**TYPESCRIPT**

1. **Defination :** TypeScript is an open-source programming language developed and maintained by Microsoft.

It is a superset of JavaScript, which means that any valid JavaScript code is also valid TypeScript code.

TypeScript adds optional static typing to JavaScript, allowing developers to catch errors and bugs during development rather than at runtime.

Note : 1. Optional Static Typing : In JavaScript, variables are dynamically typed, meaning their types are determined at runtime. TypeScript allows developers to specify types for variables, function parameters, return types, and more. These type annotations are optional, which means you can still write code without specifying types if you prefer the dynamic typing style of JavaScript.

2. By adding static typing, TypeScript can catch type-related errors and bugs during the development phase. For example, if you try to assign a string value to a variable that is supposed to hold a number, TypeScript will throw a compile-time error, alerting you to the mistake before running the code.

1. **Installation :** npm install -g typescript
2. **TSC :** TSC in TypeScript refers to the TypeScript Compiler, which is a command-line tool used to compile TypeScript code (.ts files) into JavaScript code (.js files).
3. **File Creation :**

* Create a file like this sample.ts

Add some code there like the below , or when you run the command : tsc ‘.\filename.ts’

It creates an js file which is then link with the html file.

|  |  |  |
| --- | --- | --- |
| **Sample.ts** | **Sample.js** | **Sample.html** |
| const a = "Piyush";  console.log(a);  const b =23;  console.log(b); | var a = "Piyush";  console.log(a);  var b = 23;  console.log(b); | <!DOCTYPE html>  <html lang="en">  <head>      <title>Document</title>  </head>  <body>      <script src="./01Basic.js"></script>  </body>  </html> |

* When every you do changes in the sample.ts you have to compile it into the sample.js (javascript code) by running the same command as above.
* But compile sample,ts again and again manually is not greate so you can use –watch as follows:
* **--watch :** In TypeScript (TS), the --watch option is used with the TypeScript Compiler (tsc) command to enable automatic recompilation of TypeScript files when changes are detected. This feature is particularly useful during development as it helps developers quickly see the effects of code changes without manually triggering the compilation process each time.

**EXAMPLE :** tsc ‘.\sample.ts’ -w **OR** tsc -w

1. **tsc –init :** This command generates a tsconfig.json file in your current directory. The tsconfig.json file contains compiler options and settings for the TypeScript compiler (tsc). This file allows you to specify settings such as target ECMAScript version, module resolution, output directory, and more.
2. **Datatypes in ts :** let variableName : datatypeName = value;

|  |
| --- |
| Sample.ts  let a1: any = 25;  a1 = "Piyush"; // This will not cause a TypeScript compilation error  console.log(a1);  // Typescript Error : Type 'string' is not assignable to type 'number'. |
| **Note :** TypeScript is a statically typed language, which means that variables have a specific type that cannot be changed once it is assigned.  you initially assign a value of 25 to the variable a1, implicitly assigning it the type number. However, when you later attempt to assign the string "Piyush" to a1, TypeScript detects a type mismatch error because you are trying to assign a string to a variable that was previously declared as a number. |
| **// Explicity defining databtypes**  let c1: string = "shreyash"  let d1:string = 45 >> ERROR : Type 'number' is not assignable to type 'String'  **1. Boolean**  let isDone:boolean = false  **2. String**  let fname:string = "Piyush";  let lname:string;  lname = "Thaware"  **3. Number**  let decimal: number = 6;  let hex: number = 0xf00d;  let binary: number = 0b1010;  let octal: number = 0o744;  **4. Array**  let marks : number[] = [1, 2, 3, 4, 5];  **5. Tuple :** Represents an array with a fixed number of elements where each element may have a different type.  let submarks : [string,number] = ["Shreyash",23];  **6. Any: Represents a dynamic type that can hold values of any data type. Avoid using any whenever possible as it bypasses type checking.**  let notSure : any = 4;  notSure = "may be a string"  **7. Null and Undefined: Represents null and undefined values respectively. By default, null and undefined are subtypes of all other types.**  let n: null = null;  let u: undefined = undefined;  **8. Union Type :** In TypeScript, a union type allows a variable to hold values of multiple specified types.  let myVar: number | string;  myVar = 10;  console.log(myVar);  myVar = "Piyush"  console.log(myVar); |
| **Alternate Method To Define Datatypes Explicitly**  let num = <number>343; |

1. **Function Return Types :**

**In the below code you are manually defining the the return type :**

|  |  |
| --- | --- |
| **Function :**  const func = (n:any,m:string)=>{  *// Parameter n implicitly has an type.*      console.log(n,m);  }  **OUTPUT :** const func: (n: any, m: string) => void | **Function :**  const func = (n:any,m:string)=>{  // Parameter n implicitly has an type.      console.log(n,m);      return n\*m  **OR**  return String(n\*m)  }  **OUTPUT :** const func: (n: any, m: string) => number  **OR**  const func: (n: any, m: string) => string |

**Now we want to set the dynamic return type :**

|  |
| --- |
| const func1 = (n:number,m:number**):number** =>{      return n\*m;  }  **OUT PUT :** const func1: (n: number, m: number) => number |

1. **Type Keyword :** In TypeScript, the type keyword is used to create type aliases, which allow developers to create custom names for existing types or complex type definitions. Type aliases improve code readability, reduce duplication, and make complex types more manageable.

|  |
| --- |
| **1. Basic Usage**  type ID = number;  let userId: ID = 1001; |
| **2. Union Types**:  type Username = string | number;  let fname:Username = "Piyush"  let age:Username = 23  console.log(fname,age);  let lname : Username = true 🡨 ERROR |
| **3. Object Type**  type UserDetails = {      id:number,      fullname : string,      email:string      isAdmin : boolean  }  let user1:UserDetails = {      id:1,      fullname:"Piyush Thaware",      email:"Piyush@gmail.com",      isAdmin:true  }  console.log(user1); |
| **4. Function Type**  type userJob = (n: number, m: number) => number;  const func: userJob = (n, m) => {  *return* n \* m;  } |

1. **Array :**

|  |
| --- |
| 1. **Array Declaration** |
| **a. Using Type Annotations**  let numbers: number[] = [1, 2, 3, 4, 5];  let names: string[] = ["Alice", "Bob", "Charlie"];  console.log(numbers,names);  **b. Using Generic Array Type**  let scores: Array<number> = [90, 85, 88, 92];  let mixedArray: Array<number | string> = [10, "hello", 20, "world"];  console.log(scores,mixedArray); |
| 1. **Accessing Array Elements** |
| console.log(number[0])  console.log(names[2]) |

|  |
| --- |
| 1. **Array Type Assertions :** If TypeScript cannot infer the array type correctly, you can use type assertions. Means you can add any kind of datatypes item in an array. |
| let values: any[] = [1, "hello", true];  let numbers: number[] = values as number[]; |

|  |
| --- |
| 1. **Readonly Arrays :** You can declare arrays as readonly to prevent modification after initialization. |
| let readOnlyNumbers: readonly number[] = [1, 2, 3]; |

|  |
| --- |
| 1. **Iteration Over An Array** |
| **a. Using For loop**  for (let i = 0; i < numbers.length; i++) {      console.log(numbers[i]);  }  **b. using for each loop**  let names: string[] = ["Alice", "Bob", "Charlie"];  names.forEach(name => {      console.log(name);  }); |

|  |
| --- |
| 1. **Array Tupple** |
| const arr:[number,string,number] = [2,"Piyush",55]  console.log(arr);  // adding extract item lead to error [2,”Piyush”,55,x,y,z] as size of array is now only 3 |

1. **Interface :**   
   In TypeScript, interfaces are a powerful tool for defining the shape of objects. They allow you to specify the structure of an object by defining properties and their types. Interfaces are particularly useful for ensuring type safety and providing clear contracts for objects that are expected to have specific properties and methods.

|  |
| --- |
| **1. Defining the interfaces : Objects in TypeScript can implement interfaces to ensure that they have the required properties and methods specified by the interface.**  interface ExamForm {      fname: string,      lname: string,      email: string,      age: number,      contact: number,      feesPaid: boolean  }  **3. Extending Interfaces: Interfaces can extend other interfaces to inherit their properties and methods while adding new ones.**  interface newForm extends ExamForm{      subject?:string,      totalMarks : number  }  **2. Implementing Interfaces**  const myObj: newForm = {      fname: "Shreyash",      lname: "Thaware",      email: "Shreyash@gmail.com",      age: 23,      contact: 99999999,      feesPaid: false,      totalMarks:3000  }  console.log(myObj);  **OUTPUT :**  {  "fname": "Shreyash",  "lname": "Thaware",  "email": "Shreyash@gmail.com",  "age": 23,  "contact": 99999999,  "feesPaid": false,  "totalMarks": 3000  } |

|  |
| --- |
| **Implementing function interface**  type FuncType = (n:number,m:number)=>void  Interface Employee{  company:string,  salary : number,  myFunc : FuncType  }  const myobj:Employee = {  company : “google”  salary : 10000000,  myFunc : (n,m)=>{  console.log(n\*m)  }  }  console.log(myobj);  myobj.myFunc(10,20); |

1. **Note : Sometime some methods dont want to ans some question like gender related so the below code is**

**Used**

|  |
| --- |
| subject?:string, |

1. **Functions in TS :**

|  |
| --- |
| **1. optional parameter : either you give or not doesnt matter** |
| type FuncType = (n: number, m: number, l?: number) => number;  const myFunc: FuncType = (n, m,l) => {      if (typeof(l) === "undefined") {          return n \* m;      }      else{          return n\*m\*l;      }  }  console.log(myFunc(5, 2)); |

|  |
| --- |
| **2. Default Parameter** |
| **Method 01**  type FuncType = (n: number, m: number, **l?: number**) => number;  const myFunc: FuncType = (n, m, l = 30) => {  *return* n \* m \* l;  }  console.log(myFunc(5, 2));  **Method 02**  const func = (n:number,m:number,l:number=20):number =>{  *return* n\*m\*l;  }  func(10,20) |

|  |
| --- |
| **3. Rest Parameter :**  In TypeScript, as in JavaScript, the rest parameter syntax (...) allows you to represent an indefinite number of arguments as an array. This feature is especially useful when you want to work with a varying number of arguments in a function or method. The rest parameter syntax is also known as the rest operator.  **Syntax** :  function functionName(...restParams: Type[]) {  // Function body  } |
| **Method 1**  const func = (...m:number[])=>{      console.log(m);      return m;  }  func(1,4,5,2,6,4)  **Output** : [1,4,5,2,6,4]  **Method 2**  type FuncType = (...m:number[])=>number[];  const func:FuncType = (...m)=>{      console.log(m);  *return* m;  }  func(1,4,5,2,6,4)  **Output** : [1,4,5,2,6,4] |

1. **Function With Object :**

|  |
| --- |
| type **myObj** = { title: string, description: string, thumbnail: string, id: number }  function getData(**myNewObj: myObj**):void {      console.log(myNewObj);  }  const myTestingObj = {      title: "Post1",      description: "This is description",      thumbnail: "thumbnail image",      id: 1  }  **getData(myTestingObj);** |