Assignment 5.2- Introduction to Type Script

D1.

(1) Introduction to Type Script

→ Brief Overview of Type Script

Type Script is a statically typed superset of Javascript that compiles to plain Java Script. Developed by Microsoft, it adds static types, classes and interfaces to the language, enhancing the development experience and improving code quality.

→ Advantages of Type Script Over Java Script

- Static Typing : Catches type - related errors during development, reducing runtime errors.
- Enhanced IDE Support : Offers better code navigation, auto completion, and refactoring tools.
- Improved Maintainability : Type annotations and interfaces make the code more self - documenting.
- Advanced features : Includes features such as generics, decorators, and type inferences not available in plain Javascript.

(2) Getting Started

→ Installation Instructions for Type Script Compiler (tsc)

To install the Type Script Compiler, run the following command:

code ->

```
npm install -g typescript
```

→ Setting Up a new TypeScript Project

(1) Initialize a new Node.js project:

code ->

```
npm init -y
```

(2) Install TypeScript:

code ->

```
npm install typescript --save-dev
```

(3) Create a tsconfig.json file:

code ->

```
tsc --init
```

→ Integrating TypeScript with Existing JavaScript Projects

(1.) Add TypeScript to the projects:

code ->

```
npm install typescript --save-dev
```

(2) Rename JavaScript files (.js) to TypeScript file (.ts)

(3) Run the TypeScript compiler:

code ->

```
tsc
```

(III) Basic Syntax and Types

(1) Overview of TypeScript Syntax Compared to JavaScript.

Type Script syntax is similar to JavaScript but with additional type annotations. Here's a simple comparision:

JavaScript:

code ->

```
let message = "Hello, World" ;
```

Type script :

code ->

```
let message : string = "Hello, World!";
```

-> Introduction to Basic Data types

(1.) Number

code ->

```
let num : number = 42;
```

(2) String :

code ->

```
let str : string = "Hello, Type Script" ;
```

(3.) Boolean :

code ->

```
let isOpen : boolean = true;
```

(4.) NUll and Undefined :

code ->

```
let n : null = null;
let u : undefined = undefined;
```

-> Type Annotations and type Inference

Type annotations explicitly declare variable types.

code : ->

```
let age : number = 25;
```

Type inference allows TypeScript to deduce types automatically :

code->

```
let name = "John"; // inferred as string
```

-> Static Typing

- Explanation of Static typing and its benefits

Static typing involves declaring variable types at compile time, catching errors early in the development process, improving code reliability and readability.

- Declaring Variables Types Using Type Annotations

Code ->

```
let isCompleted : boolean = false;
```

- Type Inference

TypeScript infers types based on a assigned values and contexts

code ->

```
let count = 10; // inferred as number
```

-> Interfaces

- Definition and Usage of Interfaces

Interfaces define the shape of objects, providing a way to describe object structures:

Code →
```
interface Person {
    name : string ;
    age : number ;
}
```

- Creating Interfaces for Object Shapes and Contracts

code →
```
Const john : Person = {
    name : " John Doe",
    age : 30
};
```

- Optional Properties and Read - Only Properties in Interfaces

Code →
```
interface Car {
    brand : String ;
    model ? : String ;   // optional property
    readonly year : number ;  // read - only property
}
```

→ classes

- object - Oriented Programming Concepts in Typescript

TypeScript supports OOP principles such as encapsulation, inheritance, and polymorphism.

- Defining classes with Properties and Methods code ->

```
class Animal {
    name : string;

    Constructor (name : string) {
        this. name = name;
    }

    Speak () {
        Console. log ('${this.name} makes a noise.');
    }
}
```

- Constructors and Access Modifiers

code ->

```
class Dog extends Animal {
    private breed : String;

    Constructor (name : string, breed : String) {
        Super (name);
        this. breed = breed;
    }

    public getBreed () {
        return this. breed; }

    protceted bark () {
        Console. log ('${this. name} barks.'); }

}
```

- Inheritance and Method Overriding

Code →

```
Class Cat extends Animal {
    speak () {
      console.log (`${this.name} meows.`);
    }
}
```

→ Generics

- Introduction to Generics in Typescript

Generics provide a way to create reusable components the works with any data type.

Code →

```
function identity <T> (arg : T): T {
        return arg;
}
```

- Creating Reusable Components with Generic Types.

code →

```
class GenericNumber <T> {
      value : T;
    Constructor (value : T) {
          this.value = value;
    }
}
```

- Using Generic Constraints

Code →

```
function loggingIdentity <T extends { length : number }
        > (arg : T) : void {
  console.log (arg.length); }
```

→ Advanced Type Script Concepts

- Union types and Intersection types →

Union types : Allow a variable to hold multiple types.

code→

```
let id: number | string ;
    id = 10;
    id = "42";
```

Intersection types: Combine multiple types into one:

code→

```
interface A { a: number ; }
interface B { b: String ; }
type AB = A & B;
```

- Type Aliases and Type Assertions

Type Aliases: Create a new name for a type:

code→

```
type Point = { x: number ; y: number ; };
```

Type Assertions: Override Type Script's inferred type:

code→

```
let somevalue : any = "Hello World";
let strLength : number = (someValue as String).length;
```

- Type Guards for Working with Unions

```
function isString (x: any) : x is string {
  return typeof x === "String";
}
```

- Conditional Types and Mapped Types

Conditional Types:

code→

```
type NonNullable <T> = T extends null | undefined ?
                                          never : T;
```

Mapped Types:

code→

```
type Readonly <T> = {
    readonly [P in Keyof T]: T [P];
};
```