Project: TypeScript-Based Application

1. Introduction

- TypeScript is a superset of JavaScript and its advanced version.
- All features of JS, extra features and functionality.
- This project uses basic TypeScript features like **interfaces, classes, and types** to organise the code properly
- The application is easy to understand, even for beginners, and follows a simple structure. It is useful for learning how real-world applications are built using TypeScript and JavaScript.

2. Objectives

- Apply the TypeScript fundamentals and understand
- Use interfaces, types, generics, and enums effectively
- Implement ES6+ features
- Build a type-safe and maintainable application
- Learn project structuring and documentation

3. Technology Stack

- **Language:** TypeScript
- **Runtime:** Node.js
- **Package Manager:** npm
- **Compiler:** TypeScript (tsc)
- **Tools:** VS Code, Git

4. Project Structure

- project-root
- src
- models
- User.ts
- services
- UserService.ts
- utils
- Logger.ts
- app.ts
- dist
- tsconfig.json
- package.json
- README.md

5. Functional Requirements

- Create and manage users
- Validate input data using TypeScript types
- Perform CRUD operations (Create, Read, Update, Delete)
- Display structured output in the console

6. <u>Non-Functional Requirements</u>

- Code must be **type-safe**
- Application should be **scalable and maintainable**
- Follow **clean coding standards**
- Easy to understand for beginners

7. <u>TypeScript Concepts Used</u>

7.1 <u>Interfaces</u>

- An interface is a blueprint (structure) that describes what properties and methods an object should have.
  Example:

```
Export interface Task{
Id: number;
Title: string;
Completed: Boolean;
}
```

- Why do we use interface
  i. The code is safe
  ii. The mistake is found early.
  iii. Readable & clean code

7.2 <u>Classes</u>

- A class is a blueprint or template based on which objects are created
  Example:-
  Class =Car
  Object=BMW, Audi, Toyota, etc

7.3 <u>Generics</u>

- Generics help reuse code with different types while keeping type safety.
- Solution:-

```
function identity<T>(value: T): T {

  return value;
}
```

- **<u>Using Generic Function</u>**
  ```
  identity<number>(100);
  identity<string>("Raman");
  identity<boolean>(true);
  ```
  - Generics are safe, flexible, and allow the same code to be reused.

7.4 <u>Modules</u>

- A module is a file in which we write a specific part of code, and we can reuse it in other files.

7.5 <u>Why use Modules</u>

- Code clean and Organized
- Easy maintenance
- Reusability
- Team work easy

7.6 <u>Type of Modules</u>

i. JavaScript ES Modules
ii. TypeScript Module

7.7 <u>Export</u>

```
// math.ts( comment)

    export function add(a: number, b: number): number {

      return a + b;

    }

        export const PI = 3.14;
```

NOTE:- This code is use in another file

7.8 <u>Import</u>

```
// app.ts( Comment)

import { add, PI } from "./math";

console.log(add(2, 3));
```

```
console.log(PI);
```

7.8 Module File Structure

- src
- models
- Student.ts
- services
- StudentService.ts
- app.ts

8. Application Flow

1. Application starts from app.ts
2. User data is created using interfaces
3. Business logic handled by service classes
4. Output logged using utility functions

9. Sample Use Case

- Add a new user
- Fetch user details
- Update user information
- Delete user record

10. Compilation & Execution

1. Install dependencies
2. Compile TypeScript to JavaScript
3. Run the compiled output

11. Expected Output

- Clean and structured console output
- Error-free compilation
- Type-safe execution

12. Future Enhancements

- Add database integration
- Create REST API using Express
- Add frontend using React + TypeScript
- Implement unit testing

13. Conclusion

We learned the core concepts of Advanced JavaScript and TypeScript. ES6+ features, functional programming, and design patterns helped us write clean and efficient code. TypeScript fundamentals, interfaces, and generics enabled us to create type-safe, flexible, and reusable code.