**TypeScript**

TypeScript is superscript of JavaScript. It enables us to include some rules during the development of the program to avoid the common runtime errors in the JavaScript files.

**Static Checking:**

Typescript uses static checking. This means it checks and gives the errors without running the program. It is called static checking.

Helps us find errors!

Analyses our code as we type

Only exists in development.

**Object Types:**

1. Object
2. Arrays
3. Enum
4. Tuple
5. Others.

**Variable Types:**

Assigning a basic type to a variable is easy. We just had to use “ : “ after the variable name and specify the data type. It is also called type annotation.

Ex: **let/const/var variable name : data type = value ;**

let area : number = 678;

Once assigning a type to a number. The variable only accepts that type of values. If we mismatch the data types TypeScript will give us an error.

**Numbers:**

Some programming languages have many number types such as float, int etc. But TypeScript has only one data type for numbers and that is **“ number “** data type.

All numbers (float, int) are just numbers.

Numbers can be typed with a single type annotation called **number** (all lowercase).

let num : number =7803;

**Boolean:**

Boolean variables represent simple **true** or **false** values.

Booleans can be typed with a simple type annotation of **“boolean".**

**Ex: let isLogin : boolean = false;**

**Type Inference:**

Type Inference refers to typescript compiler’s ability to infer types from certain values in your code.

TypeScript can remember a value’s type even if you didn’t provide a type annotation, and it will enforce that type moving forward.

**Any:**

‘any’ is an escape hatch! It runs off the type checking for this variable so you can do your thing.

Note: It sort of defeats the purpose of TS and types. So use it sparingly.

Ex: let place : any = 89;

place = “psh”; // Does not show error since the variable type is any.

**When to use type annotation:**

In some cases like initializing a variable at the time of its declaration we don’t need to use type annotation. Because TS’s type inference automatically assigns the type to a variable.

But if we just declarable and later on we initialize it we need to **use** the **type annotation** for the variable at the time of its declaration because **if we don’t do that** then the type of that **variable type becomes** **any**.

Ex: let num; // Just declaring but not initializing will set the type to any

num = 5;

num = “psh”;

num.working();

num();

num.firstname; // Does not give error because type of num is “any”.

**Function Parameter Types:**

In TS we can specify the type of function parameters of a function at the time of its definition.

This allows TS to enforce types for values being passed to the function. Typing parameters is just like typing variables.

Ex: function add (p1 : number, p2: number )

{

console.log(p1 + p2);

}

add(4, 3) // Does not give error

add(‘1’, 2) // Gives error

**Assigning default parameters to a function:**

Function greet (greeting : string = “hello”, name: string = “user”) // Default

{ //parameters

Console.log(p1 + “ “ +p2);

}

greet(“hi”, “Prashanth”) => output: hi Prashanth;

greet() => output: hello user

**Function Return types:**

We can specify the type returned by a function. Even though TS already infers it, I like to provide the explicit annotations.

Add the type annotation after the function parameter list.

Ex: function add ( p1: number = 0, p2: number = 0) : number //Return type

{

return p1 + p2;

}

Here, by assigning an annotation to the function return type type script tells us that the function should return a value of type number.

Return types include: number, boolean, string, void.

**Anonymous Functions:**

When TS can infer how an unnamed function is going to be called, it can automatically infer it’s parameter’s types.

const colors = [“red”, “yellow”, “blue”);

colors.map(color => {

console.log(color.toUpperCase());

}

In the above example even if don’t specify the type of the color variable in the function, Ts will not assign the “any” type to that variable because it can infer how the function is going to be called and assigns the data type based on that.

In the above case the type of color is “string”.

**Void Type:**

Void is a return type for functions that didn’t return anything. It means just that – this function is void of any data.

TS can infer this type fairly well, but sometimes it may want to annotate a function with a void return explicitly.

Ex: const function counter (p1 : number = 1) : void

{  
 for(let I = 0; I <= p1; i++)

{

Console.log(i);

}

}

In the above example the function counter does not return anything. So it’s return type can be set to void. Or TS does it automatically with its type inference.

If the return type is annotated to void and you try to return any value then it will give an error.

**Never:**

The “ never “ type represents values that never occur. We might use it to annotate a function that always throws an exception, or a function that never finishes executing.

Don’t confuse with ***void***– void returns undefined or null, which is technically still a type of value. With **never –** a function doesn’t even finish executing.

Ex: const neverStop () : never =>

{

While(true)

{

Console.log(“running”);

}

}

The above function never returns a value as it always returns because the condition in the while loop will always evaluates to be true.

Cons giveError(msg: string) =>{

Throw new Error(“msg”);

}

**OBJECT TYPES:**

Objects can be typed by declaring what the object should look like in the annotation.

Accessing a property that is undefined or performing operations without keeping types in mind will throw errors!.

// A function with an object as parameter

Const printName = (name : { first: string ; last: string } ) =>{

Return `name: ${first} ${last}`;

}

printName({first: “Prasanth”, last: “bocha”{);

Here, the printName function accepts an object type of parameter and we also annotated the contents of the keys and their data types.

Using object as a return type :

Function obj = (p1 : {first: string{ : {returned: boolean{ =>

{

Console.log(p1.first);

Return {returned: true{

}

The above function takes an object parameter and returns an object value. The type annotations of the object’s contents are also specified.

Excess properties:

If we pass other properties in the object literal directly to the function it will give us an error. Because TS checks for the known properties in the passed parameter object. If there’s an unknown property passed directly as an object parameter it will give us an error.

Const printName = (name : { first: string ; last: string { ) =>{

Return `name: ${first{ ${last{`;

}

printName({first: “Prasanth”, last: “bocha”, age: 89});

// The above line will give us an error because age is not defined in the annotation of the function’s parameter.

**But,** if we pass the same object as a parameter by storing it in a variable, then TS don’t give us an error.

Ex:

Const printName = (name : { first: string ; last: string { ) =>{

Return `name: ${first{ ${last{`;

}

Let obj = {first: “Prasanth”, last: “bocha”, age: 22}

printName(obj);

// The above line will not give us an error. It only checks for the known properties and ignores the rest of the properties.

**The order of the properties will not be checked, only the property’s existence will be checked.**

**Type Alias:**

Instead of writing object types in an annotation, we can declare them separately in a **type alias**, which is simply the desired shape of the object.

This allows us to **make our code more readable** and even **reuse the types elsewhere in our code.**

// A type alias

Ex: type Person = {

name : string;

age : number;

};

// Use the type alias in the annotation

Const sayHappyBirthday = (person : Person) => {

Return `hey ${person.name{, congrats on turning ${person.age{);

}

sayHappyBirthday({name : Prashanth, age: 22{);

**Nested Objects:**

const describePerson = (person: {

name: string;

age: number;

parentNames : {

mom: string;

dad: string

{

}) =>

{

return ` Person: ${name{,

Age: ${age{,

Mom: ${parentNames.mom{

Dad: ${parentNames.dad{ `

}

describePerson( {name: “Prashanth”, age: 22, parents: { mom: “etc”, dad : “etc”{{);

**Optional properties;** We can specify the optional properties, which are completely optional the TS will not give an error even if the optional properties are missing. This can be achieved by using “ ? “ after the annotation variable name.

Ex: type Person : {

Name : string;

Age : number;

Salary?: number;

};

Const obj : Person = {name: “Psh”, age: 22};

Function describePerson (person : Person) {

Console.log(person.name, person.age, person.salary)

}

describePerson(obj);

If we run the above code the parameter obj does not contain the optional parameter value. So if we try to access it, then the result will be undefined.

But In this way we can create optional variables.

**Readonly Modifier:**

By using this keyword we can mark properties as “**readonly”,** then TS will give an error if we try to change the properties by writing on them.

We can use it as follows:

Ex:

type User = {

readonly id : number;

name: string;

}

Const user : User = { 1234, “PSH” };

Console.log(user) // Works fine because we’re just reading it

user.id = 13435 // Gives us an error because we are rewriting it.

**Intersection Types:**

We can have multiple types and combine them by using ampersand sign **“&”.**

Ex:

type Circle = {

radius: number;

}

type Color = {

color: string;

}

type coloredCircle = Circle & Color;

The above line states that the coloredCircle type is a combination of Circle and Color types and has the properties of both types. So if we create a obj of type coloredCircle then we should specify the radius and also the color for that particular object.

let happyFace : coloredCircle = { radius: 7, color: “yellow” };

If we want to specify any other properties apart from the available ones then we can use the following syntax.

let coloredCircle = Circle & Color & {

area : number;

volume: number;

}

By specifying another ampersand we can add new properties to coloredCircle type apart from the properties that are available to Color and Circle types.

**Arrays:**

Arrays can be typed by using a type annotation followed by empty array brackets, like number[[ for an array of numbers.

These arrays only allow data of that type inside them.

Ex:

Const arr : [] = [] ;

This means that the array is an empty array. We cannot add elements into that array.

const arr : string [];

This means arr is an array of type string.

Const arr : number[] = [1, 2, 4, 4, 5];

Const arr = [] // This array is of type **ANY**

type Point = {

x: number;

y : number;

}

Const points : Point[] = [];

points.push({ x: 7.3, y: 5.8});

**Other syntax for array:**

Let str : Array <string> = [‘psh’, ‘hi’, ‘bocha’];

**Multi-Dimensional Arrays:**

Const arr : string [] [] = [

[‘p’, ‘s’, ‘h’],

[‘s’, ‘u’ ‘d’]

]

The above array is an example of a two-dimensional array.

**Union Type:**

Union types allow us to give a value a few different possible types. If the eventual value’s type is included, then TS will not give any error.

We can create a Union type by using a single “ | “ (pipe character) to separate the types that we want to include. It’s like saying “ This thing, is allowed to be like this, this, or this ". Typescript will enforce it from there.

Ex:

Const guessAge = (age : number | string ) =>{

return `your age is : ${age}`;

};

**Type narrowing in Functions while using union:**

Function calcTax (price: number | string, tax : number)

{

If(typeof price === “string”)

{

price = parseFloat(price.replace(“$”, “”));

return price;

}

Else

{

return price \* tax;

}

}

In the above function we are checking the type of price value and applying some logic according to the type of the price parameter. This is called “ Type narrowing”.

**Union types and arrays:**

When we want to declare an array which contains numbers and strings we can use Unions. The syntax will be as follows:

Ex: const arr : (number | string) [ ] = [ ];

By using the above procedure we can create arrays which can hold specific data types inside them.

**Literal Types:**

Literal types means that the type of the variable is that actual data present in it.

Const x : 5 = 5;

The type of x is 5 and its value must be 5 but not any other value.

Literal types are not just types but - the values inside themselves too.

On it’s own that’s not super helpful. But combine it with something like unions and you can have very fine tuned type options for typescript to enforce.

Ex:

Const giveAnswer = (answer : “yes” | “no” | “may be”) => {

return `The answer is ${answer}`

}

giveAnswer(“no”); // Valid

giveAnswer(5); // Invalid

giveAnswer(“don’t know”); // Invalid

**Tuples:**

Tuples are a special type exclusive to TypeScript (they don’t exist in JS).

Tuples are arrays of fixed lengths and ordered with specific types – like super rigid arrays.

Ex: let mytuple : [number, string];

mytuple = [1, “psh”]; // Valid

mytuple = [“psh”, 1]; // Invalid

mytuple [0] = 2; // Invalid because tuples are immutable

But you can push new elements or pop the existing elements from the tuple.

**Enums:**

Enums allows us to define a set of **named constants.** We can give these constants numeric or string values.

Initially the value of enums starts with 0 and all the next elements will have a value of incremented by 1.

There’s quite a few options when it comes to enums.

The elements maybe uppercase or lowercase.

Ex: // Numeric Enums

enum responses {

no, // value is 0

yes, // value is 1

maybe, // value is 2

}

enum responses {

NO = 2, // value is 2

YES, // value is 3

MAYBE, // value is 4

}

enum responses {

no = 2, // value is 2

yes = 10, // value is 10

maybe = 24, // value is 24

}

// String enums

enum responses {

no = ‘no’,

yes = ‘yes’,

maybe = ‘maybe’,

}

// Heterogeneous enums

enum responses {

no = 0, // value is 0

yes = 1, // value is 1

maybe = “maybe”, // value is “maybe”

}

**NOTE ON ENUMS:**

**TS:**

enum responses {

no, // value is 0

yes, // value is 1

maybe, // value is 2

}

If we compile the above TS code then we will get the below code in our JS file.

**JS:**

"use strict";  
var responses;  
(function (responses) {  
    responses[responses["no"] = 0] = "no";  
    responses[responses["yes"] = 1] = "yes";  
    responses[responses["maybe"] = 2] = "maybe";  
})(responses || (responses = {}));

The above code looks a bit complicated. So to avoid all that complication we can use “ const “ keyword before the enum and re-write the same code as :

**TS:**

constenum responses {

no, // value is 0

yes, // value is 1

maybe, // value is 2

}

Console.log(responses.yes);

When we compile the above TS code then it will result in the below code in out JS file:

**JS:**

"use strict";  
console.log(1 /\* response.yes \*/);

The above code looks a lot cleaner and the value of that particular element in the enum is represented directly.

**Interfaces:**

Interfaces serves almost the same purpose as type aliases (with a slightly different syntax).

We can use them to create reusable, modular types that describe the **shape of the objects**.

Ex:

interface Person {

name : string;

age : number;

};

Const sayHappyBirthday = (person : Person) => {

Return `hey ${person.name{, congrats on turning ${person.age{);

}

sayHappyBirthday({name : “Prashanth”, age: 22});

} ]