Module(01-03):

Explore basic of TypeScript:

What is TypeScript?

Typescript is an Object Oriented Programming Language that is built on top of JS with Extra Features.

(primitive datatype): shortcut: ss bb u nn

1.String

2.Symbol

3. BingIn

4. Boolean

5. Undefined

6. Number

7. Null

Non-primitive datatype:

1. Arry
2. Object
3. Function

Js Types in TS

1. Number
2. String
3. Boolean
4. Null
5. Undefined
6. Object
7. Symbol

Ts Own Types

1. Interface
2. Void
3. Array
4. Tuple
5. Enum
6. Union
7. Intersection

Drawbacks of using Typescript:

1. Type Complexities
2. Limited Library Support
3. Over Engineering
4. Migration Challenges

Download NodeJs. (using command line)

1. Open PowerShell
2. winget install Schniz.fnm
3. fnm env --use-on-cd | Out-String | Invoke-Expression
4. fnm use --install-if-missing 22

Bonus

1. fnm list (it’s give you existing version of nodejs)
2. fnm install 20.19.1 (If you need nodeJs v20.19.1)
3. fnm use 23.11.0 (if you need work with nodeJs version v23.11.0)

Install typeScript(cmd):

1. npm install typescript
2. tsc -v (it’s show the current version of typeScript)

Ts configuration:

tsc –init

tsconfig.json

Search: rootDir and change directory where your source file is stayed

Search: outDir and change directory where your compiled js file are stayed (e.g. ./dist)

Basic Data Types

Primitive

1. number
2. string
3. boolean
4. null
5. undefined
6. symbol

Non primitive type

1. Arry
2. Tuple
3. Object

Module-2:

Type Assertion:

parseFloat():

The parseFloat() function parses a string argument and returns a floating point number.

In TypeScript, type assertions allow developers to override the compiler’s inferred type, informing it of the specific type of a value.

```bash

let value: any = "This is a string";

let lengthOfString: number = (value as string).length;

console.log(lengthOfString); //16

```

To run ts file you need this command: ts-node-dev --respawn --transpile-only (filename)

Type assertion with function return value:

```bash

function getValue(): any {

return 'Hello, TypeScript!';

}

let strLength: number = (getValue() as string).length;

console.log(strLength); 18

```

Type assertion with DOM Element:

```bash

let element = document.querySelector('input[type="text"]');

let inputElement = element as HTMLInputElement;

console.log(inputElement.value); /// [Value of the input field]

```

Type assertion with Union Types:

```bash

type Pet = {

name: string;

walk: () => void;

};

type Fish = {

name: string;

swim: () => void;

};

let myPet: Pet | Fish = { name: 'Goldie', swim: () => console.log('Swimming') };

(myPet as Fish).swim(); //Swimming

```

Interface:

In TypeScript, an interface is used to define the **shape of an object**—it specifies what properties and methods an object should have. Think of it like a contract that your object must follow.

✅ Basic Example:

```ts

interface User {

name: string;

age: number;

}

const user: User = {

name: "Alice",

age: 25,

};

```

✅ Method Signatures

Interfaces can include method definitions:

```ts

interface User {

name: string;

greet(): void; /// greet function doesn’t return anything

}

const user: User = {

name: "Bob",

greet() {

console.log(`Hello, my name is ${this.name}`);

},

};

```

✅ Extending Interfaces

Interfaces can extend other interfaces:

```ts

interface Person {

name: string;

}

interface Employee extends Person {

employeeId: number;

}

const emp: Employee = {

name: "John",

employeeId: 123,

};

```

Type:

In TypeScript, type is a keyword used to create **type aliases**, which means giving a custom name to a type. It’s very flexible and powerful—more so than interface in some cases.

### **✅ Basic Type Alias**

```ts

type User = {

name: string;

age: number;

};

const user: User = {

name: "Alice",

age: 30,

};

```

### **✅ Union Types**

type lets you define a variable that can be one of several types:

```ts

type Status = "loading" | "success" | "error";

let currentStatus: Status = "loading"; // ✅

currentStatus = "success"; // ✅

// currentStatus = "done"; ❌ Error: "done" is not assignable to type 'Status'

```

### **✅ Intersection Types**

You can combine types using & (intersection):

```ts

type Person = {

name: string;

};

type Employee = {

employeeId: number;

};

type Worker = Person & Employee;

const worker: Worker = {

name: "John",

employeeId: 123,

};

```

✅ Function Types

You can define function types using type:

```ts

type GreetFunction = (name: string) => string;

const greet: GreetFunction = (name) => `Hello, ${name}`;

```

### **🔹 Use interface when:**

* You're defining **object shapes**, especially if they need to be extended later.
* You're working with **class-based code**.
* You want **declaration merging** (e.g., in library typings).
* Not useful for primitive datatype

### **🔹 Use type when:**

* You need to define **unions, intersections, tuples, or primitives**.
* You want to compose multiple types using & or |.
* You need **complex type logic**, like conditional or mapped types.

{

//interface

type User1 = {

name: string,

age: number

}

type rollNumber = number

interface User2 {

name: string,

age: number

}

const user1: User1 = {

name: "Jadu",

age: 100

}

const user2: User2 = {

name: "Jadu",

age: 100

}

type UserWithRole1 = User1 & { role: string }

const userWithRole1: UserWithRole1 = {

name: "Jadu",

age: 100,

role: "123"

}

interface UserWithRole2 extends User2 {

role: string

}

//type extends to interface

// interface UserWithRole2 extends User1 {

// role: string

// }

const userWithRole2: UserWithRole2 = {

name: "Jadu",

age: 100,

role: "123"

}

//js --> object, arry -> object function -> object

type roll1 = number[];

interface roll2 {

[index: number]: number

}

const rollNumber1: roll1 = [1, 2, 3]

const rollNumber2: roll2 = [1, 2, 3]

type Add1 = (num1: number, num2: number) => number

interface Add2 {

(num1: number, num2: number): number

}

const add1: Add1 = (num1, num2) => num1 + num2

const add2: Add2 = (num1, num2) => num1 + num2

}

Generic type:

Generics in TypeScript allow you to **write reusable and flexible code** by making your functions, interfaces, or types work with **any data type**, while still maintaining **type safety**.

## **🧠 Why Use Generics?**

Without generics, you'd have to use any to accept various types, which loses type safety:

```ts

function identity(value: any): any {

return value;

}

``

✅ With **generics**, TypeScript remembers the type:

```ts

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

return value;

}

```

Now, if you call:

```ts

identity<string>("hello"); // Type is string

identity<number>(42); // Type is number

``

🔧 Generic Function Example:

```ts

function getFirstElement<T>(arr: T[]): T {

return arr[0];

}

const num = getFirstElement<number>([1, 2, 3]); // number

const str = getFirstElement(["a", "b"]); // string (inferred)

```

## **📦 Generic Interface or Type**

```ts

interface ApiResponse<T> {

data: T;

success: boolean;

}

const userResponse: ApiResponse<{ name: string }> = {

data: { name: "Alice" },

success: true,};```

🔁 Generic with Constraints (extends):

You can **restrict** what type T can be:

```ts

function printLength<T extends { length: number }>(item: T): void {

console.log(item.length);

}

printLength("hello"); // ✅

printLength([1, 2, 3]); // ✅

printLength(123); // ❌ number doesn't have length

```

Here’s what’s happening:

* T extends { length: number }:  
   This means **T can be any type, as long as it has a length property that is a number**.
* So, when you call printLength("hello"):  
  + "hello" is a string → strings have a .length → ✅ Allowed.
* And for printLength([1, 2, 3]):  
  + Arrays have .length → ✅ Allowed.
* But for printLength(123):  
  number doesn't have .length → ❌ Error.

Constraint // Asynchronous typeScript

Module:3

What is OOP?

A programming paradigm that organizes and models software.

What is Paradigm?

The style used to write and organize code.

Procedural programming:

Functional programming:

Declarative programming;

Object Oriented programming:

Event Driven programming:

Building blocks of OOP:

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation

Class and Object:

Interview special

1. What are some differences between interfaces and types in TypeScript?
2. What is the use of the keyof keyword in TypeScript? Provide an example.
3. Explain the difference between any, unknown, and never types in TypeScript.
4. What is the use of enums in TypeScript? Provide an example of a numeric and string enum.
5. What is type inference in TypeScript? Why is it helpful?
6. How does TypeScript help in improving code quality and project maintainability?
7. Provide an example of using union and intersection types in TypeScript.