

# Overview TypeScript For Automation testing

# Objectives

1. Variable & Data types
2. Functions
3. Class
4. Tổng hợp

# Sun\* 1. Variable & Data types

## I. Biến

### A. Khai báo

Sử dụng `let` / `const` / `var`

*var là cú pháp ES5 cũ rồi -> tránh dùng*

### B. Coding convention

- CamelCase
- Không dùng từ khóa

Tiêu chí	let	const
<b>Mục đích</b>	Biến có thể thay đổi giá trị	Hằng số
<b>Khởi tạo giá trị default</b>	no must	Bắt buộc
<b>Thường dùng khi</b>	counter, flag, input value, ...	config, URL, constants
<b>Khuyến nghị dùng trong Playwright</b>	Dùng trong test logic hoặc function	Dùng cho biến hằng (URL, selector, credentials)

# Sun\* 1. Variable & Data types

## II. Data types

### A. Primitive

String

Number

Boolean

null

undefined

### B. Complex Types

Array<T> hoặc T


Tuple

any

### C. Custom Type Definitions

interface

type (alias)

Nhóm	Data type	Cú pháp	Mô tả / Ý nghĩa	Ví dụ	Ghi nhớ nhanh
 <b>Complex Types</b>	<code>Array&lt;T&gt;</code> hoặc <code>T[]</code>	<code>let arr: string[] = []</code> <code>let nums: Array&lt;number&gt;</code>	Mảng chứa các phần tử cùng kiểu dữ liệu.	<code>let fruits: string[] = ["apple", "banana"]</code>	Dùng khi muốn danh sách phần tử <b>cùng kiểu</b>
	<b>Tuple</b>	<code>let t: [string, number]</code>	Mảng có <b>số phần tử cố định</b> , mỗi phần tử có <b>kiểu riêng và được xác định trước</b> .	<code>let user: [string, number] = ["Alice", 25]</code>	Dùng khi mảng cần <b>nhiều kiểu dữ liệu khác nhau</b>
	<b>any</b>	<code>let data: any</code>	Cho phép biến nhận <b>bất kỳ kiểu nào</b> , tắt kiểm tra kiểu.	<code>let x: any = "hello"; x = 10;</code>	<b>Nên hạn chế sử dụng!</b> Chỉ dùng khi không biết rõ kiểu (ví dụ từ API ngoài)
 <b>Custom Type Definitions</b> <a href="#">&lt;doc&gt;</a>	<b>interface</b>	<code>ts interface User { name: string; age: number; }</code>	Định nghĩa <b>cấu trúc cho object</b> , cho phép mở rộng.	<code>ts let u: User = { name: "Bob", age: 30 };</code>	Dùng cho <b>object có cấu trúc rõ ràng</b> , dễ mở rộng
	<b>type (alias)</b>	<code>type ID = string   number;</code>	Tạo <b>bí danh</b> cho kiểu phức tạp (Union, Intersection...).	<code>type Point = [number, number];</code> <code>const p: Point = [0, 1];</code>	Dùng khi cần <b>union, tuple, primitive alias, function type</b>

Tính năng / Trường hợp	interface	type (type alias)
Định nghĩa object shape (interface-like)	✓ Rất trực tiếp: <code>interface User { ... }</code>	✓ Có thể: <code>type User = { ... }</code>
Khả năng mở rộng / kế thừa	✓ <code>extends</code> (interface → interface)	✓ Dùng <code>&amp;</code> (intersection): <code>type A = B &amp; C</code>
<b>Declaration merging</b> (mở rộng bằng cách khai báo thêm)	✓ Có (có thể khai báo nhiều lần)	✗ Không (khai báo trùng sẽ lỗi)
Có thể dùng cho <b>union / primitives / tuple</b>	✗ Không (chỉ object/func)	✓ Có
Dùng làm <b>implements</b> trong class	✓ <code>class C implements I {}</code>	✗ Có thể, nhưng <b>không khuyến khích</b> , thường sẽ dùng interface
Độ rõ ràng khi đọc API công khai (library/typings)	✓ Ưu tiên (nhìn giống contract)	⚖ Cũng ok, nhưng <code>type</code> đôi khi <b>phức tạp hơn</b>
Debug / error messages	✓ Thường dễ đọc hơn	⚠ Với types phức tạp (mapped / conditional) có thể dài
Khi cần <b>kết hợp nhiều kiểu</b> (union/intersection/conditional/mapped)	✗ Hạn chế	✓ Mạnh mẽ và linh hoạt

### I. Cú pháp

```
function name_func(params): return_type {  
    // body ...  
}
```

```
function async name_func(params): return_type {  
    // body ... - luôn có từ khóa await  
}
```

- Là Synchronous Functions - hàm BẮT đồng bộ
- **Bắt buộc** với các thao tác Playwright cần chờ (ví dụ: `page.click()`, `expect()`)
- **Kiểu trả về**: Luôn là `Promise<T>`

Loại tham số	Cú pháp / Ký hiệu	Hành vi / Quy tắc	Ví dụ	Trường hợp sử dụng thực tế
● Tham số bắt buộc	a: string, b: number	Bắt buộc truyền đủ các tham số khi gọi hàm	function greet(name: string, age: number) {}	Khi <b>mọi đối số luôn cần thiết</b> để logic hàm hoạt động (VD: tính toán, validate dữ liệu).
● Tham số tùy chọn	b?: number	Tham số có thể bỏ qua. Nếu không truyền, giá trị là <b>undefined</b> .	function greet(name: string, title?: string) {}	Khi tham số chỉ <b>ảnh hưởng phụ</b> , không bắt buộc VD: hiển thị tiêu đề, mô tả thêm, cấu hình.
● Tham số mặc định	b: number = 10	Nếu không truyền, sẽ dùng giá trị mặc định. Nếu truyền <b>undefined</b> , mặc định vẫn được áp dụng.	function multiply(a: number, b: number = 2) { return a * b; }	Khi muốn đảm bảo hàm có <b>hành vi mặc định ổn định</b> mà không cần ép người gọi truyền giá trị.
● Tham số rest	...rest: number[]	Gom các đối số còn lại vào <b>một mảng</b> . Chỉ có thể có <b>một rest parameter</b> , và đặt <b>cuối cùng</b> .	function sumAll(...nums: number[]) { return nums.reduce((a,b)=>a+b,0); }	Khi không biết trước <b>số lượng đối số</b> cần truyền.  VD: tính tổng, log nhiều giá trị, merge mảng.
● Kết hợp nhiều kiểu tham số	a: string, b = 5, c?: boolean, ...rest: number[]	Có thể - <b>nhưng phải đúng thứ tự</b>	function mix(a: string, b = 10, c?: boolean, ...rest: number[]) {}	Khi hàm cần <b>linh hoạt</b> , vừa có giá trị mặc định, vừa có tùy chọn, vừa có tham số chính.

Thứ tự	Loại tham số	Bắt buộc / Ghi chú
1	Bắt buộc (required)	Phải đứng <b>trước tiên</b>
2	Mặc định (default)	Có thể sau required
3	Tùy chọn (optional)	Sau default hoặc cuối cùng
4	Rest (...args)	<b>Luôn ở cuối cùng</b> , chỉ được có 1 rest parameter

- Tham số rest ( . . . ) **gom tất cả đối số còn lại** → nếu bạn đặt nó giữa các tham số khác, TypeScript **sẽ không biết phần còn lại thuộc về ai**.
- Vì vậy, nó **bắt buộc phải ở cuối danh sách**.

Tên kiểu trả về (Return Type)	Ý nghĩa	Cú pháp ví dụ	Mô tả chi tiết / Ghi nhớ
Primitive	Hàm trả về kiểu primitive	<code>function add(a: number, b: number): number { return a + b; }</code>	Hàm bắt buộc phải <b>return một số</b> .
void	Hàm <b>không trả về</b> giá trị nào	<code>function logMessage(msg: string): void { console.log(msg); }</code>	Dùng cho hàm chỉ <b>thực hiện hành động</b> , không có giá trị cần dùng tiếp.
any	Hàm có thể trả về <b>bất kỳ kiểu dữ liệu nào</b>	<code>function randomValue(): any { return Math.random() &gt; 0.5 ? "Hi" : 10; }</code>	⚠️ Tránh lạm dụng — làm mất lợi thế kiểm tra kiểu của TypeScript.
never	Hàm <b>không bao giờ hoàn thành</b> (luôn <b>throw error</b> hoặc chạy vô hạn)	<code>function throwError(msg: string): never { throw new Error(msg); }</code>	Dùng khi chắc chắn hàm <b>sẽ không return</b> — ví dụ exception, infinite loop.
object	Hàm trả về đối tượng	<code>function getUser(): object { return { name: "Alice", age: 25 }; }</code>	Khi muốn return dữ liệu dạng JSON, record,...
Array<Type>	Hàm trả về mảng	<code>function getList(): string[] { return ["A", "B", "C"]; }</code>	Dùng cho danh sách giá trị cùng kiểu.
Promise<Type>	Hàm async trả về giá trị bất đồng bộ	<code>async function fetchData(): Promise&lt;string&gt; { return "Done"; }</code>	Dùng trong <b>Playwright &amp; API call</b> — luôn đi kèm <b>await</b> .
Union Type	Hàm có thể trả về <b>nhiều kiểu khác nhau</b>	<code>function getId(): string   number { return 123; }</code>	Linh hoạt trong xử lý dữ liệu có thể khác kiểu.
Custom Type / Interface	Hàm trả về kiểu do lập trình viên định nghĩa	<code>interface User { name: string; age: number; } function getUser(): User { return { name: "Alice", age: 20 }; }</code>	Giúp mô tả chính xác cấu trúc dữ liệu trả về.

# Sun\*

## 2. Functions

### II. Arrow function

- Thường được dùng trong các hàm **callback**, **map**, **filter**
- Hay còn gọi là function type
- Ex: `const doubled = numbers.map(num => num * 2);`

### III. Use "this"

- Giữ ngữ cảnh **this** của phạm vi bao quanh, giúp tránh lỗi khi sử dụng trong Class

```
class TestData {  
  private testId = 1001;  
  
  // Arrow function giữ ngữ cảnh 'this' của class  
  public logTestId = () => {  
    console.log(`Current Test ID: ${this.testId}`);  
  };  
}  
// Nếu dùng function thông thường, 'this' có thể bị mất khi gọi hàm này.
```

## I. Access modifier

- public
- protected
- private

## II. Define

- constructor

Dùng để khởi tạo các thuộc tính cần thiết, đặc biệt là đối tượng **Page** của Playwright

- thuộc tính / readonly properties

**Ưu tiên dùng **readonly**** cho các Locators để đảm bảo chúng không bị gán lại sau khi khởi tạo.

- method (function)






## III. Actions






- extends classes






Sử dụng để tạo một **BasePage** chứa các phương thức chung (ví dụ: **gotoUrl()**, **getURL()**)

- implement interface
- **super()**

Bắt buộc gọi **super()** trong **constructor** của lớp con nếu lớp cha có **constructor**

Nhóm	Cần nhớ gì?	Áp dụng ở đâu?
 Biến & Kiểu	<code>let</code> , <code>const</code> , <code>string</code> , <code>boolean</code>	Lưu data, URL, selector
 Hàm	<code>async/await</code> , <code>return type</code> , <code>void</code>	Viết test step & helper function
 Class & Object	<code>class</code> , <code>constructor</code> (, <code>interface</code> )	Xây Page Object Model Mỗi page web là 1 page object
 Module & Import	<code>import/export</code>	Tổ chức folder, chia component
 Error & Type	<code>try/catch</code> , <code>Promise</code> , <code>Union</code>	Xử lý exception, kiểu dữ liệu API

Nhóm Kiến Thức	Cần Nhớ Gì?	Áp Dụng Trong Playwright	Ví Dụ Minh Họa
 <b>Access Modifier</b>	<code>public</code> (mặc định), <code>private</code> , <code>protected</code> — kiểm soát phạm vi truy cập thuộc tính / hàm	Ẩn locator, chỉ expose hàm hành động	<code>private username = this.page.locator('#user');</code>
 <b>Interface / Type Alias</b>	Dùng để mô tả cấu trúc dữ liệu rõ ràng, giúp kiểm tra type	Định nghĩa kiểu dữ liệu API response, config, test data	<code>interface User { name: string; email: string }</code>
 <b>Module System (import/export)</b>	Chia nhỏ code, tái sử dụng qua các file	Dùng trong Page Object, utils, constants,...	<code>import { LoginPage } from '../pages/login.page';</code>
 <b>Error Handling (try/catch)</b>	Bắt lỗi trong hàm async, tránh test crash	Xử lý lỗi khi thao tác UI hoặc API thất bại	<code>try { await page.click(btn); } catch(e) { console.log(e); }</code>
 <b>Promise &amp; Return Type</b>	<code>Promise&lt;T&gt;</code> là kết quả trả về của async function	Hầu hết các lệnh Playwright đều trả về Promise	<code>const text: string = await page.textContent('#msg');</code>

Nhóm Kiến Thức	Cần Nhớ Gì?	Áp Dụng Trong Playwright	Ví Dụ Minh Họa
 <b>Promise &amp; Return Type</b>	<code>Promise&lt;T&gt;</code> là kết quả trả về của async function	Hầu hết các lệnh Playwright đều trả về Promise	<code>const text: string = await page.textContent('#msg');</code>
 <b>Interface vs Type (mẹo phân biệt)</b>	<code>interface</code> → mô tả cấu trúc object; <code>type</code> → linh hoạt hơn (union, tuple,...)	Dùng <code>interface</code> cho data model, <code>type</code> cho custom type	<code>type Status = 'active'</code>
 <b>Async Function (Mẹo nhớ)</b>	“Async = Asynchronous, Await = Chờ kết quả”	Luôn dùng <code>await</code> trước lệnh Playwright ( <code>page.goto</code> , <code>click</code> ,...)	<code>await page.fill('#username', 'admin');</code>
 <b>Access Modifier (Mẹo nhớ)</b>	<ul style="list-style-type: none"> <li>- <code>public</code>: ai cũng gọi được</li> <li>- <code>private</code>: chỉ dùng trong class</li> <li>- <code>protected</code>: class con có thể dùng</li> </ul>	Giúp viết Page Object chuẩn, tránh code lộn xộn	<code>private</code> cho locator, <code>public</code> cho action ( <code>login()</code> )
 <b>Interface / Type (Mẹo nhớ)</b>	Interface = “Khuôn mẫu”, Type = “Định danh”	Dùng interface khi data có cấu trúc, type khi cần tùy biến	<code>interface User { name: string }, `type Role = 'Admin'</code>



- Các biến cần được khai báo và đặt tên đúng cách.
- Các kiểu dữ liệu đa dạng từ nguyên thủy đến phức tạp.
- Các hàm xử lý hành động và trả về giá trị.
- Các lớp tổ chức mã với các công cụ sửa đổi quyền truy cập.