Typescript

Functions

What is a Function?

- A **TypeScript function** is a block of code that performs a specific task or calculates a value.
- We may ***pass values to the function,*** and it may return a value.
- We define the function only once in our program, but execute or invoke it many times
- The simple way to create a function (or define a function) is to use the **function declaration or function statement**.
- A function declaration starts with a **function** keyword. It is followed by a list of parameters and then followed by statements inside curly braces {…}

Function Declaration Syntax

The Syntax for creating a function is as follows

```
function name(param1[:type], param2[:type], param3[:type]) [:returnType] {
  [statements]
}
```

```
function calcArea(width:number, height:number):number {
   let result= width * height;
   return result;
}
```

Calling a Function

- Defining or declaring a function does not execute it. We need to invoke or call or execute it.

Example: Function with Paramter and Return Types

```
let Sum = function(x: number, y: number) : number
{
    return x + y;
}

Sum(2,3); // returns 5
```

Example: Function Parameters

```
function Greet(greeting: string, name: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

JavaScript functions can accept more (or fewer) arguments than those declared in the function declaration. It will not throw any errors. If you pass fewer arguments, then the parameter that does not receive value is initialized as undefined.

Returning a value from a Function

- We use the assignment operator to capture the returned value and assign it to a variable.

```
function  power(base:number, exponent:number):number {
    let result = 1;
    for (let count = 0; count < exponent; count++) {
      result *= base;
    }
    return result;       //returning the result
 };

//Use assignment operator to capture the returned value
let result = power(10,2)
console.log(result)  //100
```

Returning Void

- The Typescript infers void as the return type for a function that doesn't return a value or a return statement that does not return anything. The return value of such a function is undefined.

```
function sayHello() {
    console.log("Hello")
}

let result=sayHello() //Data Type of result is void
console.log(result)   //undefined
```

```
function sayHello() {
  console.log("Hello")
  return
}

let result=sayHello()
console.log(result)  //undefined
```

Function with a return statement, but without an expression after it will also return void.

Functions that Never Return

- A function may not return at all. The functions with an infinite loop or function that throws an error are examples of such functions.
- TypeScript infers the return type as never.

```
//Infinite loop
// Inferred return type: never
var x = function infiniteLoop() {
  while (true) {
  }
}

// Always throws an error
// Inferred return type: never
var y=function throwError() {
 throw new Error("Some errors occurred");
}
```

Functions & Variable Scope (Local)

- A function can define a variable within its body. They become part of the function Scope. We can access that variable only within that function and not outside of it.

```
function sayHello() {
    let message = "Hello";  //This variable can be accessed only withing this function
    alert(message);
}

sayHello();

alert(message);   //Uncaught ReferenceError: message is not defined
```

Functions & Variable Scope (Outer)

- While we can access the variable defined outside the function.

```
let message = "Hello";  //This variable is defined outside and can be accessed by the function

function sayHello() {
    alert(message);   //Hello
    message="Hello Again"
}

sayHello();

alert(message);   //Hello Again
```

- In this example, we create the variable message inside the function. It will override the variable present outside the function

Functions & Variable Scope (Outer)

```
let message = "Hello";

function sayHello() {
    //declare a message inside the function.
    let message="Hello from function"
    alert(message);   //"Hello from function"
}

sayHello();

alert(message);   //Hello    No change here
```

Functions are objects

- Functions are objects in TypeScript. Functions are a special type of object that also has a code. We can execute that code whenever we want. To execute all we need to use the () Operator after the function name. Inside the () we can pass a value for each parameter.
- The function name without the () refers to the function object.

```
function sayHello() {
    console.log("Hello")
}
```

- For example, sayHello (without()) in the following code refers to the function object. To Execute the function we need to use the parentheses sayHello().

We can attach a property to functions

- Since functions are objects, we can attach properties and methods to them. In the following example, we are adding a property c to the function addNum. We can refer to that property using addNum.c

```
function addNum(a:number, b:number) {
    console.log(addNum.c)
    return a + b + addNum.c
}

addNum.c = 100

let result = addNum(1, 2)
console.log(result) //103
```

We can store functions in a variable

- We can store functions in a variable, object, or array. This example creates **addNum** function and stores it in **test** variable. Now we can invoke the function using **test()** also.

```
function addNum(a:number,b:number) {
   return a+b
}

let test = addNum

let result = test(1,2)
console.log(result)  //3
```

We can store functions in a variable

- We can read the function just like any other variable. The following code uses the alert function to display the contents of the **addNum**.

```
function addNum(a:number, b:number) {
   return a + b
}

alert(addNum)
```

Pass functions as arguments to another function

- Since functions are variables, we can pass them as an argument to another function. You can also return a function from a function.

```
function addNum(a:number, b:number) {
   return a + b
}

function multiplyNum(a:number, b:number) {
   return a * b
}


function operate(func:any, a:number, b:number):number {
   return func(a, b)
}

let result = operate(addNum, 10, 10)
console.log(result);  //20

result = operate(multiplyNum, 10, 10)
console.log(result); //100
```

Functions as object methods

- A function declared inside an object or class is known as a method (or object method).

```
let person = {
    name : "Alex",

    addNum : function(a:number,b:number) {
      console.log(a+b)
    }
}

person.addNum(10,10)
```

```
let person = {
    name : "Alex",

    addNum : function(a:number,b:number) {
      console.log(a+b)
    }
}

let addNum1=person.addNum;

addNum1(1,10)  //11
```

we assign addNum to a variable addNum1. Since the functions are objects, they are copied by reference. Hence both addNum and addNum1 points to the same function

Function Hoisting

- We have invoked the **calArea** function before its declaration. But this code works without any errors. This is because of Hoisting.

```
//Invoke the function
let result = calcArea(10,10)
console.log(result)


//Declare the function
function calcArea (width:number, height:number) {
    let result = width * height;
    return result;
};
```

Function Overloading / Method Overloading

- Function overloading (or Method Overloading) is a feature where two or more functions can have the same name but with different parameters and implementations.
- The function overloading is not available in JavaScript. But Typescript does allow us to create several overload signatures of a function.

Function Overloading / Method Overloading

- But, we cannot use both the functions either in JavaScript or in TypeScript.
- *JavaScript does allow us to create multiple functions with the same name. But it **overrides** the previously written function*. It always calls the last declared function irrespective of how many functions you have declared.
- Typescript does not allow us to same name for multiple functions. It throws a **Duplicate function implementation** error

```
function add(a:number, b:number):number {
    return a+b;
}
console.log(add(10,10))   //20
```

```
function add(numArray:number[]):number {
    var sum = numArray.reduce(function(a, b){
        return a + b;
    }, 0);
    return sum;
}

console.log(add([1,2,3,4,5,5]))   //20
```

Function Overloading / Method Overloading

- But in JavaScript where we do not have overloading, we have two options.
- One is to create separate functions for each variation in arguments like addNum and addNumArray.
- The second option is to create a single function and check the data type of the arguments passed and call the appropriate code within that function. This is the preferred way in JavaScript.

Creating overloaded functions

- Since Typescript is based on JavaScript, it does not allow us to create Overloaded functions with the same name. Instead, it allows us to **create overload signatures for a regular function**.
- To create function overloading first, we need to create a regular function. This function must check the type of the arguments passed to it and call the appropriate code within that function. We call this an **implementation function**.
- Next, we create overload signatures for that function and use those signatures to call the function.

Overload Signatures

- Overload signature defines how we intend to call the function.
- It contains only the signatures and does not contain any code. It is basically a function declaration without any code.
- We would like to call our add function in two ways. One is to pass two numbers and the other way is to pass an array of numbers.

```
add(10,10)          //Pass two numbers to the add function

add([1,2,3,4,5,6])   //Pass array of numbers to the add function
```

Overload Signatures

- Hence create a function declaration statement for the above, without the function body. This is our overload signatures.

```
function add(num1:number,num2:number):number
function add(numArray:number[]):number
```

Implementation Function

- The implementation function is a function that actually contains the code. Its signature must be generic enough to cover all the overloads.

```
function add(arg1:number|number[], arg2?:number):number {

   if (Array.isArray(arg1)) {

      var sum = arg1.reduce(function(a, b){
         return a + b;
      }, 0);
      return sum;
   }

   if (typeof arg1=="number" && typeof arg2=="number") {
      //number
      return arg1 + arg2;
   }

   return 0;  //default is zero.

}
```

Implementation Function

- Finally, we place the overload signatures directly above the implementation function.

```
function add(num1:number,num2:number):number
function add(numArray:number[]):number

function add(arg1:number|number[], arg2?:number):number {

    if (Array.isArray(arg1)) {

        var sum = arg1.reduce(function(a, b){
            return a + b;
        }, 0);
        return sum;
    }

    if (typeof arg1=="number" && typeof arg2=="number") {
        //number
        return arg1 + arg2;
    }

    return 0;

}
```

Implementation Function

- Now, you can call the function using the overload signatures.

```
add(10,10)                  //20
add([1,2,3,4,5,6])          //21
```

- Trying to call the function with any other way will result in a compiler error

```
//Calling with a array.
add(['a','b',1,2])   //No overload matches this call

add(1,2,3)           //Expected 1-2 arguments, but got 3.

//String and a number
add("1",2)           //No overload matches this call.
```

The number of Arguments

- The overload signatures can have any number of arguments. But the number of arguments in the implementation function must be equal (or greater) to the overload signature with the most number of arguments.

```
function odFunc(a:string):string
function odFunc(a:number,b:string, c:string):string

function odFunc(a:any,b?:any,c?:any):string {
    return ""
}
```

- We need to mark some arguments as optional if any of the overload signatures do not have the same number of arguments as in the implementation function.

Optional Arguments

```
function odFunc(a:string):string
function odFunc(a:number,b?:string,c?:string):string

function odFunc(a:any,b:any,c:any):string {
   return a+b+c;
}

//This overload signature is not compatible with its implementation signature.
```

- If we do not mark the argument as optional, then the first overload signature which has only one argument will not be compatible with the implementation function which has three arguments. It will result in "**This overload signature is not compatible with its implementation signature**" error.

The arguments data type must be compatible

- The data type arguments in the implementation function must be compatible with the data type in the overload signature

```
function odFunc(a:string,b:string):string
function odFunc(a:number,b:string):string

function odFunc(a:string|number,b:string):string {
   return a+b;
}

//This is also ok
//function odFunc(a:any,b:string):string {
//    return a+b;
//}

//Even this works
//function odFunc(a:unknown,b:string):string {
//    return a+b;
//}
```

The arguments data type must be compatible

- Using an incompatible type will result in “**This overload signature is not compatible with its implementation signature**” compiler error.

```
function odFunc(a:string,b:string):string
function odFunc(a:number,b:string):string

function odFunc(a:string,b:string):string {
    return a+b;
}

//This overload signature is not compatible with its implementation signature.
```

The implementation function should be the last

- The overload signatures must be placed directly above the implementation function. The following code is invalid, because of the let statement placed between the overload signature and the implementation function. Typescript compiler throws the "**Function implementation is missing or not immediately following the declaration**" error

```
function odFunc(a:string):string
function odFunc(a:number,b:string):string

let a =10

function odFunc(a:any,b?:any):string {
    return ""
}


//Function implementation is missing or not immediately following the declaration.(2391)
```

The implementation function should be the last

```
function odFunc(a:any,b?:any):string {
    return ""
}
function odFunc(a:string):string
function odFunc(a:number,b:string):string

//Function implementation is missing or not immediately following the declaration.(2391)
```

Return types can be different

- The return type of the overload signatures can be different.

```
function odFunc(str:string):string
function odFunc(num:number):number

function odFunc(arg:string|number):string|number {
    if (typeof arg=="string") return ""

    return 0
}
```

Implementation Signature is not callable

- This code does not throw any compiler error, because we are using any type in the function argument

```
function odFunc(arg:any):any {
    if (typeof arg=="string") return ""

    return 0
}

odFunc([1,2,3])    // No Compiler Error
```

Implementation Signature is not callable

- But adding an overload signature will result in "**No overload matches this call**" error. This is because once a function has an overload signature declared, you cannot use the implementation function signature to call the function.

```
function odFunc(str:string):string
function odFunc(num:number):number

function odFunc(arg:any):any {
    if (typeof arg=="string") return ""

    return 0
}

odFunc([1,2,3])   //No overload matches this call
```

Order of overload signature in Important

- Order of overload signature is important. TypeScript chooses the *first matching overload* when resolving function calls. Hence, we need to put the more specific signatures before the more general signatures.

```
function fn(x: unknown): unknown;
function fn(x: number[]): number;
function fn(x:any) {
   return x;
}

//call the function with a number array and expect a number back
var x = fn([1,2,3]); //  x is unknown
```

- This is because the unknown is a more general type than a number. The number array can be assigned to an unknown and not the other way around.

Order of overload signature in Important

```
function fn(x: number[]): number;
function fn(x: unknown): unknown;
function fn(x:any) {
    return x;
}

var x = fn([1,2,3]); // x: number
```

Method overloading

- A function that is part of a class is known as Method. You can create overload signatures in the class just like we did for the standalone functions.

```
class calculator {

   result:number;

   constructor() {
      this.result=0
   }

   //overload signature
   add(num1:number,num2:number):number
   add(numArray:number[]):number


   //function implementation
   add(arg1:number|number[], arg2?:number):number {

      this.result= 0;  //default is zero.

      if (Array.isArray(arg1)) {

         var sum = arg1.reduce(function(a, b){
            return a + b;
         }, 0);
         this.result= sum;
      }

      if (typeof arg1=="number" && typeof arg2=="number") {
         //number
         this.result= arg1 + arg2;
      }

      return this.result;

   }

}

let Calc=new calculator();
Calc.add(10,10)        //20
Calc.add([1,2,3,4,5,6]) //21
```

Function Overloading Do & Donts

- **Don't** put more general overloads before more specific overloads
- **Don't** write several overloads that differ only in trailing parameters:
- **Don't** write overloads that differ by type in only one argument position:
- **Do** sort overloads by putting the more general signatures after more specific signatures
- **Do** use optional parameters whenever possible
- **Do** use union types whenever possible

Optional Parameters in TypeScript

- Optional Parameters in TypeScript are used when passing a value to a parameter of a function is optional.

```
function addNum(a, b) {
   return a + b ;
}

//0 Argument.   Both a & b is initialzed as undefined
console.log(addNum());          //Nan

//1 Argument    b is initialzed as undefined
console.log(addNum(1));         //Nan

//2 Argumnets
console.log(addNum(1,2));       //3

//3 Arguments last argument 3 is ignored
console.log(addNum(1,2,3));     //3
```

JavaScript functions allow us the pass an arbitrary number of arguments to a function. It does not throw any errors.

Optional Parameters in TypeScript

- TypeScript results in a Compiler error

```
function addNum(a:number, b:number):number {
   return a + b ;
}

console.log(addNum());     //Expected 2 arguments, but got 0.
console.log(addNum(1));    //Expected 2 arguments, but got 1.

console.log(addNum(1,2));  //ok. 3

console.log(addNum(1,2,3));  //Expected 2 arguments, but got 3.
```

- There are two issues here
  1. Passing fewer arguments than the number of parameters
  2. Passing more arguments than the number of parameters

Optional Parameters in TypeScript

- We use the **optional parameters** to solve the first issue and the **rest parameters** for the second use case

```
function addNumber(a: number, b: number, c?: number): number {

    if (typeof c !== 'undefined') {
        return a + b + c;
    }
    return a + b;
}

addNumber(1,2,3) //6
addNumber(1,2)   //3        //No Error
```

Optional Parameters in TypeScript

- **If we do not pass any value to an optional parameter, then its value is set to undefined**. That is why we need to check the value of an optional parameter before using it. TypeScript compiler also throws the error **Object is possibly 'undefined'** if we use it before checking for **undefined**.

```
function addNumber(a: number, b: number, c?: number): number {

    return a + b+ c;   //Object is possibly 'undefined'.
}

addNumber(1,2,3) //6
addNumber(1,2)   //3
```

Optional Parameters in TypeScript

- **We must declare the optional parameters after the required parameters in the parameter list.** The optional parameter before the required parameter will result in an error.
- The compiler throws ***A required parameter cannot follow an optional parameter*** error.

```
function addNumber(a: number, b?: number, c: number): number {

    if (typeof b !== 'undefined') {
        return a + b + c;
    }
    return a + c;
}

//A required parameter cannot follow an optional parameter.
```

Strict Null Checks

- If Settings of Strict Null Checks is set to true, then TypeScript converts the type of optional parameter to a union type.
- For example, in the following code data type of c becomes number|undefined. Because of this, the following code throws an error.

```
function addNumber(a: number, b: number, c?: number): number {
    return a + b + c;          //Object is possibly 'undefined'.
}
```

- But if we set Strict Null Checks is set to false, the type of c stays as a number. Due to this, the above code will not throw any error

Optional Parameters Vs Undefined

- Instead of an optional parameter, we can assign a union type with undefined to the optional parameter.

```
function addNumber(a: number, b: number, c: number|undefined): number {

    if (typeof c !== 'undefined') {
        return a + b + c;
    }
    return a + b;
}

addNumber(1,2,3)          //6
addNumber(1,2,undefined)   //3

//Error. Third parameter is required
addNumber(1,2)          //Expected 3 arguments, but got 2.
```

This works very similarly to optional parameters except for the fact that we cannot omit the parameter.

On the other hand, we can make the parameter in any position undefined. While optional parameters must appear after the required parameters.

Default Parameters

- The default parameters allow us to initialize the parameter with default values if no value or undefined is passed as the argument.

```
function fnName(param1:type = defaultValue1, /* ... ,*/ paramN:type = defaultValueN):type {
  // ...
}
```

```
function addNum(a:number, b:number, c:number=0):number {
    let result=a+b+c
    console.log(result)
    return result ;
}

addNum(1,2) //3
```

Default Parameters

- The below code without the default value for c will result in a compiler error.

```
function addNum(a:number, b:number, c:number):number {
    let result=a+b+c
    console.log(result)
    return result ;
}

addNum(1,2)  //Expected 3 arguments, but got 2.
```

Multiple Default Parameters

- A function can have multiple default parameters.

```
function addNum(a=1, b=1) {

    let result=a+b
    console.log(result)
    return result ;
}

addNum()    //2
addNum(1)   //2
addNum(1,2) //3
```

Using Expressions as Default Values

- we can use also use any expression as a default value. In the example below b takes an expression.

```
let x=1
let y=2

function addNum(a=x, b=x+y) {
   return a + b ;
}

console.log(addNum());          //4
console.log(addNum(1));         //4
console.log(addNum(1,2));       //3
```

Evaluated at call time

- The JavaScript evaluates the parameters and updates the default values when we call the function.

```
let x=0
let y=0

function addNum(a=x, b=x+y) {
   return a + b ;
}

x=1
y=2
console.log(addNum());          //4  (a=1 & b=3)

x=5
y=5
console.log(addNum());          //15   (a=5 & b=10)
```

Using earlier Parameters

- we can also make use of earlier parameters in the default value expressions.

```
function addNum(a=1, b=a+1) {
   return a + b ;
}

console.log(addNum());          //3 (a=1 , b=2)
console.log(addNum(3));          //7 (a=3 , b=4)
```

Passing undefined

- we cannot pass undefined if we use default values. Because, If the value is undefined then TypeScript assigns the default value.

```
function addNum(a=1) {
    console.log(a)
}

addNum()          //1
addNum(undefined) //1        //gets the default value.
```

Hence if you want to pass an undefined value, then do not use default values

Default Parameters can appear anywhere

- The default Parameters can appear anywhere in the parameter list.

```
function addNum(a=1, b=2, c) {
   return a + b +c  ;
}

console.log(addNum());                  //Expected 3 got 0
console.log(addNum(3));                 //Expected 3 got 1
console.log(addNum(3,6));               //Expected 3 got 2
console.log(addNum(3,6,10));                  //19  (a=3 , b=6, c=10)
console.log(addNum(undefined,undefined,10));  //13  (a=3 , b=6, c=10)
```

Function as default Value

- You can also use another function as the Default Value.

```
function showNumber(a=getNumber()) {
    console.log(a)
}

function getNumber() {
    return 10;
}

showNumber()  //10
showNumber(5) //5
```

Making Parameter Optional

- We can make the parameter optional using the ?. But that would result in NaN.

```
function addNum(a:number, b:number, c?:number):number {
    let result=a+b+c
    console.log(result)
    return result ;
}

addNum(1,2) //NaN
```

- To solve this, we need to check if the c is undefined. If it is then initialize it with 0.

Making Parameter Optional

```
function addNum(a:number, b:number, c?:number):number {

    if (typeof c==="undefined") c=0

    let result=a+b+c
    console.log(result)
    return result ;
}

addNum(1,2) //3
```

Rest Parameters

- Rest Parameters in TypeScript allow us to accept a variable number of arguments as an array.

```
function addNum(a:number, b:number) {
   return a + b ;
}

//0 Argument.
console.log(addNum());          //Expected 2 arguments, but got 0.

//1 Argument
console.log(addNum(1));         //Expected 2 arguments, but got 1.

//2 Argumnets
console.log(addNum(1,2));        //Ok  3

//3 Arguments
console.log(addNum(1,2,3));      //Expected 2 arguments, but got 3.
```

Using Rest Parameters

- Rest Parameters in TypeScript lets us store the extra arguments that we supply to the function into an array.
- The syntax is shown below. We prefix the rest Parameter with … (three dots) followed by the name of the rest parameter.

```
function f(a:type, b:type, ...args:type[]) {
  // ...
}
```

- In the above syntax args is the rest parameter, while a & b are normal parameters. **We must provide an array type to the rest parameter**.
- TypeScript assigns the arguments to parameters starting from left to right.

Using Rest Parameters

```
function fnRest(a:number, b:number, ...args:number[]) {

  console.log("a", a);
  console.log("b", b);
  console.log("args", args);
}

fnRest(1,2,3,4,5);

//Output:

//a 1
//b 2
//args [ 3, 4, 5 ]          //All addtional arguments are now stored in array
```

Rules of Rest Parameters

- Even if there is just one extra argument, it will be saved as a single element of an array.

```
function fnRest(a:number, b:number, ...args:number[]) {

  console.log("a", a);
  console.log("b", b);
  console.log("args", args);
}

fnRest(1,2,3);

//Output:

//a 1
//b 2
//args [ 3 ]            //Array with single element
```

- **If there is no argument provided then we get an empty array**

Rules of Rest Parameters

- If there is no argument provided then we get an empty array

```
function fnRest(a:number, b:number, ...args:number[]) {

  console.log("a", a);
  console.log("b", b);
  console.log("args", args);
}

fnRest(1,2);

//Output:

//a 1
//b 2
//args []           //Empty Array
```

Rules of Rest Parameters

- The Rest Parameters must appear last in the Parameter list.

```
//Wrong
function f1(a:number,  ...restPara:number[], b:number):void {

}

//Wrong
function f2(...restPara:number[],a:number, b:number):void  {

}


//A rest parameter must be last in a parameter list.
```

Rules of Rest Parameters

- There can be only one rest parameter in a function.

```
//Wrong
function f(a:number, b:number, ...restPara1:number[],...restPara2:number[]):void {

}
```

Rest Parameters Example

```
function AddNum(...nums:number[]){
    let sum = 0;
    for(let i of nums){
        sum+=i;
    }
    return sum;
}

console.log(AddNum(1,2));                    // 3
console.log(AddNum(1,2,3));                  // 6
console.log(AddNum(1,2,3,4,5));              // 15
console.log(AddNum(1,2,3,4,5,6,7,8,9,10));   // 55
```

Rest Parameters Example

```
function printNames(...names: string[]): void {
    console.log("Count " + names.length)
    for (let name of names) {
        console.log(name);
    }
}

printNames("Angular","TypeScript","JavaScript");

//Count 3
//Angular
//TypeScript
//JavaScript
```

Function Types

- Function Types describe the parameter and return types of a function
- Typescript has a built-in global type Function, which is a generic type that we can use to describe all functions

```
let num:number
num=""    //Type 'string' is not assignable to type 'number'.
```

```
let sumFn;

sumFn = function(a:number, b:number) {
   return a+b;
}

sumFn=10;   //No Error here because sumFn is of type any.
```

We are assigning a function to the variable sumFn. However, we have not assigned a type to sumFn. The compiler will assigns any type to it, which means you can assign anything to it.

How do we ensure that it only stores a function?

Typescript has a data type for it and it is called **Function**

Global Type Function

- TypeScript has a global data type **Function**. You can use it to annotate a variable just like any other data type.

```
let sumFn:Function    //Function is a data type like string, object, number etc

sumFn = function(a:number, b:number) {
   return a+b;
}

sumFn=10;   //Type 'number' is not assignable to type 'Function'
```

- Now you can only assign a function to it and nothing else.
- The Function data type describes properties like bind, call, apply, etc. It is a generic type that applies to all functions. But it does not provide any information about the parameters and return type of the function.

Global Type Function

- For example, assigning type Function to sumFn prevents us from assigning anything other than function to it. But it will not stop us from assigning functions with different parameters and return types.

```
let sumFn:Function

sumFn = function(a:number, b:number) {
   return a+b;
}

sumFn = function(a:number, b:number,c:number) {
   return (a+b+c)/3;
}
```

How do we stop such assignments?

Global Type Function

- We can do that by creating a type that describes the Parameter and Return Type of a function. i.e. we create custom Type for the functions.
- There are three ways in which we can create a function type in TypeScript.
  1. Function Type Expression
  2. Function Call Signature
  3. Construct Signature

Function Type Expressions

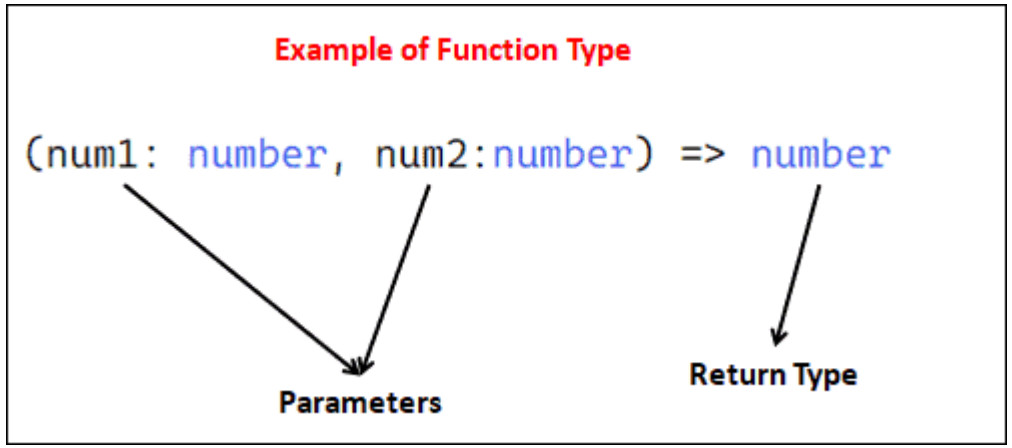
- **Function Type expressions** or **Function type literals** are the simplest way in which you can create a function type.
- It consists of two parts. List of Parameters and a Return type.
- The following is the syntax of the function type.

```
(para1: type, para2:type, ...) => type
```

- para1, para2, etc are the parameters of the function.
- We need to give names to the parameters. But the typescript compiler **ignores the names while checking if the two function types are compatible**. The parameters must be enclosed inside the bracket.
- It is followed by a => and the return type. Specifying the return type is mandatory. If the function does not return anything use void as the return type.

Function Type Expressions

```
(num1: number, num2:number) => number
```

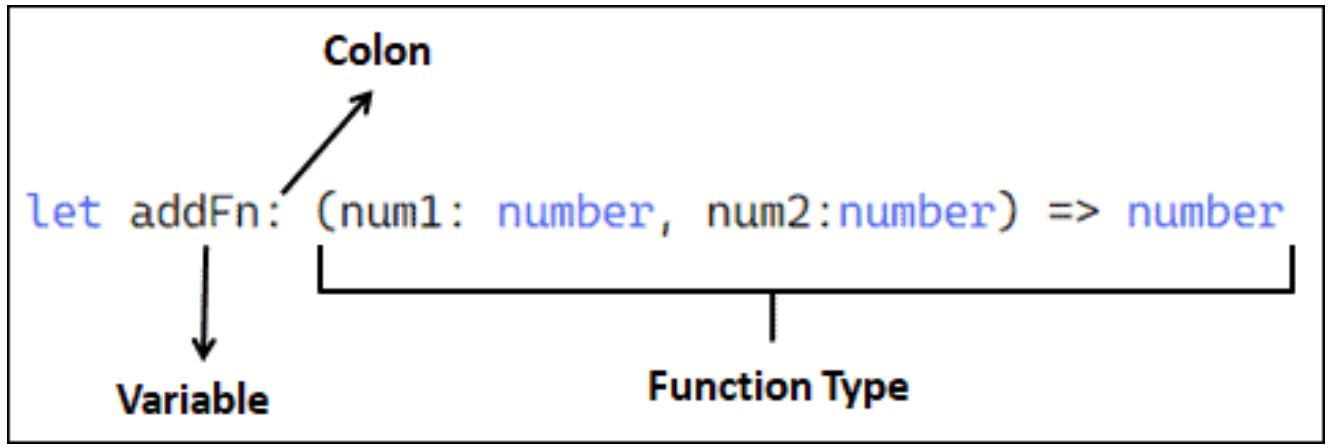


- In the following code, we assign the function type to a variable named addFn

Function Type Expressions

- In the following code, we assign the function type to a variable named addFn

```
let addFn: (num1: number, num2:number) => number
```



We can now assign any function which accepts two number parameters and returns a number to the variable addFn. The compiler will not complain as long as the function signature matches. This is because typescript follows what is known as structural typing.

Function Type Expressions

- In the following code, we have assigned a function that multiplies two numbers to addFn. **The compiler will not throw errors** because the function call signature matches the type of addFn.

```
let addFn:(num1: number, num2:number) => number

addFn= function(a:number,b:number) {
    return a+b;
}
console.log(addFn(10,10))   //20

//Assigning a function which multiplies.
//No error here since the function signature matches
addFn= function(a:number,b:number) {
    return a*b;
}
console.log(addFn(10,10))  //100
```

Function Type Expressions

- But when we assign a function with a different signature compiler throws an error. In the following example, we are trying to assign a function that accepts three-parameter to the **addFn** variable.

```
let addFn:(num1: number, num2:number) => number

addFn= function(a:number,b:number,c:number) {
    return a+b+c;
}
//Type '(a: number, b: number, c: number) => number' is not assignable to type '(num1: number, num2: number) =>
```

Function Type Expressions

- You can omit the type of parameters (a & b) when creating the function. They are automatically inferred by the compiler.

```
let addFn:(num1: number, num2:number) => number

addFn= function(a,b) {    //Types omitted.  Infered as number by the TypeScript Compiler
    return a+b;
}
```

Function Type Expressions

- You can also assign a function when declaring the variable as shown below

```
let addFn:(num1: number, num2:number) => number =
    function(a:number,b:number) {
        return a+b;
    }
```

- Again you can omit the types as they are inferred.

```
let addFn:(num1: number, num2:number) => number =
    function(a,b) {
        return a+b;
    }
```

Function Type Expressions

- Use void if the function does not return

```
let addFn:(num1: number, num2:number) => void
```

Type Inference of Function Types

- Typescript infers the type when we assign a function expression to a variable.

```
let avgFn = function getAverage(a: number, b: number, c: number): number {
    const total = a + b + c;
    const average = total / 3;
    return average;
}
```

```
    let avgFn: (a: number, b: number, c: number) => number
let avgFn = function getAverage(a: number, b: number, c: number): number {
    const total = a + b + c;
    const average = total / 3;
    return average;
}
```

Function Call Signatures

- The syntax for creating **Function Call Signatures** is very similar to the **Function Type Expression**. The only difference is that call signatures use between the parameter list and the return type while the function type expressions use =>

```
(para1: type, para2:type, ...) : type
```

- **Call Signatures are used in an object type**. Hence, we can use it to describe
  1. **Function**
  2. **Methods of an object or class**.
  3. **Functions with a property**.

Functions

- To describe a function as a property of an object first use curly brackets {} to indicate that it is an object. **Inside the curly braces { } include the function call signature.**
- Note that we cannot use Function type expressions inside the { }.

```
let addFn: {
  (num1: number, num2:number): number;     //function call signature
};

addFn = function (a,b) {
  return a+b;
}

console.log(addFn(10,10))
```

Functions

- The following codes describe the function using both function type expression and function call signature.

```
//Using function type expression to describe function
let addFn1 :  (num1: number, num2:number) => number;

//Using function call signature to describe function
let addFn2 : {
  (num1: number, num2:number): number;
};


addFn1 = function (a,b) {
  return a+b;
}

addFn2 = function (a,b) {
  return a+b;
}


addFn2 = addFn1              //Both has the same type

console.log(addFn1(10,10))     //20
console.log(addFn2(100,100))   //200
```

Functions

- We can use a Type Alias or Interface to name the function type.

```
//Type Alias
type tAddFn = {
  (num1: number, num2:number): number;
};

//Interface
interface iAddFn  {
  (num1: number, num2:number): number;
};


let addFn1:tAddFn = function (a,b) {
  return a+b;
}

let addFn2:iAddFn = function (a,b) {
  return a+b;
}


addFn2 = addFn1

console.log(addFn1(10,10))      //20
console.log(addFn2(100,100))    //200
```

Method of an object/class

- The following code shows how to describe a method when we describe an object type. We describe the method by stating the method name followed by the call signature.

```
let product : {
   id:number;
   rate:number
   calcPrice(qty:number): number      //method name following by call signature
}

//Ok
product = {
    id:1,
    rate:10,
    calcPrice(qty:number) {
        return this.rate*qty
    }
}

//Error
//Type '(qty: number, taxRate: number) => number' is not assignable to type '(qty: number) => number'.
product = {
    id:1,
    rate:10,
    calcPrice(qty:number, taxRate:number) {
        return (this.rate*qty) + (this.rate * qty *taxRate/100)
    }
}

//Error
//Type '{ id: number; rate: number; getPrice(qty: number): number; }' is not assignable to type '{ id: number; rate
//  Object literal may only specify known properties, and 'getPrice' does not exist in type '{ id: number; rate: numbe
product = {
    id:1,
    rate:10,
    getPrice(qty:number) {
        return this.rate*qty
    }
}
```

Method of an object/class

- Use an interface or type alias to assign name to the type.

```
interface IProduct  {
   id:number;
   rate:number
   calcPrice(qty:number): number
}

type TProduct = {
   id:number;
   rate:number
   calcPrice(qty:number): number
}


let prd1:IProduct = {
    id:1,
    rate:10,
    calcPrice(qty:number) {
        return this.rate*qty
    }
}

let prd2:TProduct = {
    id:1,
    rate:10,
    calcPrice(qty:number) {
        return this.rate*qty
    }
}
```

Functions with Property

- Functions are objects, hence they also can have properties. We can use the call signatures to create a type for them. The following addFn and Foo property. We use the Object.assign(target, source) to create an instance from the type.

```
let addFn : {
  (num1: number, num2:number): number;
  foo:string
};

addFn= Object.assign(
  (a:number,b:number) => {return a+b},
  {foo:''}
)

console.log(addFn(10,10))  //20
```

Construct Signatures

- We can also call the functions with **new the operator** to create a new object. Such functions are known as **constructor functions**. We can use the Construct Signatures to describe its parameter list and return type. The syntax is identical to call signatures, except that we write a new keyword in front of the call signature.

```
interface Point
{
   x: number;
   y: number;
}

interface PointConstructor
{
   new(x: number, y: number): Point;      //Construct Signature
}
```