

TypeScript Basics for Automation Testers – Day 4 (Part 2)

Topic: Data Types in TypeScript (Core Data Types)

What are Data Types?

- Data types decide **what type of data** can be stored in a variable.
 - In TypeScript, we must **strictly specify the datatype** to avoid unexpected errors.
 - If we don't specify the datatype, **TypeScript behaves like JavaScript** and infers the type automatically.
-

Three Terminologies in Data Types

- 1.Type
- 2.Annotation
- 3.Type Inference

Type:

→ Defines what kind of data (number, string, boolean, etc.) a variable can hold.

Example: number, string, Boolean

Annotation:

→ When we **explicitly specify the datatype** for a variable.

Example:

```
let age: number = 30;  
  
:number is called annotation.
```

Type Inference:

→ When we **don't specify a type**, TypeScript **automatically assigns** the datatype based on the value.

Example:

```
let age = 30;  
  
TypeScript infers that age is of type number.
```

Types in TypeScript

There are **two main categories** of types:

1. **Primitive Types (Built-in types)**
 2. **Non-Primitive Types (Objects)**
-

What is “Primitive”?

- “Primitive” means **basic or fundamental**.
- These are built-in data types that hold a **single simple value** (like a number or string).

Why called “Built-in”?

- Because TypeScript already provides them — you don’t need to define or import them.

What is “Non-Primitive”?

- “Non-primitive” means **complex or user-defined**.
- They can hold **multiple values** or a **collection of data**.

What is an Object ?

- Object means a **collection of key–value pairs** or **group of related data** stored together.
-

Difference between Primitive and Non-Primitive Types

Feature	Primitive Types	Non-Primitive Types
Meaning	Basic built-in data types	Complex or user-defined
Values stored	Single value	Multiple/group of values
Example	number, string, boolean	array, class, function, interface
Similar to (in Java)	int, char, boolean	collections like List, Map, Set

Primitive Types (Built-in Types)

1.Number

2.String

3.Boolean

4.Null

5.Undefined

6.Any

7.Union Type

8.Void

Number

→ Represents both **integers** and **decimals**.

Examples: 48, 3.14

```
let score: number = 48;
let pi: number = 3.14;
console.log(score, pi);
```

```
TS numberType.ts ×

day4 > TS numberType.ts > ...
1 let score: number = 48;
2 let pi: number = 3.14;
3 console.log(score, pi);
4

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

● TSDemo % tsx day4/numberType.ts
48 3.14
```

String

→ Represents **text data**.

→ Can be written in:

- Single quotes 'Hello'
- Double quotes "Hello"
- Backticks `Hello \${name}` (for variable interpolation)

```
let name: string = "Yogi";
let greeting: string = `Hello ${name}`;
console.log(greeting);
```

TS stringType.ts ×

```
day4 > TS stringType.ts > ...
1 let names: string = "Yogi";
2 let greeting: string = `Hello ${names}`;
3 console.log(greeting);
4
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

● TSDemo % tsx day4/stringType.ts
Hello Yogi

Boolean

- Represents **true** or **false** values.
- Commonly used in conditions.

```
let isLoggedIn: boolean = true;
if (isLoggedIn) {
  console.log("User logged in");
}
```

TS booleanType.ts ×

```
day4 > TS booleanType.ts > ...
1 let isLoggedIn: boolean = true;
2 if (isLoggedIn) {
3   console.log("User logged in");
4 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

C TSDemo % tsx day4/booleanType.ts
User logged in

Null

- Represents an **intentional empty value**.

```
let data: null = null;
console.log(data); // null
```

```
TS nullType.ts ×

day4 > TS nullType.ts > ...
1  let data: null = null;
2  console.log(data); // null
3

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

● SDemo % tsx day4/nullType.ts
null
```

Undefined

→ Represents a variable that is **declared but not assigned**.

```
let user;
console.log(user); // undefined
```

```
TS undefinedType.ts ×

day4 > TS undefinedType.ts > ...
1  let user;
2  console.log(user); // undefined
3

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

● SDemo % tsx day4/undefinedType.ts
undefined
```

Any

→ A **flexible type** that allows **any kind of value**.

→ Avoid using unless absolutely necessary.

```
let value: any = "Hello";
value = 10; // Allowed
value = true; // Also allowed
console.log(value);
```

```
ts anyType.ts ×

day4 > ts anyType.ts > ...
1   let value: any = "Hello";
2   value = 10; // Allowed
3   value = true; // Also allowed
4   console.log(value);

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    PLAYWRIGHT

TSDemo % tsx day4/anyType.ts
true
```

Union Type

→ Allows **multiple types** for a single variable.

```
let id: string | number = "yogi";
id = 8; // also valid
console.log(id);
```

```
ts unionType.ts ×

day4 > ts unionType.ts > ...
1   let id: string | number = "yogi";
2   id = 8; // also valid
3   console.log(id);
4

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    PLAYWRIGHT

TSDemo % tsx day4/unionType.ts
8
```

Void

→ Meaning of “void” = **empty or nothing**.

→ Used for functions that **don't return anything**.

```
function greet(): void {
  console.log("Hi");
}
greet();
```

The screenshot shows a terminal window with the following content:

```
TS voidType.ts ×  
day4 > TS voidType.ts > ...  
1   function greet(): void {  
2     console.log("Hi");  
3   }  
4   greet();  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT  
● 604573917@OTS-KY33K3W3WP-MAC TSDemo % tsx day4/voidType.ts  
Hi
```

Understanding `void` and `return` in Functions (with Real-Life Examples)

1. What Is a Function?

A **function** is like a small worker — you give it a job, it does something, and it might (or might not) give something back.

In TypeScript (and most programming languages), you tell what kind of thing it *returns* using a **return type**:

```
function functionName(): returnType {  
    // do something  
}
```

2. Two Types of Functions

There are **two broad types** of functions:

Type	Meaning	Returns something?	Example
Action Functions	Just <i>do</i> something (no need to give back result).	No	<code>void</code>
Result Functions	<i>Do something</i> and then <i>return a result</i> .	Yes	number, string, etc.

3. Understanding `void` (Action Functions)

Meaning:

`void` = “*This function does not give any output value back.*”

But it can still **perform useful actions** (like sending mail, writing a file, turning on a light, etc.).

Example 1: Turning On a Light (real-life)

Let's write it as a program:

```
function turnOnLight(): void {  
    // Imagine this is sending a signal to turn the light on  
    console.log("The light is now ON");  
}  
  
// Call it  
turnOnLight();
```

- When you call `turnOnLight()`, the room becomes bright (action happens).
- But the function doesn't *return* anything for you to use later.
You can't do:

```
let result = turnOnLight(); // result will be undefined
```

because there is no value being returned.

Use case: You want something *to happen*, not something *to be returned*.

Example 2: Packing Lunch for You

```
function packLunch(): void {  
    console.log("Packing lunch...");  
    // lunch packed successfully!  
}
```

Even though `packLunch()` doesn't return anything, it's very useful — it performs an **action**. So `void` doesn't mean “empty function” — it means “no return value.”

4. Understanding `return` (Result Functions)

Meaning:

A function can give something *back* to whoever called it using the `return` keyword.

When you write:

```
return someValue;
```

that value goes back to the caller.

Real-Life Example: Father, You, and the Shopkeeper

Let's make a fun analogy!

Situation:

- Father gives you ₹100.
- You go to the shop and ask for chocolate.
- Shopkeeper says: "No chocolates today!" and gives your ₹100 back.

So, your function will:

- Receive ₹100 (input).
- Return:
 - "No chocolate" (a string message), and
 - 100 (money back, a number).

Program Example

```
function buyChocolate(money: number): string {  
    // Shopkeeper checks if chocolate is available  
    let isChocolateAvailable = false;  
  
    if (isChocolateAvailable) {  
        // If available, he gives you chocolate and returns message  
        return "Here is your chocolate!";  
    } else {  
        // If not, he returns a message saying no chocolate  
        return "No chocolate available. Take your money back!";  
    }  
  
    // You call the function:  
    let message = buyChocolate(100);
```

```
// Output what you got back:  
console.log(message);
```

Breakdown:

- You gave ₹100 as input.
 - Function *did some logic*.
 - It **returned a string message** to you.
 - You stored that message in `message` variable.
-

Example 2: Returning a Number

Let's say you ask your calculator:

```
function add(a: number, b: number): number {  
    return a + b; // returns the result  
}  
  
let result = add(10, 20);  
console.log(result); // Output: 30
```

Here:

- Input = 10 and 20
- Output = 30
- The function returns a number.

If we change the return type to `void`, the above won't make sense — because we **expect** something back.

5. Comparing `void` VS `return`

Concept	void Function	Function with <code>return</code>
Purpose	Do some work (no output)	Do work and give back result
Returns value?	No	Yes
Action	Send email, write file, turn on light	Calculate, fetch data, give result
Code	<code>function sendMail(): void { ... }</code>	<code>function getMailCount(): number { return 5; }</code>
When to use	When effect matters	When result matters

Quick Mental Trick:

When you see `void`, say in your mind:

“This function doesn’t give anything back, but something *happens* inside.”

When you see a return type (`string`, `number`, etc.), say:

“This function will give me a result back, and I can store or use it.”

Summary in One Line:

`void` = does something useful but gives nothing back.

`return` = gives something back that you can use.

Non-Primitive Types (Just Meaning)

Type	Meaning
<code>Array</code>	Used to store multiple values in a single variable.
<code>Tuple</code>	Similar to array, but can store different types of values in a fixed order.
<code>Class</code>	A blueprint to create objects (like defining properties and methods).
<code>Function</code>	A reusable block of code that performs a specific task.
<code>Interface</code>	A structure that defines what properties or methods an object should have.

Questions:

1. What are data types and why are they important in TypeScript?
 2. What is the difference between annotation and type inference?
 3. Explain in simple terms what a “primitive type” means.
 4. What is the main difference between primitive and non-primitive types?
 5. What does the `any` type do, and why should it be avoided?
 6. What is a union type, and when do we use it?
 7. What does “void” mean in TypeScript and where is it used?
 8. Explain what an interface means in simple English.
-

Answers:

1. **Data types** define the kind of data a variable can hold. They help prevent errors and make the code predictable.
 2. **Annotation** is when we explicitly specify the type, while **type inference** lets TypeScript automatically decide the type.
 3. A **primitive type** is a basic built-in type that stores one single value like number, string, or boolean.
 4. **Primitive types** store single simple values; **non-primitive types** can store multiple or grouped values.
 5. The **any** type allows a variable to store any kind of value. It should be avoided because it removes type safety.
 6. A **union type** allows more than one type for a variable, e.g., `string | number`.
 7. **Void** means “nothing.” It’s used for functions that don’t return any value.
 8. An **interface** defines the structure (properties/methods) an object should follow, like a contract.
-