, JavaScript, Python
```