CORE TypeScript LEARNING

> A beginner-friendly guide to TypeScript, designed to build a solid foundation for UI testing (e.g. using Playwright).

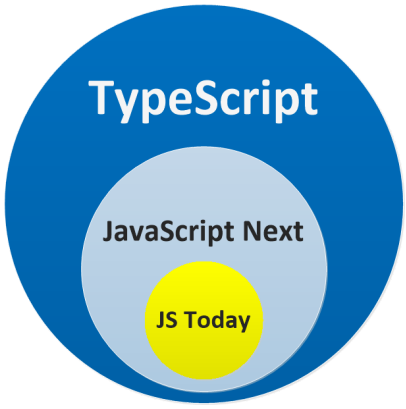

Table of Contents

---

0. Interesting links

- React Components For Creative Developers
- Github: TypeScript Coding Guideline
- TypeScript practice challenges

1. What is TypeScript?

- JavaScript is a **subset of TypeScript**. TypeScript adds *static checking* to JavaScript --> meaning that TS can detect errors in code without running it.
- It helps **catch errors early** and **makes code more readable and maintainable.**
- TypeScript *never* changes the runtime behavior of JavaScript code.
- TypeScript code is compiled into regular JavaScript to run in any browser or Node.js environment.

---

2. Getting started with TypeScript

2.1 TypeScript compiler (tsc)

- We can install TypeScript Compiler via `npm`
- Use this code to install TypeScript Compiler (tsc) globally
- Remember to install `nodejs` first

```
npm install -g typescript
```

2.2 TypeScript editor: VS Code

2.3 Types can be Explicit:

- TypeScript will infer as much as it can safely, meaning that TypeScript can guess data type based on the input
- Yet, we can also use `: <data_type>` annotation to declare data type like this:

```
var age: number = 25;
```

- So if we do something wrong the compiler will report an error like this:

```
var age: number = "25"; // Error: cannot assign a `string` to a `number`
```

---

3. Basic Types & Data Types

3.1 Hoisting

- A mechanism in JavaScript where a variable can be used before it has been declared.
- This is a behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).
- Note: it only moves the part where declaring variables, not setting values.

3.2 Three ways of Declaring Variables

a. `var`

- Scope: limited in *function-scoped*
- Hoisting is enabled: easy to cause potential error because of this. For example:

```
function test() {
  console.log(a); // undefined
  var a = 5;
}
test();
```

b. `let` **&** `const`

- Scope: limited in *block-scoped* --> exist within the block `{...}` where it is declared

```
function testLet() {
  if (true) {
    let y = 20;
  }
  console.log(y); // ❌ Error: y is not defined
}
```

- `let` allows us to change value of an existing variable but `const` does not

```
let name = "Alice";
name = "Bob"; // ☑
```

Explanation:

- `let`: declares a block-scoped variable.
- `age`: name of the variable.
- `:`: specifies the type of the variable.
- `number`: the type.
- `= 25`: assigns the value 25 to the variable.

3.3 Common Data Types

In TypeScript, we have some basic data types including:

	Type	Syntax	Example
1	number	`let x: number`	`let price = 100;`
2	string	`let s: string`	`let name = "Alice";`
3	boolean	`let b: boolean`	`let isReady = true;`
4	array	`let a: type[]`	`let items: string[] = []`
5	enum	`enum a {v1 v2}`	`enum Direction {up down}`
6	void	used to indicate a function has no value to return	`function log(): void { console.log("hi") }`

◇ `any` **type**

```
let something: any;
something = 42;
something = "Hello";
something = true;
something = { name: "Alice" };
something = [1, 2, 3];
console.log(something);
```

- `any` disables all type checking --> It **bypasses** TypeScript's static type system.
- Avoid using `any` in production code.

◇ **`unknown` type**

```
let dontKnow: unknown;
dontKnow = 42;
dontKnow = "Hello";
dontKnow = true;
dontKnow = { name: "Bob" };
dontKnow = [4, 5, 6];
console.log(dontKnow);
```

- `unknown` is safer than `any` because it forces to **narrow the type** before using it.
- TypeScript won't let us access properties or call methods on `unknown` without type checks.

```
if (typeof dontKnow === "string") {
  console.log(dontKnow.toUpperCase()); // Safe
}
```

3.4 Enum Types

> - Enums provide a set of named constants.
> - Improve code readability and reduce hardcoded strings.

```
enum dayOfWeek {
  Monday = "Monday",
  Tuesday = "Tuesday",
  Wednesday = "Wednesday",
}
let day = dayOfWeek.Monday;
```

3.5 Arrays

◇ **Typed Arrays**

```
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: string[] = ["one", "two", "three"];
let booleans: boolean[] = [true, false, true];
```

◇ **Mixed-Type Arrays**

```
let mixed: (number | string | boolean)[] = [1, "two", true, 3];
let mixedArray: any[] = [1, "two", true, { name: "Charlie" }, [5, 6, 7]];
```

- Union types maintain **type safety**.
- `any[]` disables type checking on the array elements.

◇ **Type Inference**

```
let numberInference = [1, 2, 3]; // number[]
let mixedInference = [1, "two", true]; // (number | string | boolean)[]
```

◇ `const` **Arrays vs** `readonly` **Arrays**

```
const constArray = [1, 2, 3]; // Mutable elements
constArray[0] = 10; // ☑ Allowed

const constReadOnlyArray: readonly number[] = [1, 2, 3];
// constReadOnlyArray[0] = 10;        // ❌ Error: readonly array
```

> ☑ Use `readonly` for truly immutable arrays in tests and shared data.

💡 **When do we use mixed arrays in testing?**

In UI tests (e.g. form validation), inputs can be of mixed types:

```
const inputs: arrays[] = ["abc", 123, false];
```

3.6 Tuple Types

```
let userInfo: [string, number, boolean];
userInfo = ["Tony", 24, true];
// userInfo = [24, "Tony", true]; // ❌ Error
```

> - ☑ Tuples have fixed length and fixed types at each index.
> - ☑ Used for tightly coupled data (like database row: name, age, status).

3.7 `never` type

◇ **Functions that never return**

```
function throwError(message: string): never {
  throw new Error(message);
}
```

- Used to represent a value that **should never occur**.
- Commonly used in error handling or infinite loops.

```
let myNever: never;
// myNever = 123; // ❌ Type error
```

> ☑ In UI testing, this is useful in strict utility helpers to ensure exhaustive type checking.

3.8 Spread Operator

```
let user3 = { name: "Alice", age: 30 };
let user4 = { ...user3, isEmployed: true };
```

- ☑ Useful for object cloning and merging.
- ⚠ Later values override earlier ones.

---

4 Date

- Reference:
  - MDN - Date - JavaScript
  - W3W Schools - Date - JavaScript
- Times and dates are handled using the built-in `Date` object. This object represents a specific moment in time, measured in millisecond since 01/01/1970 (the **epoch**)

```
const now: Date = new Date(); // 2025-09-28T10:43:34.342Z
```

Method	Output example
`getFullYear()`	`2025`
`getMonth()`	`8` (which is September)
`getDate()`	`25`
`getDay()`	`4`
`toISOString()`	`"2025-09-25T14:00:00.000Z"`
`toLocaleDateString()`	`"9/25/2025"` (computers format)

5. Number (nice to know)

> Whenever we are handling numbers in any programming language, we need to be aware of how the language handles numbers.

A few critical pieces of information about `numbers` in JavaScript that we should be aware of.

5.1 Core Type

- JavaScript has only one number type. It is a double-precision 64-bit Number.
- Meaning that, there are no seperate between `float` vs `double` vs `int` vs `long`
- Below we discuss its limitations along with a recommended solution.

5.2 Decimal / Floating Point Precision Issue

- Example:

```
0.1 + 0.2 = 0.30000000000000004
```

- While calculating money, rate, etc; we should use libraries like: `big.js` for precise decimal arithmetic

5.3 Integer limits

- The integer limits represented by the built in number type are `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`

```
console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991
console.log(Number.MIN_SAFE_INTEGER); // -9007199254740991
```

- Past those bounds, addition/subtraction can break (“round off”).
- **Safe** in this context refers to the fact that the value cannot be the result of a rounding error.
- The **unsafe values** are `+1 / -1` away from these values any amount of addition / subtraction will round the result. For demonstration:

```
console.log(Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2); //
true!
console.log(Number.MIN_SAFE_INTEGER - 1 === Number.MIN_SAFE_INTEGER - 2); //
true!

console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991
console.log(Number.MAX_SAFE_INTEGER + 1); // 9007199254740992 - Correct
console.log(Number.MAX_SAFE_INTEGER + 2); // 9007199254740992 - Rounded!
console.log(Number.MAX_SAFE_INTEGER + 3); // 9007199254740994 - Rounded -
```

```
correct by luck
  console.log(Number.MAX_SAFE_INTEGER + 4); // 9007199254740996 - Rounded!
```

`big.js` library

Whenever we use math for financial calculations (e.g. GST calculation, money with cents, addition etc) use a library like big.js which is designed for

- Perfect decimal math
- Safe out of bound integer values

Installation is simple:

```
  npm install big.js @types/big.js
```

Quick Usage example:

```
  import { Big } from "big.js";

  export const foo = new Big("111.11111111111111111");
  export const bar = foo.plus(new Big("0.00000000000000000001"));

// To get a number:
  const x: number = Number(bar.toString()); // Loses the precision
```

5.4 `NaN` (Not a Number)

- This happens when operations fail or are not representable. For example:
  - Square root of a negative number
  - Divided by 0
- **NOTE**: `NaN === NaN` is `false`; because `NaN` does not equal to itself.
- To check for NaN, do not use `Number.isNaN(x)`, instead, use this:

```
  Number.isNaN();
```

5.5 Infinity and -Infinity

Represent overflow / extremely large values.

- Beyond Number.MAX_VALUE → `Infinity`.

```
  console.log(Number.POSITIVE_INFINITY); // Infinity
  console.log(Number.NEGATIVE_INFINITY); // -Infinity
```

```
console.log(1 / 0); // Infinity
console.log(-1 / 0); // -Infinity

console.log(Infinity > 1); // true
console.log(-Infinity < -1); // true
```

5.6 Infinitesimal / Tiny numbers

- `Number.MIN_VALUE` is the smallest positive non-zero number `(~5e-324)`.

```
console.log(Number.MIN_VALUE); // 5e-324
```

- If we receive values smaller than `MIN_VALUE` ---> underflow to `0`.

```
console.log(Number.MIN_VALUE / 10); // 0
```

> Further intuition: Just like values bigger than `Number.MAX_VALUE` get clamped to INFINITY, values smaller than `Number.MIN_VALUE` get clamped to `0`.

---

6. Equality operator (== vs ===)

6.1 Loose Equality vs Strict Equality

Operator	Name	Type Coercion
`==`	Loose Equality	☑ Yes
`===`	Strict Equality	❌ No
`!=`	Loose Not Equal	☑ Yes
`!==`	Strict Not Equal	❌ No

Best practice:

- Should always use `===` and `!==`
- Avoid using `==` and `!=` except for verifying `null/undefined`

6.2 Examples

```
console.log(5 == "5"); // true   → type coercion ("5" becomes 5)
console.log(5 === "5"); // false  → different types (number vs string)

console.log("" == "0"); // false  → both are strings, but not equal
console.log(0 == ""); // true   → "" coerces to 0
```

```
console.log("" === "0"); // false  → both strings, but different
console.log(0 === ""); // false  → number vs string
```

---

7. Reference types

7.1 Object

Object (including array, function, ...) is reference types, not value types.

```
var foo = {};
var bar = foo; // bar trỏ cùng object với foo
```

- When we set `bar = foo`, we do not copy the object, but we copy the address to it --> bar is a reference to the same object.
- Therefore, `foo` and `bar` points to the same object in memory space

```
foo.baz = 123;
console.log(bar.baz); // Output: 123
```

- If we change that object through `foo.baz = 123` then `bar` would change too.

> ==NOTE==: we do not have 2 objects, we just have two variables that points to the same object.

7.2 Equality is for references

```
var foo = {};
var bar = foo; // bar = cùng object với foo
var baz = {}; // baz = object MỚI, khác với foo

console.log(foo === bar); // true (cùng tham chiếu)
console.log(foo === baz); // false (2 object khác nhau)
```

> ==REMEMBER==: Comparing object in JS is comparing references, not values !

---

8. Null & Undefined

---> Useful Youtube video: https://www.youtube.com/watch?v=kaUfBNzuUAI

8.1 Definition

JavaScript (and by extension TypeScript) has two bottom types : `null` and `undefined`. They are intended to mean different things:

- Something hasn't been initialized : `undefined`. For example:
  - a variable that we have defined without setting value
  - function that returns nothing
- Something is currently unavailable: `null`
  - we use `null` when we want to declare a variable may have value, but not at the moment

8.2 How TypeScript / JavaScript treat them in comparison

```
// Both null and undefined are only `==` to themselves and each other:
console.log(null == null); // true (of course)
console.log(undefined == undefined); // true (of course)
console.log(null == undefined); // true

// You don't have to worry about falsy values making through this check
console.log(0 == undefined); // false
console.log("" == undefined); // false
console.log(false == undefined); // false
```

- Therefore, using `==` is recommended because it checks for both `null` or `undefined`
- For example,

```
function foo(arg: string | null | undefined) {
  if (arg != null) {
    // here, arg is NOT null and NOT undefined → must be string
  }
}
```

8.3 Checking for “root-level undefined” & referencing undeclared variables

- If we use variables that are not yet declaired (global-scoped) --> `ReferenceError`
  - E.x: `console.log(someGlobal)` while `someGlobal` is not yet imported or `var/let/const`
- So to check if a variable is defined or not at a global level you normally use `typeof`:

```
if (typeof someGlobal !== "undefined") {
  // someGlobal is now safe to use
  console.log(someglobal);
}
```

8.4 Best practice: Avoid explicit use of `undefined` in Return Values

a. Functions that return `undefined` implicitly

In JavaScript/TypeScript, if a function has no `return` statement, it implicitly returns `undefined`.

```
function sayHello() {
  console.log("Hello");
}

const result = sayHello(); // prints "Hello"
console.log(result); // 👉 undefined
```

b. Explicitly returning `undefined` in an object

```
function getUser() {
  return {
    name: "Alice",
    age: undefined, // 👈 explicit undefined
  };
}
```

This adds a property `age` with a value of `undefined`. But here's the problem:

- When SERIALIZING to JSON, the `undefined` property will be removed
- It can lead to inconsistent or confusing behavior in APIs

c. Better strategy: use Optional Properties `?`

We can simply **ignore** the property and use `?` (optional) in the return type like this:

```
function getUser(): { name: string; age?: number } {
  return {
    name: "Alice",
    // no age field at all
  };
}
```

8.5 Callback & Validation function

To be continued 😃

---

9. Truthy

9.1 Definition

JavaScript has a concept of `truthy` i.e.

- Things that evaluate like true would in certain positions (e.g. if conditions and the boolean `&&`, `||` operators).
- An example is any number other than 0 e.g.

9.2 `truthy` & `falsy`

- Reference: MDN - Truthy - Falsy

Type	falsy	truthy
boolean	false	true

Type	falsy	truthy
boolean	false	true
string	Empty string ('', "")	Not empty string
number	0, NaN, -0	Others number
null	Always falsy	
undefined	Always falsy	
object		always truthy (even for empty [],{})

For example:

```
if (123) {
  // Will be treated like `true`
  console.log("Any number other than 0 is truthy");
}
```

🛠 10. Functions

- Block of code that can take inputs (parameters), do work, return output.
- Can be named, anonymous, arrow.
- There are different ways of declaring functions:

Type	Syntax Example	When to Use / Pros	Cons / Caveats
Function Declaration	`function add(a: number, b: number): number { return a + b; }`	Hoisted (can be used before it's defined), clean for main logic	❌ Verbose for simple callbacks
Function Expression	`const add = function(a: number, b: number): number { return a + b; };`	Flexible, can be passed as arguments or assigned to variables	❌ Not hoisted; anonymous functions are harder to debug
Arrow Function	`const add = (a: number, b: number): number => a + b;`	Short syntax, great for callbacks, `this`-safe (lexical `this`)	❌ Not hoisted; no own `this`, `arguments`, or `super`

> 💡 💡 💡 Use **arrow functions** when working with UI event handlers, promises, or concise operations.
> Use **declarations** when writing reusable named logic.

> Use **expressions** for inline logic or when we want to assign a function to a variable.

10.1 Function concepts & Type Annotations

- Parameters / Arguments: We can declare
  - data type: We declare the data type of each parameter using `:` syntax.

```
function greet(name: string, age: number): void {
  console.log(`Hello ${name}, you are ${age} years old.`);
}
```

  - optional parameters: using (`?`) annotation

```
    ⚠⚠⚠ // Optional parameters must always come after required
ones.
    function greet(name: string, age?: number): void {
      console.log(`Hello ${name}`);
      if (age !== undefined) {
        console.log(`You are ${age} years old.`);
        }
    }
```

  - default parameters: providing a default value to a parameter. If the caller doesn't supply the value, the default will be used. --> Hence, default parameters can be treated as optional, but with a fallback value.

```
function greet(name: string = "Guest"): void {
  console.log(`Hello ${name}`);
}

greet(); // 👉 "Hello Guest"
greet("Charlie"); // 👉 "Hello Charlie"
```

    UI use case example: Used to return a fallback avatar image when `size` is not specified.

```
function getAvatarUrl(userId: string, size: number = 50): string {
  return `https://api.adorable.io/avatars/${size}/${userId}.png`;
}
```

  - rest parameters: when we want to accept multiple arguments and handle them as an array.

```
function sum(...nums: number[]): number {
  return nums.reduce((acc, val) => acc + val, 0);
```

```
}
sum(1, 2, 3); // 👉 6
sum(); // 👉 0
```

- Here `...nums` gathers all remaining arguments into an array of `numbers[]`.
- Only 1 rest parameter is allowed per function, and it must come last.

- Return type: we can define the type of value that a function returns, after the parameter list, using `:` annotation.

COMBINING ALL CONCEPTS

```
function logInfo(name: string, age: number = 18, ...tags: string[]): string {
  return `${name}, Age: ${age}, Tags: ${tags.join(", ")}`;
}

logInfo("Sam"); // "Sam, Age: 18, Tags: "
logInfo("Linh", 22, "student", "intern"); // "Linh, Age: 22, Tags: student,
intern"
```

10.2 Function Declaration

- By doing this, the function will be declared before executed
- We should use this method

```
// Function Declaration
sayHi("John"); // Hello, John

function sayHi(name: string): string {
  alert(`Hello, ${name}`);
}
```

Explanation:

- `name: string`: parameter `name` must be a string.
- `: string`: return type of the function.

10.3 Function Expression

- A function created within another syntax
- It won't work if we put it like this

```
// Function Expression
sum(1, 2); // error!
let sum = function (a, b) {
  return a + b;
};
```

10.4 Arrow Function

- This one usually goes well with Function Expression

```
// Instead of coding like this
  let func = function (arg1, arg2, ...argN) {
    return expression;
  };
// We can do like this, which is much shorter
  let func = (arg1, arg2, ...argN) => expression;
```

- In JAVASCRIPT (not TYPESCRIPT) where there is only one parameter, we can ommit the `()`, which is conveniently short:

```
// let double = function(n) { return n * 2 }
let double = (n) => n * 2;
alert(double(3)); // 6
```

- Reference:
  - Arrow Function
  - Arrow Function - Vietnamese

```
  const add = (a: number, b: number): number => {
    return a + b;
  };
```

---

11. `Class` vs `Interface` vs `Type Alias`

TypeScript gives us different ways to define the **shape of data:** ☑ `type` alias ☑ `interface` ☑ `class`

> Each serves a different purpose.

11.1 `type` Alias

🧩 Defining custom data type

A `type` alias help us simplify complex or repetitive structures.

> - Type aliases define **reusable structures** for objects.
> - Use `?` for **optional fields**.
> - Reusing `types` for function parameters or return values.

◇ `type` **alias Syntax**

```
// syntax
  type Car = {
    name: string;
    model: string;
    powerHorse: number;
    isHybrid?: boolean;
  };
```

Union & Intersection

We can also define `union` and `intersection` types with `|` and `&`

```
  type Addidas = { color: string };
  type Point = [number, number];                          // Tuple
  type mixCarAddidas = Car & Addidas;                     // Intersection
  type orType = Car | Addidas;                            // Union
  type ResponseStatus = "success" | "error" | "loading"; // Type Literal
```

Note:

- Can define **primitive types**, unions, tuples, etc.
- Cannot implement or extend like a `class` or `interface`.

11.2 `interface`

> - Similar to `type`, but ***extendable***.
> - Preferable when working with class-based OOP.

◇ **`interface` Syntax**

```
// syntax
  interface AddidasInterface {
    color: string;
  }
```

Note:

- Can use `extends` for inheritance.
- Can be merged (declaration merging).
- ❌ Interfaces CANNOT DEFINE UNION TYPES.

◇ **Extending `interface`**

```ts
// Extends in `interface`
interface Engine {
  engineType: string;
}

interface CarInterface {
  name: string;
  model: string;
  powerHorse: number;
  isHybrid?: boolean;
}

interface FullCar extends CarInterface, Engine {}

let myCar: FullCar = {
  name: "Tesla",
  model: "S",
  powerHorse: 500,
  engineType: "Electric"
};
```

◇ `interface` **merging**

> We can define the same interface multiple times, and TypeScript merges them.

```ts
interface Animal {
  name: string;
}

interface Animal {
  age: number;
}

const a: Animal = { name: "Dog", age: 3 }; // valid!
```

11.3 `class`

> - A `class` is a blueprint for creating ***objects*** with specific structure and behavior.
> - Unlike others, a `class` can contain both: - data: *properties* - logic: *methods*

◇ `class` **Syntax**

- `class`: declares a class.
- `constructor`: the `constructor` method is called when you create an instance of a class.
  - We use `this.propertyName` to refer to the class's own properties.
- `Access Modifiers`:

Modifier	Description	Use case example

Modifier	Description	Use case example
`public`	(default) accessible anywhere	Default for most properties
`private`	only accessible inside the class	Internal logic, hidden data
`protected`	accessible in class & subclasses	OOP with inheritance

```
class Employee {
  public name: string;
  private salary: number;
  protected department: string;
  readonly id: string;

  constructor(name: string, salary: number, dept: string, id: string) {
    this.name = name;
    this.salary = salary;
    this.department = dept;
    this.id = id;
  }

  getSalary(): number {
    return this.salary; // OK
  }

  printId() {
    console.log(this.id);
  }
}
```

- **Shortcuts**

Instead of declaring properties separately and then assigning them in the constructor, TypeScript allows this shorthand:

```
class Book {
  constructor(
    public title: string,
    private author: string,
    readonly isbn: string
  ) {}

  getAuthor(): string {
    return this.author;
  }
}
```

- **Inheritance (Extending a class)**

We can create a subclass that **inherits** from a base class using `extends`:

```
class Animal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name); // must call parent constructor
  }

  makeSound(): void {
    console.log(`${this.name} barks.`);
  }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Buddy barks.
```