# **Learning TypeScript**

## ****1. Installation & Basic Commands****

**Install TypeScript globally**  
To install TypeScript globally on your system so you can use tsc from any directory:

npm i typescript -g

**Initialize TypeScript configuration**  
To create a tsconfig.json file in your project, which stores compiler options and settings:

tsc --init

**Compile a TypeScript file**  
To compile your app.ts file into app.js:

tsc app.ts

**Enable watch mode**  
To watch TypeScript files for changes and automatically recompile them:

tsc --watch

## ****2. Type Inference****

**Definition:**  
When TypeScript automatically infers the type of a variable or function based on the assigned value without explicitly mentioning the type.

**Example:**  
If you write:

* let a = 10; → TypeScript infers a as a number.
* let str = "harsh"; → TypeScript infers str as a string.

Here, you are not explicitly defining the type, but TypeScript understands it based on the value assigned.

## ****3. Type Annotation****

**Definition:**  
When you explicitly mention the type of a variable, parameter, or function return type.

**Example:**

* let a : number = 10;
* let str : string = "harsh";

Here, you are telling TypeScript the exact type of each variable.

## ****4. Using Type Annotation in Functions****

**Example:**  
If you have a function that adds two numbers:

Function definition:  
You specify the type of parameters as number and also mention the return type as number.

For example:

* function add(a : number, b : number) : number { return a + b; }

Here:

* a : number → parameter type annotation
* b : number → parameter type annotation
* : number after parentheses → function return type annotation

**Void Return Type Example:**  
If a function does not return anything:

* function greet(name : string) : void { console.log("Hello, " + name); }

## ****5. Type Aliases****

**Definition:**  
Type aliases allow you to create a custom name for a type. It improves code readability and makes complex or union types easier to reuse.

**Example:**  
If you want a variable to store either string or number:

* type StringOrNumber = string | number;
* let id : StringOrNumber = 123;
* id = "ABC123"; // also valid

**Example with function using type alias:**  
If you have a User type:

* type User = { name : string; age : number; }

And a function using it:

* function printUser(user : User) : void { console.log("Name: " + user.name + ", Age: " + user.age); }

**6. Interface**

**Definition:**  
An interface is used to define the structure of an object. It enforces that an object must have specific properties with their types.

**Example:**  
If you want to define a user object with name and age:

• interface User {  
  name : string;  
  age : number;  
 }

Using interface in a function:

• function printUser(obj : User) {  
  console.log("name is " + obj.name + " and age is " + obj.age);  
 }

**Correct usage:**

• printUser({ name : "harsh", age : 22 });

**Wrong usage:**

• printUser({ name : "harsh", age : "22" }); // age should be number, not string.

### Extending Interfaces

**Definition**:  
When one interface inherits properties from another interface using the extends keyword, allowing reusability and structured design in TypeScript.

Create a USER interface  
Define an interface with properties:  
• name : string  
• email : string  
• password : string  
• age : number

Extend USER interface with ADMIN  
Create an ADMIN interface that extends the USER interface and adds its own property:  
• isAdmin : boolean

**Example**:

We create a user object with:  
• name as "alpha"  
• age as 21  
• email as "alpha@a.com"  
• password as "password"

We create an admin object with:  
• name as "admin1"  
• age as 22  
• email as "harsh@h.com"  
• password as "admin"  
• isAdmin as true

**Explanbation**:  
Here, the ADMIN interface extends USER, meaning it inherits all properties of USER (name, email, password, age) and has its own property isAdmin. This ensures the admin object contains both user properties and admin-specific properties without redefining them.

### Interface Merging

**Definition**:  
When two interfaces have the same name in TypeScript, they merge into a single interface containing all combined properties.

**Example**:

If you declare interface abab with:  
• name : string

And declare another interface abab with:  
• email : string

Both interfaces merge, so when using the abab type, both name and email properties are required.

**Explanation**:  
This feature is useful for extending types across different parts of your codebase without modifying the original interface directly.

### 7. Class and Objects

**Definition**:  
A class is a blueprint for creating objects with specific properties and methods in TypeScript. Objects are instances of a class.

Create a Device class  
Define a class Device with properties:  
• name = "dell"  
• price = 50000  
• category = "laptop"

Create objects of Device  
We create two objects using the Device class:

d1 is created as:  
• let d1 = new Device()

d2 is created as:  
• let d2 = new Device()

**Explanation**:  
Here, both d1 and d2 are instances of the Device class and have the same default properties: name as "dell", price as 50000, and category as "laptop". Using classes helps in creating multiple objects with the same structure efficiently.

This in oop  
  
when we want to access any variable that is being created in class and if we want to access it in the method we use this reference to give current context.

**this in OOP**

**Definition**:  
In object-oriented programming, this is a reference used to access variables or properties of the current class within its methods. It gives the current context of the object.

**Usage**:  
When we want to access any variable created in a class inside its method, we use this to refer to that variable belonging to the current object.

**Example**:

If we have a class with a property:  
• name = "alpha"

And a method inside the class wants to access this name property, we write:  
• this.name

**Explanation**:  
Here, this refers to the current object of the class. Without this, the method will not know that you are referring to the class-level property. It ensures we are accessing the property or method of the current instance on which the function is called.

**8. Generics**

**Definition:**

Generics allow us to create reusable components, functions, or classes that can work with different data types without losing type safety in TypeScript. It provides a way to create flexible and type-safe code.

**Example:**

function ab<T>(val: T) {

console.log(val);

}

ab<string>("harsh");

ab<boolean>(true);

**Explanation:**

Here,

* We define a **generic function ab** with a type parameter <T>.
* Inside the function, **val is of type T**, which means it can take any data type that is passed during function call.
* When we call ab<string>("harsh");, **T becomes string**, so val is a string.
* When we call ab<boolean>(true);, **T becomes boolean**, so val is a boolean.

Using generics ensures:

* **Code reusability:** Same function works with different types.
* **Type safety:** Prevents passing unintended types, avoiding runtime errors.

**Generics in OOP:**

We can also use generics in classes to create flexible class definitions that work with different data types.

**Example:**

class Box<T> {

content: T;

constructor(value: T) {

this.content = value;

}

display(): void {

console.log(this.content);

}

}

let b1 = new Box<string>("Hello");

let b2 = new Box<number>(100);

b1.display(); // Output: Hello

b2.display(); // Output: 100

**Explanation:**

* **Box is a generic class** with type parameter <T>.
* content is of type T, and the constructor assigns the passed value to content.
* display() method prints the content.
* b1 is created with T as string.
* b2 is created with T as number.

**Summary:**

Generics help in creating components or classes that work with **multiple types while retaining type safety**. It makes the code **more flexible, reusable, and robust**.

**9. Type Assertion**

**Definition:**

Type assertion in TypeScript is used to tell the compiler about the specific type of a variable when it cannot infer it automatically. It does not change the type at runtime but helps with type checking during compilation.

**Example:**

let a: any = "test";

(a as string).indexOf("e");

**Explanation:**

Here,

* a is of type any.
* We assert a as string using (a as string).
* This allows us to use string-specific methods like indexOf() without compiler errors.
* **Syntax:** (variable as type) or <type>variable (second syntax is not recommended in JSX).

Type assertion is used **when you are sure about the type of a variable** to assist TypeScript in type checking.

**10. Type Casting**

**Definition:**

Type casting converts one data type to another. In TypeScript (and JavaScript), we can convert types using built-in functions like Number(), String(), etc.

**Example:**

let num = Number(12);

let num1 = Number("127");

let str = String("testing type casting");

console.log(num, str);

**Explanation:**

Here,

* Number(12) converts 12 to a number (although it is already a number here).
* Number("127") converts string "127" to number 127.
* String("testing type casting") converts the value to string type.

Type casting is useful when we need **conversion between string, number, boolean, etc. for operations, comparisons, or validations**.

**11. Non-null Assertion Operator**

**Definition:**

The non-null assertion operator ! is used when we want to tell TypeScript that a variable is **not null or undefined** at that point, even if its type allows it.

**Example:**

let d: null | undefined | string = "test1";

d!.charAt(3);

**Explanation:**

Here,

* d can be null, undefined, or string.
* Using d! tells the compiler that **d is definitely not null or undefined here**.
* d!.charAt(3) calls charAt on d assuming it has a string value.

**Usage:**

Use ! carefully because:

* It **overrides strict null checks**.
* If d is actually null or undefined at runtime, it will throw an error.