




# Content

- Introduce  TypeScript
- Basic  examples
- Practical  exercise

# Lời nói đầu 🧐

- giới thiệu bản thân.
- đây là 1 buổi chia sẻ và bàn luận!
- lý do tại sao mình lại muốn chia sẻ?
- lý do mọi người lại đến nghe?
- khi nào có thắc mắc thêm thì phải đặt câu hỏi ngay, sợ quên lắm!
- có thời gian nghỉ mỗi 45 phút.
- khi nào cảm thấy nội dung "nhàm chán", "buồn ngủ" thì phải lên tiếng luôn!
- "chưa hiểu" thì phải bảo là "chưa hiểu"!

# Part 1 - Introduce 🌱 TypeScript



Photo by [Jack Anstey](#) on Unsplash.

# Tại sao người ta lại yêu thích ❤️ TypeScript

Theo khảo sát của [stack-overflow năm 2021](#) và [năm 2020](#) thì TypeScript đều nằm trong top 3 ngôn ngữ được các developer yêu thích nhất.

Vì sao lại vậy?

**Chiến thuật *code ít bug* của TypeScript ở đây là gì?**

# Định nghĩa

“ TypeScript is JavaScript with syntax for types.  
TypeScript is a strongly typed programming language ”

Đây là 2 điều viết ngắn gọn ở [trang chủ TypeScript](#).

# Vấn đề của JavaScript

Ta nhắc lại nhược điểm cũng là ưu điểm của JavaScript, rằng JavaScript là 1 trong số những ngôn ngữ dynamically typed language.

# Ví dụ với JavaScript

Đoạn code sau sẽ không có lỗi gì, nhưng sẽ *toang* khi chạy:

```
```js
const compact = (arr) => {
  // error: `orr` (not `arr`)
  if (orr.length > 10) {
    // error: `arr` can be anything,
    // so maybe property 'trim' does not exist on it!
    return arr.trim(0, 10)
  }

  return arr
}
```
```



# Ví dụ với TypeScript

Đoạn code sau sẽ không có lỗi gì, nhưng sẽ *toang* khi chạy:

```
```ts
const compact = (arr: string[]) => {
  if (arr.length > 10)
    return arr.slice(0, 10)
  return arr
}
```
```

# JavaScript vs TypeScript

| JavaScript  | TypeScript   |
|---|--|
| <b>code nhanh hơn</b> 🐇, vì code không cần quan tâm tới type                      | <b>code chậm hơn</b> 🐢, cần quan tâm tới type khi code   |
| <b>dễ học</b> 🐇, code không cần quan tâm tới type                                 | <b>học khó hơn</b> 🐢, cần học cách sử dụng type  |
| <b>khó bảo trì</b> 🤔, khi hệ thống phức tạp, nhiều thứ không được định nghĩa type | <b>bảo trì dễ hơn</b> 😎, mọi thứ được định nghĩa type, dễ hiểu hơn, đặc biệt là định nghĩa params, response API, function... |

Ví dụ đoạn code JS sau:

```
// Accessing the property 'toLowerCase'  
// on 'message' and then calling it  
message.toLowerCase();
```

```
// Calling 'message'  
message();
```

Những câu hỏi hoang mang:

- Is message callable?
- Does it have a property called toLowerCase on it?
- If it does, is toLowerCase even callable?
- If both of these values are callable, what do they return?

Giải pháp:

“ a static type system to make predictions about what code is expected before it runs. ”

# Một số lợi ích khác của TypeScript

Chính vì TypeScript hiểu object, interface, class... nên nó có thể *auto-complete* hay gợi ý các lựa chọn cho ta.

```
const cat = { eat: '🐟' };
```

```
// suggestion: `cat.run`  
cat.
```

Để sử dụng lib/package bên thứ 3 cung cấp. Ví dụ:

```
// `sftp` is a client object instance of 'ssh2-sftp-client' package  
// `filePath` is a string  
const type: string | boolean = await sftp.exists(filePath);
```

```
// node_modules/@types/ssh2-sftp-client/index.d.ts  
declare class sftp {  
    exists(remotePath: string): Promise<false | FileInfoType>;  
}
```

# typescript compiler

```
npm install -g typescript
```

- type checking
- chuyển file .ts sang file JavaScript .js, từ đây ta có thể chạy file .js như vẫn làm với JavaScript.
  - Thế nên ta có thể dùng TypeScript trên môi trường browser or Node.js.

Ví dụ compile đoạn code sau từ ts sang js:

```
const add = (num1: number, num2: number): number => {  
    return num1 + num2;  
};
```

```
console.log("🚀 ~ file: add.ts ~ line 5 ~ add(2, 3)", add(2, 3));
```

```
tsc add.ts  
# expected output: add.js
```



## Part 2 - Basic 🧱 examples



Photo by [Elliot Andrews](#) on Unsplash.

# Ví dụ đơn giản - add function

```
const add = (num1: number, num2: number): number => {  
    return num1 + num2;  
};
```

```
console.log("🚀 ~ file: add.ts ~ line 5 ~ add(2, 3)", add(2, 3));
```

# Run TypeScript file globally 🚀 with ts-node

```
node -v
```

```
# expected version is currently stable version
```

```
npm install -g typescript ts-node tslib @types/node
```

```
# verify
```

```
ts-node -v
```

# Prepare a simple sample

```
touch test.ts
```

Write a simple sample:

```
const teamName: string = '🦆🦆🦆';
```

```
console.log('🐙', teamName);
```

# Run ts file everywhere

```
ts-node test.ts
```

Expected output is "   .

# Ngoài ra có thể chạy bằng **deno** trên local 🦖

Cách cài đặt `deno` cụ thể có tại [đây](#).

MacOS:

```
brew install deno
```

Linux (ví dụ như Ubuntu):

```
sudo snap install deno
```

# Chạy bằng deno command

```
deno run ./test.ts
```

# Kiểu nguyên thủy

- string: ', "Hello, world"
- number: 0, -1, 3.14 (float), not BigInt (BigInt support has been added on [TypeScript 3.2](#))
- boolean: two values true and false



Ví dụ:

```
let aStr: string;
```

```
aStr = 'hello';
```

```
// 😎 err: Type 'undefined' is not assignable to type 'string'
```

```
aStr = undefined;
```

# Array type

```
const aArray: number[] = [1, 2, 3]
```

Có 2 cách viết: `number[]` hoặc `Array<number>`.

Cách viết `number[]` dễ đọc và ngắn gọn hơn.

# `any` type

Trong TypeScript mọi thứ sẽ được check type, ngoại trừ `any`.

“ Using `any` disables all further type checking. ”

```
let obj: any = { x: 0 };
```

```
// no error 🤔
```

```
obj.foo();
```

```
obj();
```

```
obj.bar = 100;
```

```
obj = 'hello';
```

Mọi biến được gán bởi 1 giá trị `any` đều được chấp nhận.

```
let obj: any = { x: 0 };
```

```
// no error 🤔
```

```
const num: number = obj;
```

Code typescript mà để `any` thì chẳng khác nào code JavaScript!

Nên hạn chế sử dụng `any` nhất có thể.

Vậy thì khi nào chấp nhận dùng `any`?

Vậy thì khi nào chấp nhận dùng `any`?

- Thiết kế model chưa rõ ràng, trong khi thay đổi quá nhiều lần. Ta tạm thời dùng `any`.

Đối với phần code common có quá nhiều params/return type thì ta có thể sử dụng [generic types](#).

# Hạn chế **any** bằng tools

Có 2 cách là: dùng *tsconfig* và *eslint*.

# noImplicitAny - không được ngầm hiểu là any

Đây là [1 config trong file tsconfig.json](#).

“ In some cases where no type annotations are present, TypeScript will fall back to a type of any for a variable when it cannot infer the type. ”



This can cause some errors to be missed, for example:

```
const takeFrom3rdPosition = (str) => {  
  // No error? `.subtr` -> `.substr`  
  console.log(str.subtr(3));  
}
```

```
// No error?  
takeFrom3rdPosition(42);
```

Bật config `noImplicitAny` trong `tsconfig.json`:

```
{
  "compilerOptions": {
    // ...
    "noImplicitAny": true,
  }
}
```

Kết quả:

```
const takeFrom3rdPosition = (str) => {  
  // err: Parameter 'str' implicitly has an 'any' type.  
  console.log(str.substr(3));  
}
```

# Sử dụng eslint

`.eslintrc.json`:

```
{
  // ..
  {
    "files": "*.ts",
    "parser": "@typescript-eslint/parser",
    "plugins": ["@typescript-eslint"],
    "rules": {
      "@typescript-eslint/no-explicit-any": 1,
    }
  }
}
```

# Về việc chú thích kiểu với biến

```
let myName: string = "Alice";
```

“ In most cases, though, this isn’t needed. Wherever possible, TypeScript tries to automatically infer the types in your code. ”

Chú thích kiểu là phần nằm bên phải biến, với cú pháp là  
`<2 dots mark><space><type>`.

Khi nào thì cần phải dùng chú thích kiểu - type annotations?

Khi mà TypeScript không tự hiểu được kiểu của biến đó, tại thời điểm khởi tạo. Ví dụ:

```
let myLand: string | undefined = undefined;
```

```
myLand = 'LaLa';
```

# Object Types

```
// The parameter's type annotation is an object type  
function printCoord(pt: { x: number; y: number }) {  
    console.log("The coordinate's x value is " + pt.x);  
    console.log("The coordinate's y value is " + pt.y);  
}  
printCoord({ x: 3, y: 7 });
```



# Union Types - phép hợp

TypeScript's type system cho phép chúng ta tạo ra các type mới dựa trên các type có sẵn và các toán tử.

```
const printId = (id: number | string) => {  
  console.log('Your ID is: ' + id);  
};
```

Union, dịch là "phép hợp".

`id: number | string` nghĩa là param id có thể nhận 1 trong 2 kiểu number, hoặc string.

# Intersection types - phép giao

```
type TLocation = { x: number } & { y: number };  
const myLocation: TLocation = { x: 0, y: 0 };
```

`{ x: number } & { y: number }` bao gồm cả thuộc tính x và y.

# Type Aliases

```
type Point = {  
  x: number;  
  y: number;  
};
```

```
const printCoord = (pt: Point) => {  
  console.log("The coordinate's x value is " + pt.x);  
  console.log("The coordinate's y value is " + pt.y);  
};
```

Đơn giản chỉ là 1 alias - biệt danh. Ví dụ:

```
type UserInputSanitizedString = string;
```

# Type aliases với union

```
type Size = 'small' | 'medium' | 'large' | 'big';
```

# Interfaces

“ An interface declaration is another way to name an object type: ”

```
interface Point {  
  x: number;  
  y: number;  
}
```

# Differences Between Type Aliases and Interfaces

Extending an interface:

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

```
interface Bear extends Animal {  
    honey: boolean  
}
```



## Extending a type via intersections (phép giao):

```
type Animal = {  
    name: string  
}
```

```
type Bear = Animal & {  
    honey: boolean  
}
```

## Adding new fields to an existing interface:

```
interface IWindow {  
  title: string;  
}  
interface IWindow {  
  color: string;  
}  
const inputWindow: IWindow = {  
  title: '👤 alert!',  
  color: '🌈',  
};
```

A type cannot be changed after being created!

## Tóm lại:

- interface có tính mở rộng cao hơn
  - có thể update - thêm thuộc tính
  - class có thể extends từ interface
  - có thể được sử dụng như type aliases, khi chú thích kiểu
- type aliases thì được dùng như 1 biến, có thể nhận kết quả từ các toán tử với type. Còn interface cần phải có 1 cấu trúc.

Khi định nghĩa models trong thực tế, hay để tái sử dụng type thì người ta thường dùng interface.

Type aliases được dùng khi muốn tái sử dụng biểu thức với type.

# generic types

Sinh ra để giải quyết bài toán tái sử dụng src/logic với type thay đổi theo từng trường hợp.

```
function identity<Type>(arg: Type): Type {  
  return arg;  
}
```

```
const doSomething = <Type>(arg: Type): Type => {  
  return arg;  
};
```

# Classes

```
class Point {  
    x: number;  
    y: number;  
}
```

```
const pt = new Point();  
pt.x = 0;  
pt.y = 0;
```

# Cách dùng từ khóa `readonly` trong class, type, interface

Trong 1 `class`, `type` hoặc `interface`, nếu 1 thuộc tính không bao giờ thay đổi thì ta có thể sử dụng `readonly`.

Nếu được gán lại bằng 1 giá trị khác thì sẽ báo lỗi  
`không thể gán giá trị mới cho thuộc tính này`.

“ error TS2540: Cannot assign to XXX because it is a constant or a read-only property. ”

# Ví dụ trong 1 component có thuộc tính

```
public readonly columnTitles = columnTitles;  
public readonly nameMaxLength = nameMaxLength;
```



Ví dụ về khởi tạo giá trị trong `constructor` của `class`:

```
class Employee {  
    readonly code: number;  
    name: string;  
  
    constructor(code: number, name: string) {  
        this.code = code;  
        this.name = name;  
    }  
}
```

```
let emp = new Employee(10, "John");  
emp.code = 20; // Compiler Error  
emp.name = 'Bill';
```

## Ví dụ về thuộc tính `readonly` của `interface`:

```
interface IEmployee {  
    readonly code: number;  
    name: string;  
}
```

```
let empObj: IEmployee = {  
    code: 1,  
    name: 'Steve';  
}
```

```
empObj.code = 100; // Compiler Error: Cannot change readonly 'code'
```

## Ví dụ về thuộc tính `readonly` của `type`:

```
// or `interface IEmployee`
```

```
type Employee = {  
    empCode: number;  
    empName: string;  
}
```

```
let emp1: Readonly<Employee> = {  
    empCode: 1,  
    empName: 'Steve'  
}
```

```
emp1.empCode = 100; // Compiler Error: Cannot change readonly 'empCode'  
emp1.empName = 'Bill'; // Compiler Error: Cannot change readonly 'empName'
```

```
let emp2: Employee = {  
    empCode: 1,  
    empName: 'Steve'  
}
```

```
emp2.empCode = 100; // OK  
emp2.empName = 'Bill'; // OK
```

Việc dùng `readonly` chỉ nhằm mục đích tránh những nhầm lẫn về logic, như việc đáng ra nó là biến không được thay đổi nhưng lại bị gán lại chẳng hạn.

# Data Type - Never

`never` là 1 type không tồn tại trong JavaScript.

`never` hay kiểu không bao giờ, được định nghĩa là kiểu trả về khi mà bạn chắc chắn là không thể trả về được 1 giá trị nào đó do vòng lặp vô hạn hay chủ động dừng, không thực hiện tiếp.

Ví dụ 1 function không thực hiện được hết, hay function đó luôn bị dừng do throw 1 exception - ngoại lệ:

```
const throwError = (errorMsg: string): never => {  
    throw new Error(errorMsg);  
};
```

```
const keepProcessing = (): never => {  
  while (true) {  
    console.log('I always does something and never ends.');  }  
};
```

function không thực hiện được hết.

`throwError` và `while (true)` sẽ ngăn function thực hiện xong, thế nên nó tất nhiên không bao giờ có thể return `void`. Và kiểu function này, ta có thể đặt kiểu trả về là `never`.



# Ứng dụng của `never`

## Hữu dụng khi báo lỗi

Nếu bạn code nhiều TypeScript thì sẽ quen mặt thẳng `never` này ở thông báo lỗi khi build project. Bởi lẽ `never` thường đi kèm với việc có gì đó đang không thể return được.

# Sử dụng với Generic types và condition types

```
type TNonNullable<T> = T extends null | undefined ? never : T;
```

```
// ❌ error: Type 'undefined' is not assignable to type 'never'.
```

```
const value: TNonNullable<undefined> = undefined;
```

```
console.log('🚀 ~ value', value);
```

```
// another example:  
type TNotANumber<T> = T extends number ? never : T;  
  
// ❌ error: Type 'number' is not assignable to type 'never'.  
const notANumberValue: TNotANumber<number> = 1;  
console.log('🚀 ~ notANumberValue', notANumberValue);
```

TNonNullable sẽ không chấp nhận `null | undefined`. TNotANumber cũng không chấp nhận `1` `number` type.

## Part 3 - Practical 🏃 exercise



Photo by [Hoach Le Dinh](#) on Unsplash.

Cảm ơn vì đã lắng nghe 🌱



Photo by [Jack Anstey](#) on Unsplash.