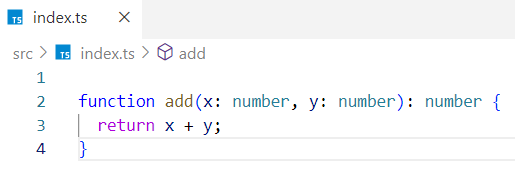
📘 CHAPTER 5: Functions in TypeScript — Parameters, Return Types, and Function Patterns

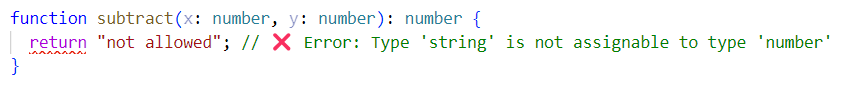
***1. Typing Function Parameters and Return Values***

Basic structure -

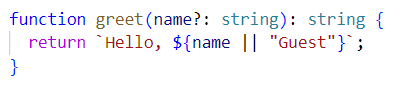


* x and y are both numbers
* : number after parentheses means the return value is a number

Example: Wrong return type -

***2. Optional Parameters***

An **optional parameter** is a parameter that may or may not be passed. It's marked with a ?, and its value will be undefined if not provided.



☑️ greet(); // "Hello, Guest"  
☑️ greet("Likan"); // "Hello, Likan"

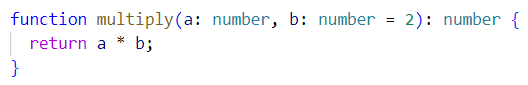
Explanation -

The function returns "Hello, [name]" if a truthy name is provided; otherwise, it defaults to "Hello, Guest".

Rule - Optional parameters must come after required ones.

***3. Default Parameters***

A **default parameter** is a function parameter that has a default value assigned. If no argument is passed, the default value is used.



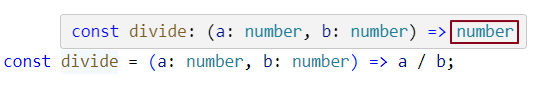
☑️ multiply(5); // 10  
☑️ multiply(5, 3); // 15

Explanation –

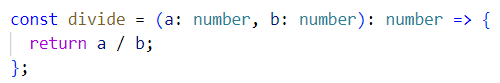
A parameter becomes a default when you assign it a value in the function definition like b = 2, which means if no value is passed for b, 2 will be used automatically. If the second argument b is not provided, it defaults to 2, so multiply(a) behaves like a \* 2.

***4. Arrow Functions***

✅ Inferred return type (TS figures it out) -



✅ Explicit return type -



***5. Function as Types (Callbacks)***

Function as types (also known as callback typing) in TypeScript.

***What Does "Function as Type" Mean?***

It means that you treat a function like a value and just like values can have types, functions can too.

You're defining the function's signature: the types of its input parameters and its return type.

Think of it like -This variable holds a function that must accept two numbers and return a number.

***1. Example: Assigning a Function to a Variable (with Type)***

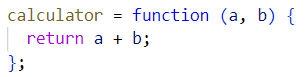
Here’s how you declare a variable that must hold a function:



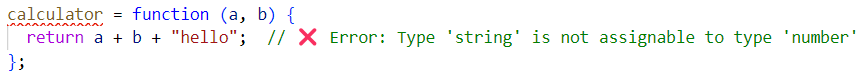
This means -

* calculator must be a function.
* It must take two number arguments.
* It must return a number.

Valid assignment -



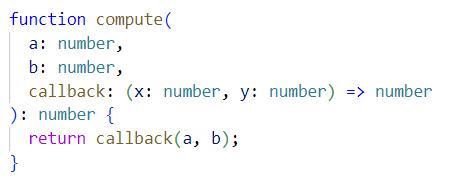
❌ Invalid assignment:



***2. Example: Passing a Function as a Callback***

This is one of the most **common and powerful** use cases - when a function receives **another function as an argument** (a callback).

Function Signature (Typed) -



**🧠** What this function does ?

This function -

* Accepts two numbers: a and b
* Accepts a callback function:
  + The callback must take two numbers
  + The callback must return a number
* Returns the result of the callback

Example Usage -



### Why Is This Useful?

* This **makes sure** the callback does exactly what it’s supposed to.
* This **avoids** mistakes when writing or calling the callback.
* This **gets** you smart suggestions from your editor (like parameter names and types).
* This **keeps** your code safe and consistent by enforcing types.

Can you pass a callback function by just its reference in TypeScript?

Yes, absolutely - you can pass the function reference directly without calling it.

### **Function Reference vs Function Call -**

#### ✅ **Passing a Function Reference** (✔️ Correct)

You pass the function **without parentheses** — this passes the function itself, not its result.



You're saying: "Here’s the function you can call later."

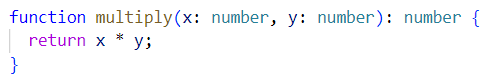
#### ❌ **Calling the Function Instead** (❌ Incorrect in this context)

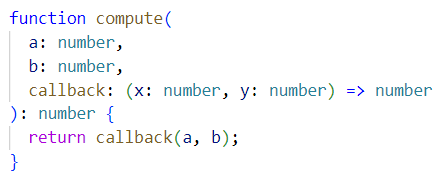
You accidentally **call** the function by adding parentheses, and pass its **return value**, not the function.



* You're passing the **number** returned by multiply(4, 2), not the function itself.
* This causes an error because compute expects a **function**, not a number.

Defining a Reusable Callback Function -



Now Define the Higher-Order Function –

✅ Correct Usage (Passing a Function Reference) -



We're telling compute -

**Here’s a function named** multiply**. You can call it using the values you already have (**a **and** b**).**

## **Callback Typing –**

### How do you type a callback function?

Syntax –

(<param1>: Type, <param2>: Type) => <ReturnType>

Example -

let callback: (x: number, y: number) => number;

***6. Function Overloads***

### **What Are Function Overloads?**

**Function overloads** allow a function to have **multiple signatures** - so it can be called in different ways, with different types or numbers of parameters — but all handled by **one single implementation.**

**What is a function signature ?**

A **function signature** defines how a function should be called — it includes the function’s name, the number and types of its parameters, and its return type. It does not include the actual implementation (what the function does).

**Real-World Analogy –**

Imagine you're building a sendMessage function. You want to allow users to:

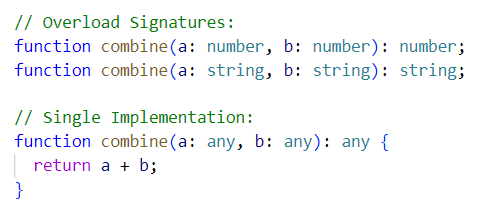
* Send a message using an email → (email: string, message: string)
* Or send a message using a user ID → (userId: number, message: string)

You want both to work — but using the **same function name** and **one internal logic**.

### **Function Overload Syntax -**

* **Overload Signatures -** define the allowed ways to call the function (no logic here)
* **One Implementation -** handles all the cases (logic goes here)

### **✅ TypeScript Example 1 -**



### Valid Calls -



### Invalid Call -



### ***Why is combine(1, "Dev") invalid even though the implementation uses function combine(a: any, b: any): any?***

Because in TypeScript -

* Only the **overload signatures** are used for **type-checking at call sites**.
* The **implementation** is ignored by the compiler when calling the function.

So, TypeScript looks at your overloads -

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

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

It does **not find** a matching signature for (number, string) → ❌ Error.

### **Fix:** Add a third overload .If you want that call to be valid, you **must explicitly add** the overload - function combine(a: number, b: string): string;

### ***Why Not Just Use the Implementation Type?***

That would defeat the entire purpose of function overloads.

Because:

* Overloads exist to **restrict how functions are used or called**
* The implementation must handle **all possible overloads.** So, the implementation type needs to be **broad** (like any or a union type) to cover all cases
* The **implementation signature is intentionally hidden** from the caller — it’s not used for type checking.

### **Key Rules of Overloading -**

1. You can write **multiple overload signatures.**
2. You must write **exactly one function implementation.**
3. That implementation should be **generic enough** (usually using any, unknown, or union types like string | number)
4. **Only overload signatures are exposed** to the caller - not the implementation’s type

### **✅ TypeScript Example 2 –**

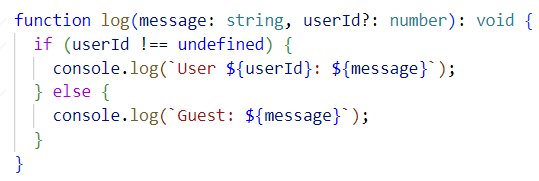
You want a log function that should work in **two ways.**

* log(message: string) → for guest users
* log(message: string, userId: number) → for logged-in users

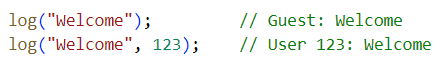
Write Overload Signatures -



One General Implementation -



Usage -



### Why This Works ?

* The function can be called with **1 or 2 parameters**
* The **single implementation** uses an optional parameter (userId?: number)
* Inside the function, it checks if userId is present using if (userId !== undefined)

**Note -** Function overloads let you define multiple call signatures for one function - with a single implementation that handles them all.

## Optional Parameters in TypeScript -

### **Rule - Optional parameters must come** after **required ones.**

### ***Invalid Example:***

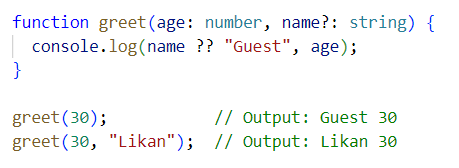
function greet(name?: string, age: number) { } // ❌ Error

### ❓ Why?

greet(30); // ❌ Ambiguous: Is 30 `name` or `age`?

TypeScript can’t determine where the argument should go if the optional comes first.

### ***Correct Example:***



### Behind the Scenes:

* TypeScript uses the **position of arguments** to assign them to parameters.
* Skipping an optional param only works **if it comes later.**
* If it’s placed before required ones → **confusion and error.**

Optional parameters go **after** required ones to prevent argument-matching confusion at the call site.

Nullish Coalescing Operator -

To provide a default value when a variable is null or undefined

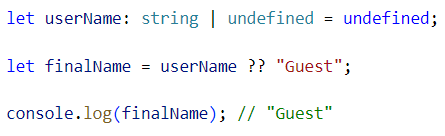
**Syntax -**

let result = value ?? defaultValue;

This means:

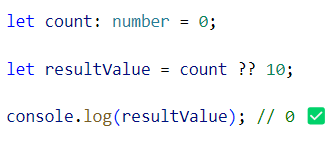
👉 If value is null or undefined → use defaultValue  
👉 Otherwise → use value itself

***Example 1 - Using ?? -***



💡 userName is undefined → so "Guest" is used as fallback

***Example 2 - Not triggered for falsy values like 0 or ""***



***Different between OR (||) operator and Nullish coalescing operator (??) -***

* **||** considers all falsy values: false, 0, "", null, undefined.
* **??** only considers: null and undefined

**Using || -**

let val = 0 || 10; // result → 10 ❌ (0 is falsy)

**Using ?? -**

let val = 0 ?? 10; // result → 0 ✅ (0 is not null/undefined)

✅ So, if you want to provide fallback only when something is null or undefined, use ??

Interview Insights

***Q: What’s the difference between optional and default parameters?***

Optional means the argument can be omitted. Default means if omitted, it gets a default value

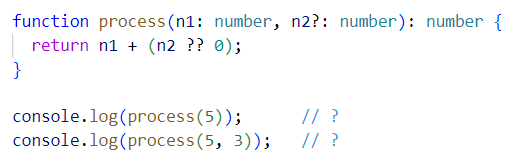
***Q: Can TypeScript infer function return types?***

Yes - but it’s best to be explicit, especially in exported/public APIs

***Q: Can arrow functions be overloaded?***

❌ No. Only function declarations can be overloaded

***Q Output based question -***



✅ Output:

5 + 0 = 5  
5 + 3 = 8

***Q: What’s the difference between overload signatures and the implementation?***

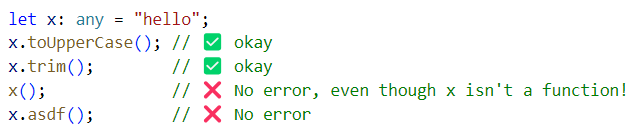
Signatures describe the valid ways to call the function (no logic) but Implementation is the actual code that handles those variations

***Q: Why should we avoid using the type any ?***

Because -

* any disables all type checks
* You can call properties or methods that don’t exist
* It removes all protection from TypeScript
* It spreads silently and creates runtime bugs

Example -



💥 So in application logic, public APIs, or shared types → avoid any.

Note - Avoid any in general, but it’s acceptable inside the implementation of overloaded functions — just don’t expose it to the caller.