**TYPESCRIPT**

TypeScript is JavaScript with added syntax for types. TypeScript is a syntactic superset of JavaScript which adds **static typing**, means that TypeScript adds syntax on top of JavaScript, allowing developers to add **types**

**Why should I use TypeScript?**

**JavaScript** is a **loosely typed language**. It can be difficult to understand what types of data are being passed around in JavaScript.

TypeScript allows **specifying the types of data being passed around** within the code, and has the **ability to report errors** when the types don't match.

A common way to use TypeScript is to use the official **TypeScript compiler**, which transpiles TypeScript code into JavaScript, ( means it runs anywhere where Javascript runs ) , TypeScript is transpiled into JavaScript using a compiler.

***npm install typescript --save-dev***

**Configuring the compiler :-**

The compiler can be configured using a **tsconfig.json** file.

You can have TypeScript create tsconfig.json with the recommended settings with:

***npx tsc --init***

<https://www.typescriptlang.org/tsconfig>

# **TypeScript Simple Types :-**

TypeScript supports some simple types (**primitives**) you may know. There are three main primitives in JavaScript and TypeScript.

* **boolean** - true or false values
* **number** - whole numbers and floating point values
* **string** - text values like "TypeScript Rocks"

## Type Assignment :-

When creating a variable, there are two main ways TypeScript assigns a type:

* **Explicit 🡺 let firstName: string = "Dylan";**
* **Implicit** 🡺 let firstName = "Dylan";

let firstName: string = "Dylan"; // type string  
firstName = 33; // attempts to re-assign the value to a different type

**JavaScript** will **not** throw an error for mismatched types.

## Unable to Infer :-

TypeScript may not always properly infer what the type of a variable may be. In such cases, it will set the type to ***any*** which disables type checking

\*\* This behavior can be disabled by enabling noImplicitAny as an option in a TypeScript's project tsconfig.json

# **TypeScript Special Types :-**

TypeScript has special types that may not refer to any specific type of data.

## *Type: any*

***any*** is a type that disables type checking and effectively allows all types to be used.

let u = true;  
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.  
Math.round(u); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number'.

Setting any to the special type any disables type checking:

let v: any = true;  
v = "string"; // no error as it can be "any" type  
Math.round(v); // no error as it can be "any" type

## *Type: unknown*

unknown is a similar, but safer alternative to any. TypeScript will prevent unknown types from being used, unknown is best used when you don't know the type of data being typed. To add a type later, you'll need to **cast** it.

**Casting**  is when we use the "as" keyword to say property or variable is of the casted type.

There are times when working with types where it's necessary to override the type of a variable, such as when incorrect types are provided by a library.

Casting is the process of overriding a type.

## Casting with as

A straightforward way to cast a variable is using the ***as*** keyword, which will directly change the type of the given variable.

let x: unknown = 'hello';  
console.log((x as string).length);

## *Casting with <>*

Using <> works the same as casting with as.

let x: unknown = 'hello';  
console.log((<string>x).length);

## *Type: never*

never effectively throws an error whenever it is defined.

let x: never = true; // Error: Type 'boolean' is not assignable to type 'never'.

never is rarely used, especially by itself, its primary use is in advanced generics.

.

***Type: undefined & null***

***undefined*** and ***null*** are types that refer to the JavaScript primitives undefined and null respectively.

let y: undefined = undefined;  
let z: null = null;

These types don't have much use unless strictNullChecks is enabled in the tsconfig.json file.

# **TypeScript Arrays**

TypeScript has a specific syntax for typing arrays.

const names: string[] = [];  
names.push("Dylan"); // no error

// names.push(3); // Error: Argument of type 'number' is not assignable to parameter of type 'string'

## Readonly

The readonly keyword can prevent arrays from being changed.

### **Example**

const names: readonly string[] = ["Dylan"];  
names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.  
// try removing the readonly modifier and see if it works?

## Type Inference

TypeScript can infer the type of an array if it has values.

### **Example**

const numbers = [1, 2, 3]; // inferred to type number[]  
numbers.push(4); // no error  
// comment line below out to see the successful assignment  
numbers.push("2"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.  
let head: number = numbers[0]; // no error

# **TypeScript Tuples**

## Typed Arrays

A **tuple** is a typed [array](https://www.w3schools.com/js/js_arrays.asp) with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array:

// define our tuple  
let ourTuple: [number, boolean, string];  
  
// initialize correctly  
ourTuple = [5, false, 'Coding God was here'];

Even though we have a boolean, string, and number the order matters in our tuple and will throw an error.

## Readonly Tuple

A good practice is to make your **tuple** readonly.

// define our tuple  
let ourTuple: [number, boolean, string];  
// initialize correctly  
ourTuple = [5, false, 'Coding God was here'];  
// We have no type safety in our tuple for indexes 3+  
ourTuple.push('Something new and wrong');  
console.log(ourTuple);

O/p:- [ 5, false, 'Coding God was here', 'Something new and wrong' ]

// define our readonly tuple  
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];  
// throws error as it is readonly.  
ourReadonlyTuple.push('Coding God took a day off'); //**ERROR**