TypeScript

- TypeScript is an open-source, strongly-typed, object-oriented programming language developed by Microsoft.
- It is called a "superset" of JavaScript because:
  - All valid JavaScript code is also valid TypeScript code.
  - But it adds additional features that JavaScript does not have.
- The main focus of TypeScript is:
  - Static Type Checking: Checking data types before code runs.
  - Advanced Tooling: Powerful code suggestions, navigation, and refactoring support.
  - Scalability: Writing applications that can grow from small projects to massive codebases without breaking.
- While JavaScript is dynamic and flexible, it does not check types at runtime, which can cause bugs. TypeScript solves this by catching errors during development itself (before running the code).
- TypeScript cannot run directly in browsers or Node.js. It must first be compiled (transpiled) to plain JavaScript using the TypeScript Compiler (tsc).

Setting up the TypeScript Compiler (tsc)

Now, to work with TypeScript, you need to install the TypeScript compiler (tsc) on your system.

Step 1: Install Node.js: Download and install Node.js from https://nodejs.org/ according to your operating system i.e. Windows, macOS, or Linux.

Step 2: Install TypeScript Globally: Once Node.js and npm are ready, open your terminal/command prompt and run:

```
npm install -g typescript
```

- -g means "global" — it will install TypeScript system-wide.
- Now, the tsc command becomes available from anywhere.

Step 3: Verify Installation: After installation, check if it’s properly installed, using:

```
tsc --version
```

You can upgrade TypeScript later, using:

```
npm install -g typescript@latest
```

Step 4: Initialize a TypeScript Project (Optional but Recommended): Inside your project folder:

```
tsc --init
```

This creates a tsconfig.json file, which helps manage how TypeScript behaves in your project (compilation rules, output folder, etc.).

Your tsconfig.json file will looks like:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  }
}
```

which means:

- The TypeScript compiler (tsc) will look for your original .ts files inside the src/ folder and the code will be compiled from src/ to dist/
- Strict mode enabled (stronger type checking)
- Output JavaScript will be ES6 compatible.
- The compiled JS files will use CommonJS module syntax (require(), module.exports). This is mainly used in Node.js applications.

Writing and running your first .ts file

- First, create a simple .ts file (index.ts)

```
let instituteName: string = "Learn2Earn Labs Training Institute";
console.log(instituteName);
```

- Compile the TypeScript file to JavaScript

```
tsc index.ts
```

The above command will generate a index.js file.

- Run the compiled JavaScript file using Node.js

```
node index.js
```

output:

```
Learn2Earn Labs Training Institute
```

Common Mistakes:

- Forgetting to compile before running: Always compile .ts to .js.
- Running .ts directly in Node.js (not possible without extra setup like ts-node).

Compile the whole project

You can compile the whole typescript project using

```
tsc
```

It will convert all .ts files in src/ to .js files in dist/.

tsc –watch command

- tsc --watch command automatically watches your .ts files in your TypeScript project for any changes.
- As soon as you make a change and save a .ts file, TypeScript automatically detects the change and recompiles it.
- tsc --watch is most useful:
  - While developing applications quickly.
  - When you are working with multiple .ts files.
  - While debugging and experimenting with code.
  - In large projects, where running manual compile after every small change is a waste of time.

Commands Summary

Command	Purpose
tsc filename.ts	Compile one file manually
tsc	Compile entire project based on tsconfig.json
tsc --watch	Watch and auto-compile on every file change

ts-node-dev

- ts-node-dev is a powerful tool that lets you:
  - Run TypeScript files directly (no manual tsc compilation needed)
  - Restart your app automatically when you make changes
  - Watch .ts files for changes and re-run the code instantly
  - Faster than nodemon + tsc --watch because it uses Hot-Reloading
- It combines:
  - ts-node (which runs TypeScript directly without compiling manually)
  - nodemon (which watches your files and restarts the server automatically)
- It is even faster because it reloads only the modified files (called Hot Reloading).

Installing ts-node-dev

- In the terminal, install it globally (system-wide), using command:

```
npm install -g ts-node-dev
```

- Or install it locally (project-specific) for production-grade projects to keep versions locked, using command:

```
npm install --save-dev ts-node-dev
```

How to use ts-node-dev

After creating typescript project and typescript file (index.ts), use command:

```
ts-node-dev --respawn src/index.ts
```

It will:

- Start your TypeScript file directly.
- Watch for file changes.
- Restart the server automatically after every save!

Adding ts-node-dev to package.json

To avoid typing full commands every time, add a script inside package.json:

```
{
  "scripts": {
        "dev": "ts-node-dev --respawn src/index.ts"
  }
}
```

Now you can simply run:

```
npm run dev
```

Important Options with ts-node-dev

Option	Meaning
--respawn	Restart app if it crashes
--transpile-only	Speed up by skipping type-checking (good for speed during development)
--ignore-watch	Ignore some folders (e.g., node_modules)
--debug	Debugging support

Example command:

```
ts-node-dev --respawn --transpile-only src/index.ts
```

It will speeds up your development even more by skipping type-checking!

Difference Between ts-node, ts-node-dev, nodemon, and tsc

Tool	Purpose
tsc	Only compiles .ts to .js
nodemon	Watches .js files and restarts Node.js server
ts-node	Runs .ts files directly, no compilation
ts-node-dev	Runs .ts files directly + watches changes + hot reloads

Conclusion: ts-node-dev is the most powerful and developer-friendly tool for TypeScript development!

Common Errors with ts-node-dev (and fixes)

Error	Solution
Cannot find module	Check file paths
Syntax Errors in .ts files	ts-node-dev will stop; fix TypeScript errors
Not watching certain files	Use correct --watch and --ignore-watch options

Primitive Types

- In TypeScript, Primitive Types are the basic building blocks of data.
- They are simple, non-object types that store a single value.

1. **string Type:** A string type represents textual data — words, sentences, etc.

   Syntax

   ```
   let username: string = "nehajain234";
   ```

   Example

   ```
   let username: string = "nehajain234";
   console.log(username);

   username = 100;              // Error: Type 'number' is not assignable to type 'string'
   username = "pragati452";  // valid
   console.log(username);
   ```

   Template Strings (using backticks)

   ```
   let age: number = 25;
   let student: string = "Neha Jain";

   let sentence: string = `${student} is ${age} years old.`;
   console.log(sentence);
   ```

   Points to note

   - Only string values (like "Hello", 'World') are allowed.
   - If you try to assign a number or boolean, TypeScript will show an error.

2. **number Type:** The number type represents all numbers — integers, floats, hexadecimals, etc.

   Syntax

   ```
   let score: number = 100;
   ```

   Example

   ```
   let marks: number = 85;
   console.log(marks);

   marks = "Ninety";     // Error: Type 'string' is not assignable to type 'number'
   marks = 92;              // Valid
   console.log(marks);
   ```

TypeScript treats integers, floats, and negative numbers as number type.

Example

```
let pi: number = 3.14;
let temp: number = -10;
let hex: number = 0xff; // 255
console.log(pi, temp, hex);
```

3. **boolean Type:** The boolean type represents two values: true or false. Boolean types are mainly used for conditions, flags, status checks.

Syntax

```
let isCompleted: boolean = true;
```

Example

```
let isActive: boolean = false;
console.log(isActive);

isActive = "true";    // Error: Type 'string' is not assignable to type 'boolean'

isActive = true;      // valid
console.log(isActive);
```

Example

```
let isLoggedIn: boolean = true;
if (isLoggedIn) {
        console.log("Welcome back!");
} else {
        console.log("Please login first.");
}
```

4. **null Type:** The null type has only one value — null, which means empty or intentional absence of any object value.

Syntax

```
let user: null = null;
```

Example

```
let data: null = null;
console.log(data);

data = "Hello";       //Error: Type 'string' is not assignable to type 'null'
```

Points to note

- Used when you want to explicitly say: “This variable is empty”.
- Commonly used when resetting objects or values.

Nullable Variables Example

```
let profile: string | null = null;
profile = "Admin";
console.log(profile);
```

where, string | null is a union type — meaning profile can be either a string or null.

5. **undefined Type:** The undefined type has only one value — undefined. It means the variable has been declared but no value has been assigned yet.

Syntax

```
let city: undefined = undefined;
```

Example

```
let city: undefined = undefined;
console.log(city);

city = "Noida"; // Error: Type 'string' is not assignable to type 'undefined'
```

Example: Functions with no return value

```
function logMessage(): undefined {
        console.log("This function returns undefined.");
        return undefined;
}
logMessage();
```

Example: Nullable or Optional Variables

```
let address: string | undefined;
console.log(address); // Output: undefined

address = "Greater Noida";
console.log(address); // Output: Greater Noida
```

6. **any Type:** The any type disables type checking. You can assign anything (string, number, object, etc.) to a variable of type any.

   Syntax

   ```
   let anything: any = "Hello";
   ```

   Example

   ```
   let randomData: any = 10;
   console.log(randomData);

   randomData = "Welcome to Learn2Earn Labs Training Institute";
   console.log(randomData);

   randomData = true;
   console.log(randomData);

   randomData = { speciality: "Job Oriented Training Programs" };
   console.log(randomData);
   ```

   Points to note

   - any, removes all type safety.
   - It should be avoided as much as possible because it defeats the purpose of using TypeScript!
   - Use any only when:
     - ✓ Migrating old JavaScript code to TypeScript.
     - ✓ Working with very dynamic content like 3rd party libraries without types.

7. **unknown Type:** The unknown type is similar to any, but stricter. You can assign any value to an unknown variable, but you cannot use it until you narrow down (check) the type.

   Syntax

   ```
   let value: unknown = "Any Value";
   ```

   Example

   ```
   let result: unknown = 42;
   console.log(result); // Output: 42

   result = "Best Training Providers";
   console.log(result); // Output: Best Training Providers
   ```

```
result.toUpperCase(); // Error: Object is of type 'unknown'.

// Correct way
if (typeof result === "string") {
  console.log(result.toUpperCase()); // Output: BEST TRAINING PROVIDERS
}
```

Example with Functions

```
function handleData(input: unknown) {
        if (typeof input === "number") {
                console.log(input * 2);
         }
        else if (typeof input === "string") {
                console.log(input.toUpperCase());
         }
        else {
                console.log("Unknown data type.");
         }
}

handleData(10); // Output: 20
handleData("hello"); // Output: HELLO
handleData(true); // Output: Unknown data type.
```

Points to note

- unknown is safer than any.
- You must do type checking before you can operate on unknown values.

Arrays

- In TypeScript, an Array is a collection of multiple values of the same type stored together.
- You define an array by Specifying the element type followed by square brackets [].

Syntax

```
let arrayName: type[] = [values];
or
let arrayName: Array<type> = [values];
```

Example: Basic String Array

```
let fruits: string[] = ["Apple", "Banana", "Mango"];
console.log(fruits);
```

Points to note

- The array only allows strings.
- Trying to insert a number would throw an error.

Example: Basic Number Array

```
let scores: number[] = [90, 85, 77];
console.log(scores);
```

Points to note

- Only numbers are allowed.
- Useful when storing marks, salaries, counts, etc.

Example: Boolean Array

```
let passed: boolean[] = [true, false, true];
console.log(passed);
```

Points to note

- Each value must be true or false.
- Helpful for managing flags or condition results.

Example: Array Using Array<Type> Syntax

```
let cities: Array<string> = ["Delhi", "Mumbai", "Chennai"];
console.log(cities);
```

Points to note

- Same as string[], but different style.
- Preferred in some larger projects.

Example: Array of Objects

```
let users: { name: string; age: number }[] = [
        { name: "Prateek", age: 28 },
        { name: "Sonali", age: 26 }
];
console.log(users);
```

Points to note

- Each element must be an object with both name (string) and age (number).
- Perfect for real-world applications.

Example: Array of Mixed Objects

```
let products: { id: number; name: string; price: number }[] = [
  { id: 1, name: "Laptop", price: 50000 },
  { id: 2, name: "Phone", price: 30000 },
];
console.log(products);
```

Points to note

- Each element must follow { id, name, price } structure.
- Very useful in E-commerce applications.

Example: Array of Array (Matrix Representation)

```
let matrix: number[][] = [
  [1, 2],
  [3, 4],
  [5, 6]
];
console.log(matrix);
```

Points to note

- Array of arrays (2D array).
- Represents a matrix.

Important Points about Arrays

- Type checking is strict — you cannot push a different type accidentally.
- Arrays can be mutated (push, pop, shift, unshift, etc.).
- You can create readonly arrays using readonly keyword.

Tuples

- A Tuple in TypeScript is a special kind of array with Fixed length & Fixed types for each position.
- Each position in a tuple has a specific type defined.

Syntax

```
let tupleName: [type1, type2, type3, ...] = [value1, value2, value3, ...];
```

Example: Simple Tuple (String + Number)

```
let person: [string, number] = ["Sonali", 26];
console.log(person);
```

Points to note

- The first element must be a string.
- The second element must be a number.

Example: Tuple Representing Coordinates

```
let coordinates: [number, number] = [28.6139, 77.2090];
console.log(coordinates);
```

Points to note

- Two numbers representing (latitude, longitude).
- Very useful for maps, geolocation apps.

Example: Tuple with Boolean, String, and Number

```
let student: [boolean, string, number] = [true, "Vaibhav", 342];
console.log(employee);
```

Points to note

- Used when data has mixed types.

Example: Tuple Array (List of Tuples)

```
let data: [string, number][] = [
            ["ItemA", 10],
            ["ItemB", 20],
            ["ItemC", 30]
];
console.log(data);
```

Points to note

- An array where each element is a tuple ([string, number]).

Example: Tuple with Optional Elements

```
let userInfo: [string, number?] = ["Sonali"];
console.log(userInfo);
```

Points to note

- The second element (number) is optional.
- Tuples can have optional items using ?.

Example: Tuple with Function Return

```
function getUser(): [string, number] {
  return ["Sonali", 26];
}

const user = getUser();
console.log(user);
```

Points to note

- A function returning a tuple (name, age).
- Helps return multiple values safely.

Example: Tuple with Rest Elements

```
let address: [number, ...string[]] = [123, "Street Name", "City", "State"];
console.log(address);
```

Points to note

- First element must be a number (like house number),
- Remaining can be variable-length list of strings (street, city, state).

Example: Tuple with Labels (for better readability)

```
let person: [name: string, age: number] = ["Vaibhav", 30];
console.log(person);
```

Points to note

- Tuple labels are just for readability (in editors and IDEs).
- They do not change behavior, but make code more understandable.

Important Points about Tuples

- Order is important: You cannot shuffle types around.
- Length is fixed unless you allow optional elements or rest elements (...).
- Tuples allow better modeling of small fixed-size data structures (like (x, y), (id, name), etc.).