

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
Mục đích	Biến có thể thay đổi giá trị	Hằng số
Khởi tạo giá trị default	no must	Bắt buộc
Thường dùng khi	counter, flag, input value, ...	config, URL, constants
Khuyến nghị dùng trong Playwright	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
 Complex Types	Array<T> hoặc T[]	let arr: string[] = [] let nums: Array<number>	Mảng chứa các phần tử cùng kiểu dữ liệu.	let fruits: string[] = ["apple", "banana"]	Dùng khi muốn danh sách phần tử cùng kiểu
	Tuple	let t: [string, number]	Mảng có số phần tử cố định , mỗi phần tử có kiểu riêng và được xác định trước .	let user: [string, number] = ["Alice", 25]	Dùng khi mảng cần nhiều kiểu dữ liệu khác nhau
	any	let data: any	Cho phép biến nhận bất kỳ kiểu nào , tắt kiểm tra kiểu.	let x: any = "hello"; x = 10;	Nên hạn chế sử dụng! Chỉ dùng khi không biết rõ kiểu (ví dụ từ API ngoài)
 Custom Type Definitions doc	interface	ts interface User { name: string; age: number; }	Định nghĩa cấu trúc cho object , cho phép mở rộng.	ts let u: User = { name: "Bob", age: 30 };	Dùng cho object có cấu trúc rõ ràng , dễ mở rộng
	type (alias)	type ID = string number;	Tạo bí danh cho kiểu phức tạp (Union, Intersection...).	type Point = [number, number]; const p: Point = [0, 1];	Dùng khi cần union, tuple, primitive alias, function type

Tính năng / Trường hợp	interface	type (type alias)
Định nghĩa object shape (interface-like)	✓ Rất trực tiếp: <code>interface User { ... }</code>	✓ Có thẻ: <code>type User = { ... }</code>
Khả năng mở rộng / kế thừa	✓ <code>extends</code> (interface → interface)	✓ Dùng & (intersection): <code>type A = B & C</code>
Declaration merging (mở rộng bằng cách khai báo thêm)	✓ Có (có thẻ khai báo nhiều lần)	✗ Không (khai báo trùng sẽ lỗi)
Có thẻ dùng cho union / primitives / tuple	✗ Không (chỉ object/func)	✓ Có
Dùng làm implements trong class	✓ <code>class C implements I {}</code>	✗ Có thẻ, nhưng không khuyến khích , thường sẽ dùng interface
Độ rõ ràng khi đọc API công khai (library/typings)	✓ Ưu tiên (nhìn giống contract)	⚖️ Cũng ok, nhưng type đôi khi phức tạp hơn
Debug / error messages	✓ Thường dễ đọc hơn	⚠️ Với types phức tạp (mapped / conditional) có thể dài
Khi cần kết hợp nhiều kiểu (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 BẤT 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 mọi đối số luôn cần thiết 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 undefined .	function greet(name: string, title?: string) {}	Khi tham số chỉ ảnh hưởng phụ , 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 undefined , mặc định vẫn được áp dụng.	function multiply(a: number, b: number = 2) { return a * b; }	Khi muốn đảm bảo hàm có hành vi mặc định ổn định mà không cần ép người gọi truyền giá trị.
🟣 Tham số rest	...rest: number[]	Gom các đối số còn lại vào một mảng . Chỉ có thể có một rest parameter , và đặt cuối cùng .	function sumAll(...nums: number[]) { return nums.reduce((a,b)=>a+b,0); }	Khi không biết trước số lượng đối số 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 - nhưng phải đúng thứ tự	function mix(a: string, b = 10, c?: boolean, ...rest: number[]) {}	Khi hàm cần linh hoạt , 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 trước tiên
2	Mặc định (default)	Có thể sau required
3	Tùy chọn (optional)	Sau default hoặc cuối cùng
4	Rest (...args)	Luôn ở cuối cùng , 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	function add(a: number, b: number): number { return a + b; }	Hàm bắt buộc phải return một số .
void	Hàm không trả về giá trị nào	function logMessage(msg: string): void { console.log(msg); }	Dùng cho hàm chỉ thực hiện hành động , không có giá trị cần dùng tiếp.
any	Hàm có thể trả về bất kỳ kiểu dữ liệu nào	function randomValue(): any { return Math.random() > 0.5 ? "Hi" : 10; }	⚠️ Tránh lạm dụng — làm mất lợi thế kiểm tra kiểu của TypeScript.
never	Hàm không bao giờ hoàn thành (luôn throw error hoặc chạy vô hạn)	function throwError(msg: string): never { throw new Error(msg); }	Dùng khi chắc chắn hàm sẽ không return — ví dụ exception, infinite loop.
object	Hàm trả về đối tượng	function getUser(): object { return { name: "Alice", age: 25 }; }	Khi muốn return dữ liệu dạng JSON, record,...
Array<Type>	Hàm trả về mảng	function getList(): string[] { return ["A", "B", "C"]; }	Dùng cho danh sách giá trị cùng kiểu.
Promise<Type>	Hàm async trả về giá trị bất đồng bộ	async function fetchData(): Promise<string> { return "Done"; }	Dùng trong Playwright & API call — luôn đi kèm await .
Union Type	Hàm có thể trả về nhiều kiểu khác nhau	function getId(): string number { return 123; }	Linh hoạt trong xử lý dữ liệu có thể khác kiểu.
Custom Type / Interface	Hàm trả về kiểu do lập trình viên định nghĩa	interface User { name: string; age: number; } function getUser(): User { return { name: "Alice", age: 20 }; }	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`

4. Bảng tổng hợp

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

Nhóm Kiến Thức	Cần Nhớ Gì?	Áp Dụng Trong Playwright	Ví Dụ Minh Họa
 Access Modifier	<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>
 Interface / Type Alias	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>
 Module System (import/export)	Chia nhỏ code, tái sử dụng qua các file	Dùng trong Page Object, utils, constants,...	<code>import { LoginPage } from './pages/login.page';</code>
 Error Handling (try/catch)	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>
 Promise & Return Type	<code>Promise<T></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
Promise & Return Type	Promise<T> là kết quả trả về của async function	Hầu hết các lệnh Playwright đều trả về Promise	const text: string = await page.textContent('#msg');
Interface vs Type (Mẹo phân biệt)	interface → mô tả cấu trúc object; type → linh hoạt hơn (union, tuple,...)	Dùng interface cho data model, type cho custom type	type Status = 'active'
Async Function (Mẹo nhớ)	“Async = Asynchronous, Await = Chờ kết quả”	Luôn dùng await trước lệnh Playwright (page.goto, click,...)	await page.fill('#username', 'admin');
Access Modifier (Mẹo nhớ)	- public: ai cũng gọi được - private: chỉ dùng trong class - protected: class con có thể dùng	Giúp viết Page Object chuẩn, tránh code lộn xộn	private cho locator, public cho action (login())
Interface / Type (Mẹo nhớ)	Interface = “Khuôn mẫu”, Type = “Định danh”	Dùng interface khi data có cấu trúc, type khi cần tùy biến	interface User { name: string }, `type Role = 'Admin'



- 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.