TypeScript - Function Overloading

TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.

Example: Function Overloading

function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {

return a + b;

}

add("Hello ", "Steve"); // returns "Hello Steve"

add(10, 20); // returns 30

In the above example, we have the same function add() with two function declarations and one function implementation. The first signature has two parameters of type string, whereas the second signature has two parameters of the type number.

The last function should have the function implementation. Since the return type can be either string or number as per the first two function declarations, we must use compatible parameters and return type as any in the function definition.

Function overloading with different number of parameters and types with same name is not supported.

Example: Function Overloading

function display(a:string, b:string):void //Compiler Error: Duplicate function implementation

{

console.log(a + b);

}

function display(a:number): void //Compiler Error: Duplicate function implementation

{

console.log(a);

}

Thus, in order to achieve function overloading, we must declare all the functions with possible signatures. Also, function implementation should have compatible types for all declarations.

# TypeScript Data Type - Tuple

TypeScript introduced a new data type called Tuple. There are other data types such as number, string, boolean etc. in TypeScript which only store a value of that particular data type. Tuple is a new data type which includes two set of values of different data types.

Consider the following example of number, string and tuple type variables.

Example: Tuple vs Other Data Types

var empId: number = 1;

var empName: string = "Steve";

// Tuple type variable

var employee: [number, string] = [1, "Steve"];

In the above example, we have defined a variable empId as number type and empName as string type with values. Here, we declared and assigned two variables to id and name of an employee. The same thing can be achieved by using a single tuple type variable. employee is the tuple type variable with two values of number and string type. Thus, removing the need to declare two different variables.

A tuple type variable can include multiple data types as shown below.

Example: Tuple

var employee: [number, string] = [1, "Steve"];

var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string];// declare tuple variable

user = [1, "Steve", true, 20, "Admin"];// initialize tuple variable

You can declare an array of tuple also.

Example: Tuple Array

var employee: [number, string][];

employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];

TypeScript generates an array in JavaScript for the tuple variable. For example, var employee: [number, string] = [1, 'Steve'] will be compiled as var employee = [1, "Steve"] in JavaScript.

## Accessing Tuple Elements

We can access tuple elements using index, the same way as an array. An index starts from zero.

Example: Accessing Tuple

var employee: [number, string] = [1, "Steve"];

employee[0]; // returns 1

employee[1]; // returns "Steve"

## Add Elements into Tuple

You can add new elements to a tuple using the push() method.

Example: push()

var employee: [number, string] = [1, "Steve"];

employee.push(2, "Bill");

console.log(employee); //Output: [1, 'Steve', 2, 'Bill']

This is allowed because we are adding number and string values to the tuple and they are valid for the employee tuple.

Now, let's try to add a boolean value to the tuple.

employee.push(true)

The above example will throw the following error:

test.ts(4,15): error TS2345:

Argument of type 'true' is not assignable to parameter of type 'number | string'.

We get an error saying that adding a boolean value to a tuple of type 'number | string' is not permitted. Hence, a tuple declared as 'number | string' can store only number and string values.

The tuple is like an array. So, we can use [array methods](https://www.tutorialsteacher.com/typescript/typescript-array#array-methods) on tuple such as pop(), concat() etc.

Example: Use Array Methods

var employee: [number, string] = [1, "Steve"];

// retrieving value by index and performing an operation

employee[1] = employee[1].concat(" Jobs");

console.log(employee); //Output: [1, 'Steve Jobs']

# TypeScript Data Type - Enum

Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enums. This is now available in TypeScript too.

In simple words, enums allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

There are three types of enums:

1. Numeric enum
2. String enum
3. Heterogeneous enum

## Numeric Enum

Numeric enums are number-based enums i.e. they store string values as numbers.

Enums can be defined using the keyword enum. Let's say we want to store a set of print media types. The corresponding enum in TypeScript would be:

Example: Numeric Enum

enum PrintMedia {

Newspaper,

Newsletter,

Magazine,

Book

}

In the above example, we have an enum named PrintMedia. The enum has four values: Newspaper, Newsletter, Magazine, and Book. Here, enum values start from zero and increment by 1 for each member. It would be represented as:

Newspaper = 0

Newsletter = 1

Magazine = 2

Book = 3

Enums are always assigned numeric values when they are stored. The first value always takes the numeric value of 0, while the other values in the enum are incremented by 1.

We also have the option to initialize the first numeric value ourselves. For example, we can write the same enum as:

enum PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

The first member, Newspaper, is initialized with the numeric value 1. The remaining members will be incremented by 1 from the numeric value of the first value. Thus, in the above example, Newsletter would be 2, Magazine would be 3 and Book would be 4.

It is not necessary to assign sequential values to Enum members. They can have any values.

enum PrintMedia {

Newspaper = 1,

Newsletter = 5,

Magazine = 5,

Book = 10

}

The enum can be used as a function parameter or return type, as shown below:

Example: Enum as Return Type

enum PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

function getMedia(mediaName: string): PrintMedia {

if ( mediaName === 'Forbes' || mediaName === 'Outlook') {

return PrintMedia.Magazine;

}

}

let mediaType: PrintMedia = getMedia('Forbes'); // returns Magazine

enum PrintMedia {

Newspaper = "NEWSPAPER",

Newsletter = "NEWSLETTER",

Magazine = "MAGAZINE",

Book = "BOOK"

}

// Access String Enum

PrintMedia.Newspaper; //returns NEWSPAPER

PrintMedia['Magazine'];//returns MAGAZINE

# TypeScript - Union

TypeScript allows us to use more than one data type for a variable or a function parameter. This is called union type.

Syntax:

(type1 | type2 | type3 | .. | typeN)

Consider the following example of union type.

Example: Union

let code: (string | number);

code = 123; // OK

code = "ABC"; // OK

code = false; // Compiler Error

let empId: string | number;

empId = 111; // OK

empId = "E111"; // OK

empId = true; // Compiler Error

In the above example, variable code is of union type, denoted using (string | number). So, you can assign a string or a number to it.

The function parameter can also be of union type, as shown below.

Example: Function Parameter as Union Type

function displayType(code: (string | number))

{

if(typeof(code) === "number")

console.log('Code is number.')

else if(typeof(code) === "string")

console.log('Code is string.')

}

displayType(123); // Output: Code is number.

displayType("ABC"); // Output: Code is string.

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

In the above example, parameter code is of union type. So, you can pass either a string value or a number value. If you pass any other type of value e.g. boolean, then the compiler will give an error.