# TypeScript for Automation Testers – Day 9

Topic: Functions in TypeScript – Parameter and Return Type Handling in Functions (Part 2)

---

## Understanding Functions (Quick Recall)

We already know that a **function** is a block of code designed to perform a specific task and we can reuse it multiple times simply by **calling** it.

When we call a function in TypeScript, the **program flow jumps** to that function's body, executes the code inside and then returns control back to the calling place.

TypeScript uses the keyword `function` to define a function.

---

## Why Return Types Matter

TypeScript allows us to define the **return type** of a function.
If we don't explicitly define it, TypeScript will **infer the return type automatically** based on the function's behaviour.

But it's always a **best practice to define the return type explicitly** — this makes your code safer and easier to maintain.

---

## Return Type vs Data Type (Let's Clear the Confusion)

"Aren't both the same?"

| Concept | Where It Is Used | What It Describes | Example |
|---|---|---|---|
| **Data Type** | Variables | The type of data a variable *stores* | `let name: string = "Yogi";` |
| **Return Type** | Functions | The type of data a function *returns* | `function add(): number { return 10; }` |

So when a **variable** holds a value → it has a **data type**.
When a **function** returns a value → it has a **return type**.

That's the key difference.

---

## What Is `void` and Why Only in Functions?

Now here's something new — `void`.
You've probably seen this in function definitions, but wondered what it really means.

**Meaning:**
`void` simply means **"nothing is returned."**

It tells TypeScript that this function will **not return any value**.

**Why and Where:**
We use `void` when we just want to perform an action (like clicking a button, logging a message or navigating to a page) and don't need to get any value back.

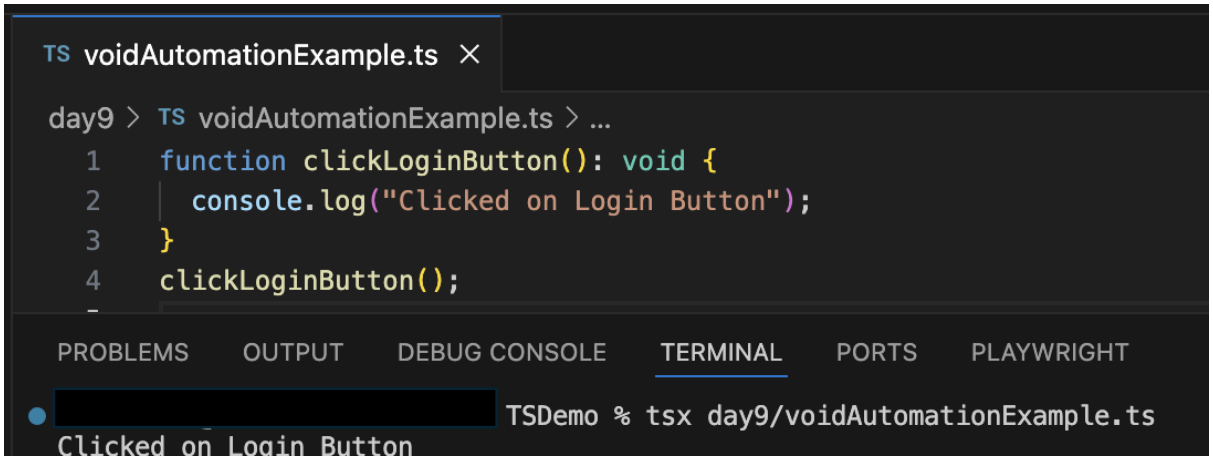
**Example 1 – Logging Function**

```
function logMessage(): void {
  console.log("Execution started...");
}
logMessage();
```

Here, the function doesn't return anything. It just performs an action — logging a message.

**Example 2 – Automation Example**
Imagine in Selenium automation we just want to click a button.
That action doesn't return a value — it just performs an operation.
So, this is also a **void function**.

```
function clickLoginButton(): void {
  console.log("Clicked on Login Button");
}
clickLoginButton();
```




So, **void = do something, but don't send anything back.**

---

## Understanding “void” — The Real Meaning

When we say a function **returns nothing** (or has a **void** return type), we mean:

The function **does not send any value back to the place where it was called**.

It **can still do work**, perform actions, print things on the console, click a button, update a UI — but it **does not return a value** that the caller can store or use in an expression.

---

### Let’s look at same example

```
function clickLoginButton(): void {
  console.log("Clicked on Login Button");
}

clickLoginButton();
```

When you call `clickLoginButton()`, yes — something **happens** (you see the message printed).
But that doesn’t mean the function *returns* a value.

Try this:

```
let result = clickLoginButton();
console.log(result);
```

**Output:**

```
Clicked on Login Button
undefined
```

That `undefined` shows that **nothing was returned**.
So technically:

- The function performed an **action** → printed a message.
- It **did not return any value** → return type is `void`.

---

## Another way to visualize it

Think of two types of people:

1. **Messenger who delivers something to you.**

That's like a function returning a value.

Example:

```
function getUserName(): string {
  return "John";
}
let name = getUserName(); // name = "John"
```

2. **Worker who just performs a task but doesn't give you anything back.**

That's like a `void` function.

Example:

```
function cleanRoom(): void {
  console.log("Room cleaned!");
}
cleanRoom(); // prints message, but gives no return value
```

You can **see** the result (because of `console.log`) but you **don't receive** it programmatically.

---

## In summary

| Concept | Meaning | Example | Return Value |
|---|---|---|---|
| Returns a value | Function sends data back to caller | `function getName():string { return "John" }` | `"John"` |
| `void` return type | Function performs action but returns nothing | `function logMessage():void { console.log("Hi") }` | `undefined` |

So, in your example,
`clickLoginButton()` gives visible output (because of `console.log`),
but programmatically, it **returns nothing** — hence `void`.

---

## Parameters – What Are They Really?

**Parameters** are like placeholders or variables that a function expects to receive.
They define what kind of data the function needs to do its work.

**Example:**

```
function greetUser(name: string) {//greetUser expects a name
  console.log(`Hello, ${name}!`);
}
greetUser("Yogi");
```


Here,

- `name` is the **parameter** — defined inside the function.
- `"Yogi"` is the **argument** — actual value passed when calling the function.

---

## Parameter vs Argument

| Term | Where It Appears | Description | Example |
|---|---|---|---|
| **Parameter** | Inside the function definition | Acts as a placeholder | `function greet(name: string)` |
| **Argument** | While calling the function | Actual data passed | `greet("Yogi")` |

Think of it like this —
Parameters are *questions* the function asks; arguments are the *answers* you give.

---

# Function Possibilities in TypeScript

Functions can work in several combinations of parameters and return types.
Let's explore all of them one by one with examples and short automation-related analogies.

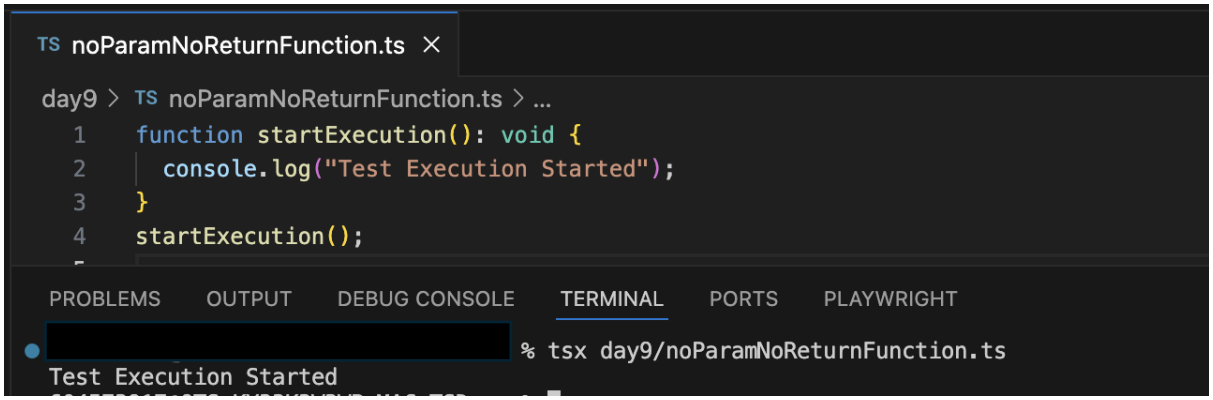
---

## 1.Function with No Parameter and No Return Type

Used when you just want to execute a block of code without needing input or output.

**Example:**

```
function startExecution(): void {
  console.log("Test Execution Started");
}
startExecution();
```




**Automation Example:**
Think of it like initializing your test environment — no input, no return value.

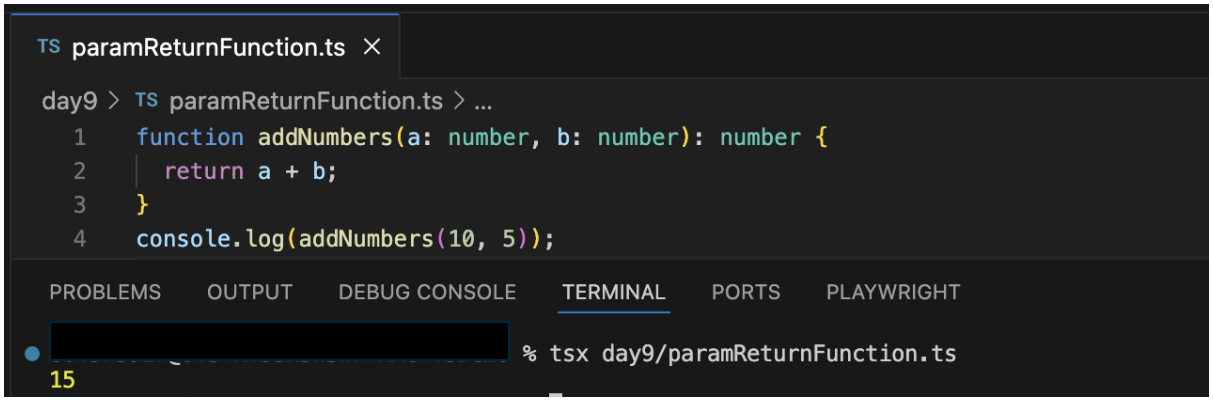
---

## 2.Function with Parameter and Return Type

Used when you want to pass data into a function and get something back.

**Example:**

```
function addNumbers(a: number, b: number): number {
  return a + b;
}
console.log(addNumbers(10, 5));
```




**Automation Example:**
A function that calculates total test cases executed and returns the count.

---

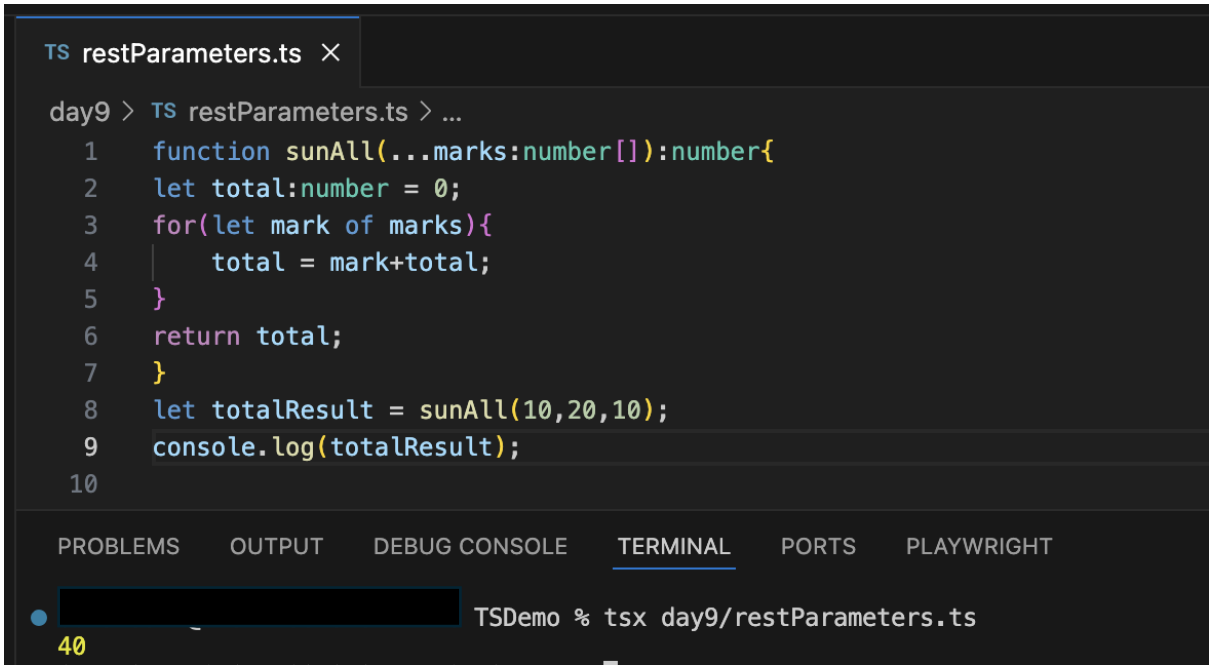
## 3.Function with Rest Parameters

What if you don't know how many inputs you'll get?
That's where **rest parameters** come in.

They allow passing multiple arguments as an array.

**Example:**

```
function sumMarks(...marks: number[]): number {
  let total = 0;
  for (let mark of marks) {
    total = total + mark;
  }
  return total;
}

console.log(sumMarks(90, 80, 70));    // Output: 240
console.log(sumMarks(100, 95));       // Output: 195 \
```



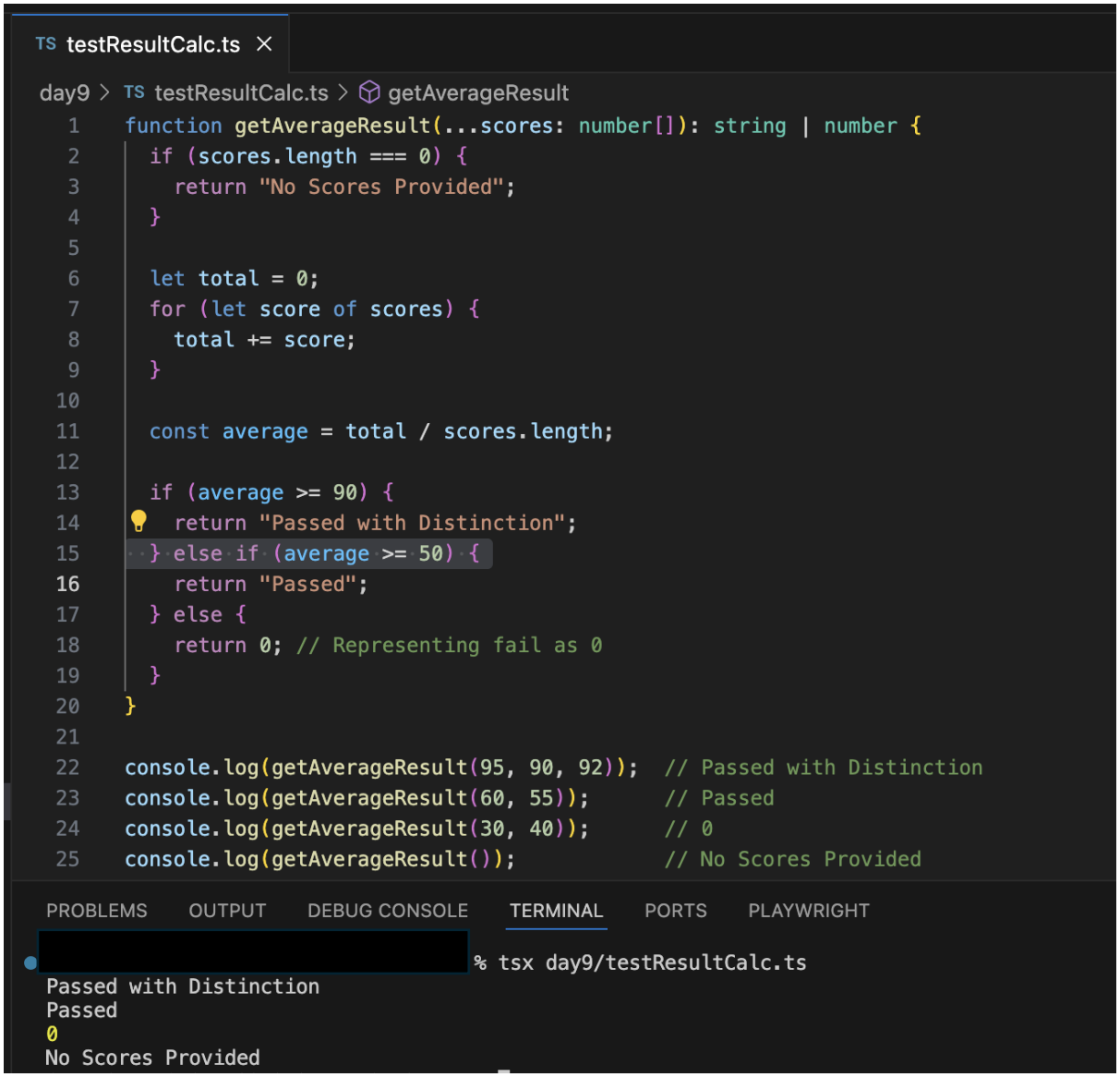

**Automation Example:**
If you want to log results for multiple browsers at once, rest parameters handle them dynamically.

---

## 4.Function with Rest Parameters and Multiple Return Types

```
// File name: testResultCalculator.ts (screenshot)
function getAverageResult(...scores: number[]): string | number {
  if (scores.length === 0) {
    return "No Scores Provided";
  }

  let total = 0;
  for (let score of scores) {
    total += score;
  }

  const average = total / scores.length;

  if (average >= 90) {
    return "Passed with Distinction";
  } else if (average >= 50) {
    return "Passed";
  } else {
    return 0; // Representing fail as 0
  }
}

console.log(getAverageResult(95, 90, 92));  // Passed with Distinction
console.log(getAverageResult(60, 55));      // Passed
console.log(getAverageResult(30, 40));      // 0
console.log(getAverageResult());            // No Scores Provided
```

---




## Let's break it down

- `...scores: number[]` → rest parameter, meaning it can take *any number* of scores.
- We calculate the **average** of all given scores.
- Based on that, the function may return:
  - A **string** (`"Passed"` or `"Passed with Distinction"`)
  - Or a **number** (`0` for fail)

Hence, the **return type** is `string | number`.

---

## Automation Example (realistic connection)

Imagine a function in an automation test framework that takes multiple response times from API test runs:

```
function evaluatePerformance(...responseTimes: number[]): string | number {
  const avg = responseTimes.reduce((a, b) => a + b, 0) /
responseTimes.length;
  return avg < 200 ? "Good Performance" : 0;
}
```

If the average response time is below 200ms → returns `"Good Performance"`,
otherwise returns `0` (indicating performance failure).

---

## Why is this useful?

Because in real test automation, sometimes:

- You might get **numeric** values (like status codes, durations),
- Sometimes you return **string** status (like "Pass" or "Fail").

Using **union return types** gives your function the flexibility to handle both cases safely.

---

## 5.Function with Optional Parameter

Sometimes not all parameters are always required.
TypeScript lets us define **optional parameters** using the `?` symbol.

**Example:**

```
function displayMessage(name: string, message?: string): void {
  console.log(`${name}, ${message || "no message provided"}`);
}
displayMessage("Yogi");
displayMessage("Yogi", "Automation Complete");
```


**Automation Example:**
You can write a log function that optionally takes a timestamp.

---

## 6.Function with Default Parameter

When a function parameter has a default value, it's automatically used if no argument is passed.

**Example:**

```
function connectToServer(env: string = "QA"): void {
  console.log(`Connecting to ${env} environment...`);
}
connectToServer();
connectToServer("Production");
```


**Automation Example:**
You can have a function that connects to the QA environment by default but can switch to Production when needed.

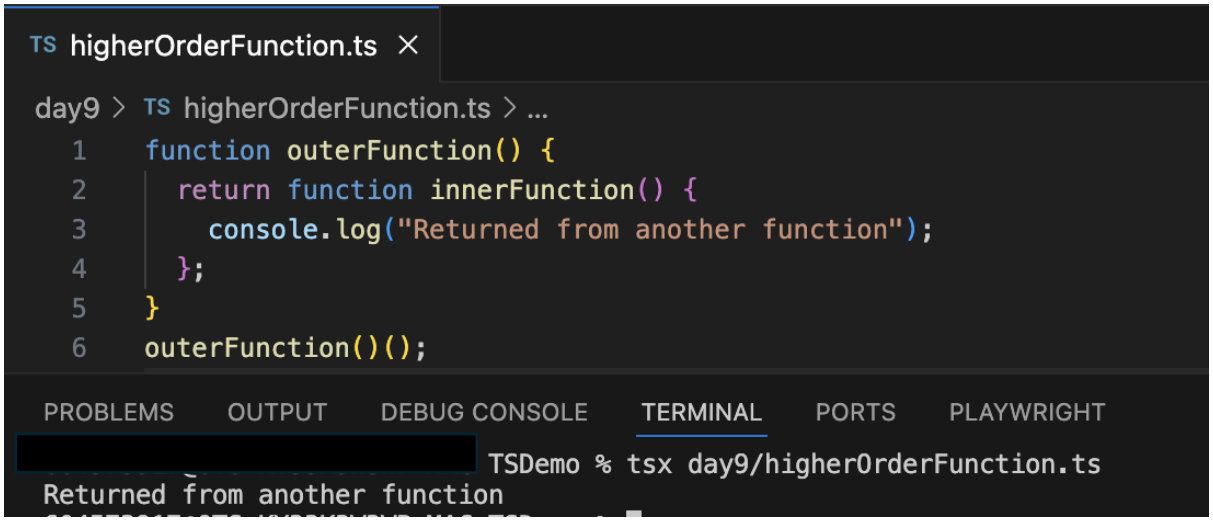
---

## Bonus: Function Returning Another Function

Sometimes, a function can **return another function** — this is called a **higher-order function**.

**Example:**

```
function outerFunction() {
  return function innerFunction() {
    console.log("Returned from another function");
  };
}
outerFunction()();
```




This concept is often used in frameworks like Playwright or Selenium to wrap reusable logic.

---

### Java vs TypeScript – Function Comparison

| Concept | Java | TypeScript |
|---|---|---|
| Function Keyword | Defined inside a class as method | Defined directly using `function` |
| Return Type | Mandatory | Optional but recommended |
| Void | Used for no return | Same usage |
| Overloading | Supported | Supported (with type annotations) |
| Access Modifiers | Required | Optional |

---

# Questions

1. What is the difference between data type and return type?
2. What does `void` mean in TypeScript functions?
3. What is the difference between parameters and arguments?
4. When do we use rest parameters?
5. How do optional parameters differ from default parameters?
6. Can a TypeScript function return multiple types of values?
7. Why should we always define a function's return type?
8. What is a higher-order function in TypeScript?

---

# Answers

1. Data type defines what type of data a variable can hold; return type defines what type of data a function returns.
2. `void` means the function doesn't return any value — it just performs an action.
3. Parameters are placeholders defined in a function; arguments are the actual values passed during a function call.
4. Rest parameters are used when the number of inputs is unknown or dynamic.
5. Optional parameters may or may not be provided, while default parameters have predefined values if not provided.
6. Yes, using union types (e.g., `string | number`), a function can return multiple types of values.
7. Defining return types makes code predictable, safer, and helps catch errors during compilation.
8. A higher-order function is a function that returns another function or accepts one as an argument.

---