**TYPESCRIPT**

**[2025]**

|  |
| --- |
|  |

# : Introduction:

* What is Typescript?
* Why Typescript?
* We can define data types here sometime we can call static typing.
* Improved code maintainability.
* Code quality.
* Advanced features.
* Important Typescript features?
* Define data types.
* Interfaces.
* Decorators.
* Namespaces.
* Type interface.
* Advanced features.
* Code quality.
* Installation setup?
* Locally installation means only one projects inside typescript install.

npm install typescript --save-dev

* Globally installation means we can use typescript entire our system (laptop).

npm install -g typescript

* How to run Typescript File?
* Create, app.ts file.
* Run this command to generate Js file because browser understand only JS file not TS (compilation every time).

npx tsc app.ts

* Compilation every time specific file.

npx tsc app.ts --watch

* Compilation every time all file.

tsc --watch

* Interview Questions?
* **tsc full form** → TypeScript Compiler.
* **--watch command use** → Recompiles automatically on file changes.
* **npm full form** → Node Package Manager
* **Can we create name, name variable in TypeScript?**

No, because name, name is already defined variable in window object that is why we cannot create variable name, name.

# : Data Type in TypeScript:

* What are Data Types?

Data types specify what kind of data can be stored in a variable.

* Data Types in Typescript?
* Primitive types.
* Objects types.
* Special types.
* Advanced types.
* Function types.
* What are primitive data types?

Primitive data types are inbuild data types.

Ex: number, string, Boolean, bigInt, null, undefined, symbol.

* Object Data types?

These included Arrays, Objects, Tuples.

Ex: Array, Object, Tuple.

* Array: array is a collection of elements same types, because it is number types of arrays.

Ex: var num: number [] = [1,2,3,4,5,6,7];

* Tuples: Tuples is an array here we can include numbers, string, object, Boolean, all types of data.

Ex: var person: [number, string] = [25, “Ashu”];

* Object: Object is a key and values pair which is separate defined data.

Ex: var obj: {id: number, name: string, isActive: boolean} = {id:1, name: “Ashu", isActive: true};

console.log(obj);

* Special Data Types?

These are extra types provided by Typescript,

Ex: Any, unknown, void, never.

* Any: any can hold any types of data.
* Unknown: unknown like any but must be typed checked before use.
* Void: Void used for functions that don’t return anything.
* Never: We can use Never with the function which we do not want to return.
* Advanced Data Types?

Ex: union, intersection, Type alias, Enum, Literal types.

* Union: union allows a variable to hold multiple types.
* Intersection: intersection is a combination of multiple types into one.
* Type alias: Type alias, you can create a custom name for a type.
* Enum: Enum defines a set of named constants.
* Literal types: Literal types, restrict variable to specific values.
* Function Data Types?

Defines a type of a function.

# : Primitive Data Types & TypeScript Configuration:

* **What are primitive data types?**

Primitive data types are inbuild data types.

Ex: number, string, Boolean, bigInt, null, undefined, symbol.

**Syntax**:

var variable\_name: datatypes = Value;

* **Number Data Types**: Represents both integer and floating-point numeric values.
* **String Data Types**: Represents textual data enclosed in single, double, or backticks.
* **Boolean Data Types**: Represents only two values: true or false.
* **BigInt Data Types**: Represents integers of arbitrary length, larger than Number can hold. Maximum Integer Number, MAX\_SAFE\_INTEGER = 2^53 -1 (TWO KI POWER 53 MINUS ONE)
* **Null Data Types**: Represents an intentional absence of any object value.
* **Undefined Data Types**: Represents a variable that has been declared but not assigned a value.
* **Symbol Data Types**: Represents a unique and immutable identifier.
* **TypeScript Config file:**
* How to generate config file:

tsc --init

* Use of config file:

We can change according to my use case.

* Converts all TS file into JS file:

tsc

* Compilation every time all file.

tsc --watch

# : Object Data Types:

* **Object Data types?**

These included Arrays, Objects, Tuples.

Ex: Array, Object, Tuple.

* **Array: Array** is a collection of elements of the same or mixed data types stored in a single variable. Elements are accessed by index.

let numbers: number[] = [1, 2, 3, 4];

let mixed: (string | number)[] = ["Hello", 42];

* **Object: Object** is a collection of keys–value pairs used to represent structured data. Keys are usually strings; values can be any type.

let person: {name: string; age: number } = { name: "Alice", age: 25 };

* **Tuple: Tuple** is a fixed-length array with predefined types for each position, ensuring order and type safety.

let user: [string, number, boolean] = ["Alice", 25, true];

# : Special Data Types:

* **Special Data Types?**

These are extra types provided by Typescript,

Ex: Any, unknown, void, never.

* **Any:** any can hold any types of data.

let variable: any;

* **Unknown:** unknown like any but must be typed checked before use.

let variable: unknown;

* **Void:** Void used for functions that do not return anything.

function functionName(): void;

* **Never:** We can use Never with the function which we do not want to return.

function functionName(): never;

# : Advanced Data Types:

* **Advanced Data Types in Typescript?**

There are mainly five types of advanced data types in Typescript.

Ex: union, Intersection, Type Alias, Enum, Literal types.

* **Union: Union** Allows a variable to hold values of multiple types.

**Example:**

let id: string | number;

id = "ABC123";

id = 101;

* **Interface:** An interface in TypeScript is a way to define the structure (shape) of an object. it specifies the properties and methods that an object must have, without providing the actual implementation.
* **Intersection: Intersection** Combines multiple types into one, requiring all properties.

**Example:**

type Person = { name: string };

type Employee = { id: number };

type Staff = Person & Employee;

let staff: Staff = { name: "Alice", id: 101 };

* **Type: Type** Creates a custom name for a type to improve readability.

**Example:**

type UserID = string | number;

let userId: UserID = 123;

* **Enum: Enum** Defines a set of named constants.

**Example:**

enum Direction {

Up,

Down,

Left,

Right,

}

let move: Direction = Direction.Up;

* **Literal Types: Literal Types** Restricts a variable to specific values only.

**Example:**

let role: "admin" | "user" | "guest";

role = "admin"; // ✅ allowed

// role = "manager"; ❌ not allowed

* Type Vs Interface and Intersection?
* We can use extends with interface not use with type.
* We can use type with union not with Interface.
* We can type with intersection very easily.
* We cannot interface with intersection.

# : Dom handling and Typecasting:

* **DOM Handling in TypeScript?**

DOM handling in TypeScript means accessing and manipulating HTML elements (like button, input, image, div) using TypeScript.

The advantage of TypeScript is that it provides type safety + autocompletion, so you get fewer runtime errors.

* **Typecasting (Type Assertion) in TypeScript?**

Typecasting (Type Assertion) in TypeScript means telling the compiler what specific type a variable is.

When you query DOM elements, TypeScript usually gives them a generic type Element | null.

If you know it is a certain type (like input, button, or image), you cast it to the correct type.

* Two common ways:
* let el = document.querySelector("input") as HTMLInputElement; // common way
* let el2 = <HTMLInputElement>document.querySelector("input"); // alternative (avoid in JSX)
* Example 1: Reading input value.

function handleInput() {

let inputEl = document.querySelector("input") as HTMLInputElement;

console.log("User typed:", inputEl.value);

}

* Example 2: Changing an image

function handleImage() {

let imgEl = document.querySelector("img") as HTMLImageElement;

imgEl.src = "https://picsum.photos/300/200";

imgEl.alt = "Random Image from TypeScript";

}

* Example 3: Non-null Assertion (!)

function handleHeading() {

let heading = document.querySelector("h1")!; // ! means "definitely not null"

heading.textContent = "Hello from TypeScript!";

}

* **Quick Summary:**
* DOM Handling → Accessing and manipulating HTML elements with TypeScript.
* Typecasting (Assertion) → Telling the compiler the exact type of a DOM element (HTMLInputElement, HTMLImageElement, HTMLButtonElement, etc.).
* Benefits → Type safety, autocompletion, fewer runtime errors.

# : Classes in TypeScript:

* **Class:** A **class** is a blueprint for creating objects with properties (variables) and methods (functions).

**Points:**

* constructor initializes the object.
* this refers to the current object.
* Objects are created using new.
* **Access Modifiers?**

Access modifiers control the visibility of class members.

* public (default) → Accessible everywhere.
* private → Accessible only inside the class.
* protected → Accessible inside the class & subclasses.
* **Inheritance in TypeScript?**

Inheritance allows one class (child) to use properties & methods of another class (parent) using the extends

**Points:**

* extends is used for inheritance.
* Child classes can **override** parent methods.
* super () is used to call parent constructor or methods.

# : Modules in TypeScript:

* **Wha is a Module?**

A **module** in TypeScript is simply a **file** that contains some code (classes, functions, variables) which can be **exported** and **imported** into other files.

By default, **every .ts file is its own module**.

* **Why Modules?**

Code organization → break big project into smaller files.

Reusability → write once, use multiple times.

Encapsulation → only expose what you export.

* **Import & Export in TypeScript?**
* **Export**: In TypeScript, *export* is used to make a class, function, or variable available to be used in other files.
* **Import**: *Import* is used to bring in classes, functions, or variables from other files so you can use them in the current file.

In short:

**Export = share** something from a file.

**Import = use** something that was shared from another file.

* **Getter and Setter in TypeScript**
* **Getter**: A *getter* is a special method that lets you read the value of a private property as if it were a normal property.
* **Setter**: A *setter* is a special method that lets you set (or update) the value of a private property in a controlled way.

In short:

**Getter = read access** to private data.

**Setter = write access** to private data.

# : Interfaces & Static Keyword:

* **Interface with Class in TypeScript?**

An **interface** defines a **contract** (the structure) for a class.

Any class that **implements** an interface must define all the properties and methods declared in that interface.

* **Static Keyword in TypeScript?**

A **static property or method** belongs to the **class itself**, not to instances (objects).

You can access it using the class name directly, without creating an object with new.

# : Advanced Type Handling:

* **Type Guard in TS?**

Type Guard is a way to narrow down a type at runtime so TypeScript knows exactly what type (data type) you are working with. There are 3 types we can check data type at runtime.

* typeof: normal variable type check.
* instanceof: check class type and
* Custom type: interface type check.
* **Generics in TS?**

A feature to write reusable code that works with any type instead of one fixed type.

* **keyof Operator in TS?**

Keyof operator in TS is used to get the keys of an object type as a union of string literals.

* **Index Signatures**

Index signature allows you to define objects with dynamic keys (keys not known in advance).

* **In short:**
* Type Guard → runtime check to refine type.
* Generics → reusable, type-safe functions/classes.
* keyof → get keys of a type.
* Index Signature → allow objects with flexible keys.

# : Utility Types:

* **What is Utility in TypeScript?**

In TypeScript, utility types are built-in helper types that make it easier to transform, modify, or extract types.  
They are predefined generic types provided by TypeScript to simplify type manipulations.

* **Common Utility Types:**
* **Partial<T>:** Makes all properties optional in each type.Useful when updating an object partially.
* **Required<T>:** Makes all properties mandatory (opposite of Partial).Even optional properties become required.
* **Readonly<T>:** Makes all properties of a type read-only (cannot be reassigned).
* **Pick<T, K>:** Creates a new type by picking only specific properties from another type.
* **Omit<T, K>:** Opposite of Pick.Creates a type excluding certain properties.
* **Exclude<T, U>:** Excludes types from T that are **assignable to U**.
* **Extract<T, U>:** Opposite of Exclude. Extracts only types from T that are assignable to U.
* **NonNullable<T>:** Removes null and undefined from a type.
* **Record<K, T>:** Constructs an object type with keys K and values of type T.
* **Summary Table:**

| **Utility Type** | **What it does** |
| --- | --- |
| Partial<T> | Makes all properties optional |
| Required<T> | Makes all properties mandatory |
| Readonly<T> | Makes all properties immutable |
| Pick<T, K> | Selects a subset of properties |
| Omit<T, K> | Removes given properties from a type |
| Exclude<T, U> | Removes from T those types that are assignable to U |
| Extract<T, U> | Keeps only those types from T assignable to U |
| NonNullable<T> | Removes null and undefined |
| Record<K, T> | Creates an object type with keys K and values T |

# : Namespaces & Decorators:

* **What is Namespace in TS?**

A **namespace** is a way to group related code (classes, functions, interfaces, variables) under a single name to avoid naming conflicts.

* Members inside a namespace are **private by default**.
* Use export to make them accessible outside.
* **How to import namespace?**

Namespaces are often used across multiple files.

Example:

You use /// <reference path="file.ts" /> to import.

Make sure tsconfig. json has "module": "system" or "amd" if needed.

* **Decorators in TypeScript?**

A **decorator** is a special function (with @) that can attach behavior to classes, methods, properties, or parameters.

It allows meta-programming (modifying or extending behavior).

* **You must enable it in tsconfig. json:**

"experimentalDecorators": true

* **Override Functions with Decorators?**

You can override or extend methods using decorators.

# : Async Programming

* **What is a Promise?**

A Promise in TypeScript (and JavaScript) is an object that represents the result of an asynchronous operation (which may complete in the future).

**It has 3 states:**

* pending → operation still running
* fulfilled → operation finished successfully
* rejected → operation failed
* Promises are mainly used for handling asynchronous tasks like API calls, file operations, or timers.
* **Typed Promise in TypeScript**

In TypeScript, you can define the type of data a Promise will return using Promise<Type>.

* **API Calls in TypeScript (with Promises)**

You can use fetch (built-in) or axios (library). Both return Promises.

* **Summary:**
* Promise → represents future completion/failure of async task.
* Define → new Promise((resolve, reject) => { ... }).
* Typed Promise → Promise<Type> ensures type safety.
* API Calls → Use fetch or axios with Promise<T> or async/await.

# : End TypeSecipt.