# Type Inference in TypeScript

TypeScript is a typed language. However, it is not mandatory to specify the type of a variable. TypeScript infers types of variables when there is no explicit information available in the form of type annotations.

Types are inferred by TypeScript compiler when:

* Variables are initialized
* Default values are set for parameters
* Function return types are determined

var a = "some text"; //compiler infers as string annotation

var b = 123; //compiler infers as number annotation

a = b; // Compiler Error: Type 'number' is not assignable to type 'string'

## **Type inference in complex objects**

There may be scenarios where an object may be initialized with multiple types.

For example:

var arr = [ 10, null, 30, 40 ];

In the above example, we have an array that has the values 10, null, 30, and, 40 . TypeScript looks for the most common type to infer the type of the object. In this case, it picks the one thats is compatible with all types i.e. number, as well as null.

Consider another example:

var arr = [0, 1, "test"];

Here, the array has values of type number as well as type string. In such cases, the TypeScript compiler looks for the most common type to infer the type of the object but does not find any super type that can encompass all the types present in the array. In such cases, the compiler treats the type as a union of all types present in the array. Here, the type would be (string | number) which means that the array can hold either string values or number values. This is called [union type](https://www.tutorialsteacher.com/typescript/typescript-union).

Lets try to add a new element to the array:

var arr = [0, 1, "test"];

arr.push("str")

The compiler accepts the new value since the new value is of type string which is okay.

Now, lets try to add a new type to the array which was not already a part of the array:

var arr = [0, 1, "test"];

arr.push("str") // OK

arr.push(true); // Compiler Error: Argument of type 'true' is not assignable to parameter of type 'string | number'

The above code will show a compiler error because boolean is not a part of union (string | number).

The return type of a function is also inferred by the returning value. For example:

function sum(a: number, b: number )

{

return a + b;

}

var total: number = sum(10,20); // OK

var str: string = sum(10,20); // Compiler Error

In the above function, return type of the function sum is number. So, the result can be stored in a number type variable but not a string type variable.

Thus, type inference is helpful in type-checking when there are no explicit type annotations available.

Type Assertion in TypeScript

It is similar to type casting.

how TypeScript infers and checks the type of a variable using some internal logic mechanism called Type Assertion.

Type assertion allows you to set the type of a value and tell the compiler not to infer it.

Example:

let code: any = 123;

let employeeCode = <number> code;

console.log(typeof(employeeCode)); //Output: number

conclusion:

we know that code is of type number, even though it has been declared as 'any'. So, while assigning code to employeeCode, we have asserted that code is of type number

Example:Object

let employee = { };

employee.name = "John"; //Compiler Error: Property 'name' does not exist on type '{}'

employee.code = 123; //Compiler Error: Property 'code' does not exist on type '{}'

conclusion:

compiler error, because the compiler assumes that the type of employee is {} with no properties. But, we can avoid this situation by using type assertion as below

interface Employee {

name: string;

code: number;

}

let employee = <Employee> { };

employee.name = "John"; // OK

employee.code = 123; // OK

Interfaces are used to define the structure of variables.

Be careful while using type assertion. The TypeScript compiler will autocomplete Employee properties, but it won't show any compile time error if you forgot to add the properties. For example:

interface Employee {

name: string;

code: number;

}

let employee = <Employee> {

// Compiler will provide autocomplete properties,

but will not give an error if you forgot to add the properties

};

console.log(employee.name); // undefined;

There are two ways to do type assertion in TypeScript:

1. Using the angular bracket <> syntax. So far in this section, we have used angular brackets to show type assertion.

let code: any = 123;

let employeeCode = <number> code;

However, there is another way to do type assertion, using the 'as' syntax.

2. Using as keyword

let code: any = 123;

let employeeCode = code as number;

Both the syntaxes are equivalent and we can use any of these type assertions syntaxes

# TypeScript - Interfaces

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

The TypeScript compiler does not convert interface to JavaScript. It uses interface for type checking. This is also known as "duck typing" or "structural subtyping".

An interface is defined with the keyword interface and it can include properties and method declarations using a function or an [arrow function](https://www.tutorialsteacher.com/typescript/arrow-function).

interface IEmployee {

empCode: number;

empName: string;

getSalary: (number) => number; // arrow function here after arrow is return type

getManagerName(number): string;

}

This means that any object of type IEmployee must define the two properties and two methods.

## **Interface as Type**

Interface in TypeScript can be used to define a type and also to implement it in the class.

Example: Interface as Type

interface KeyPair {

key: number;

value: string;

}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK

let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in type 'KeyPair'

let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:

conclusion:

it must follow the same structure as KeyPair. It means only an object with properties key of number type and value of string type can be assigned to a variable kv1. The TypeScript compiler will show an error if there is any change in the name of the properties or the data type is different than KeyPair.

Thus, TypeScript uses an interface to ensure the proper structure of an object.

## **Interface as Function Type**

TypeScript interface is also used to define a type of a function. This ensures the function signature.

interface KeyValueProcessor

{

(key: number, value: string): void;

};

function addKeyValue(key:number, value:string):void {

console.log('addKeyValue: key = ' + key + ', value = ' + value)

}

function updateKeyValue(key: number, value:string):void {

console.log('updateKeyValue: key = '+ key + ', value = ' + value)

}

let kvp: KeyValueProcessor = addKeyValue;

kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill

kvp = updateKeyValue;

kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve

conlusion:

an interface KeyValueProcessor includes a method signature. This defines the function type. Now, we can define a variable of type KeyValueProcessor which can only point to functions with the same signature as defined in the KeyValueProcessor interface. So, addKeyValue or updateKeyValue function is assigned to kvp. So, kvp can be called like a function.

Trying to assign a function with a different signature will cause an error.

function delete(key:number):void {

console.log('Key deleted.')

}

## let kvp: KeyValueProcessor = delete; //Compiler Error

## **Interface for Array Type**

An interface can also define the type of an array where you can define the type of index as well as values.

Example: Type of Array

 Copy

interface NumList {

[index:number]:number

}

let numArr: NumList = [1, 2, 3];

numArr[0];

numArr[1];

interface IStringList {

[index:string]:string

}

let strArr : IStringList;

strArr["TS"] = "TypeScript";

strArr["JS"] = "JavaScript";

In the above example, interface NumList defines a type of array with index as number and value as number type. In the same way, IStringList defines a string array with index as string and value as string.

## **Optional Property**

Sometimes, we may declare an interface with excess properties but may not expect all objects to define all the given interface properties. We can have optional properties, marked with a "?". In such cases, objects of the interface may or may not define these properties.

Example: Optional Property

 Copy

interface IEmployee {

empCode: number;

empName: string;

empDept?:string;

}

let empObj1:IEmployee = { // OK

empCode:1,

empName:"Steve"

}

let empObj2:IEmployee = { // OK

empCode:1,

empName:"Bill",

empDept:"IT"

}

In the above example, empDept is marked with ?, so objects of IEmployee may or may not include this property.

## **Read only Properties**

TypeScript provides a way to mark a property as read only. This means that once a property is assigned a value, it cannot be changed!

Example: Readonly Property

 Copy

interface Citizen {

name: string;

readonly SSN: number;

}

let personObj: Citizen = { SSN: 110555444, name: 'James Bond' }

personObj.name = 'Steve Smith'; // OK

personObj.SSN = '333666888'; // Compiler Error

In the above example, the SSN property is read only. We define the personObj object of type Citizen and assign values to the two interface properties. Next, we try to change the values assigned to both the properties-name and SSN. The TypeScript compiler will show an error when we try to change the read only SSN property.

## **Extending Interfaces**

Interfaces can extend one or more interfaces. This makes writing interfaces flexible and reusable.

Example: Extend Interface

 Copy

interface IPerson {

name: string;

gender: string;

}

interface IEmployee extends IPerson {

empCode: number;

}

let empObj:IEmployee = {

empCode:1,

name:"Bill",

gender:"Male"

}

In the above example, the IEmployee interface extends the IPerson interface. So, objects of IEmployee must include all the properties and methods of the IPerson interface otherwise, the compiler will show an error.

## **Implementing an Interface**

Similar to languages like Java and C#, interfaces in TypeScript can be implemented with a Class. The Class implementing the interface needs to strictly conform to the structure of the interface.

Example: Interface Implementation

 Copy

interface IEmployee {

empCode: number;

name: string;

getSalary:(number)=>number;

}

class Employee implements IEmployee {

empCode: number;

name: string;

constructor(code: number, name: string) {

this.empCode = code;

this.name = name;

}

getSalary(empCode:number):number {

return 20000;

}

}

let emp = new Employee(1, "Steve");

In the above example, the IEmployee interface is implemented in the Employee class using the the implement keyword. The implementing class should strictly define the properties and the function with the same name and data type. If the implementing class does not follow the structure, then the compiler will show an error.

Of course, the implementing class can define extra properties and methods, but at least it must define all the members of an interface.