Chapter 1: Getting started **3**
#0. Setup Environment 3
#1. What is TypeScript 4
#2. TypeScript Setup 5
#3. TypeScript “Hello, World!” 5
#4. Why TypeScript 7

Chapter 2: Basic Types **8**
#5. TypeScript Types 8
#6. Type Annotations - Keyword Type 9
#7. Type Inference - Tự động gán type 10
#8. Number Type 11
#9. String Type 12
#10. Boolean Type 13
#11. Object Type 14
#12. Array Type 16
#13. Tuple Types 17
#14. Enum Types 18
#15. TypeScript any Type 19
#16. TypeScript void Type 19
#17. TypeScript Data Type - Never 20
#18. TypeScript union Type 21
#19. TypeScript Type Aliases 22

Chapter 3: Control Flow Statements **23**
#20. TypeScript if else 23
#21. TypeScript switch case 25
#22. TypeScript for 27
#23. TypeScript while 29
#24. TypeScript do...while 30
#25. TypeScript break 31
#26. TypeScript continue 32

Chapter 4: Functions **33**
#27. TypeScript Functions 33
#28. TypeScript Function Types 34
#29. TypeScript Optional Parameters 35
#30. TypeScript Default Parameters 36

#31. TypeScript Rest Parameters 38
#32. TypeScript Function Overloading 39
#33. Classes 41
#34. TypeScript Access Modifiers 43
#35. TypeScript readonly 46
#36. Typescript Getters and Setters 47
#37 TypeScript Inheritance 49
#38. TypeScript Static Methods and Properties 50
#39 TypeScript abstract classes 52

Chapter 6: Interfaces 53
#40. Interfaces 53
#41. How to extend interfaces in Typescript 56

Chapter 7: Advanced Types 59
#43. Intersection Types 59
#44 TypeScript Type Guards 62

Chapter 1: Getting started

#0. Setup Environment

VSCode - Công cụ để code Javascript/Typescript:

- Tải VSCode: https://code.visualstudio.com/download
- Sử dụng VSCode vì nó free, và support mạnh mẽ cho javascript/typescript

Node.js - Môi trường thực thi Javascript/Typescript

- Download Node.js: https://nodejs.org/en/download/
- Node.js là môi trường để chạy code, chứ không phải là framework hay library. Nó là platform. Tương tự như windows là môi trường để chạy ứng dụng Microsoft Words (ở đây, node.js là môi trường để chạy ứng dụng React - viết bằng ngôn ngữ js)

Full source code của cả khóa học:

- Download project với Github: https://github.com/haryphamdev/hoidanit-typescript-basic

- Các bước cần thực hiện để cài đặt project thực hành:
  B1: Đảm bảo rằng máy tính các bạn đã cài đặt Git

  B2: Download project từ link ở trên với Git clone

  B3: Chạy project theo từng video hướng dẫn trong khóa học (sẽ đề cập trong quá trình học)

SOS - Cần support thì làm thế nào ?

Các bạn có thể comment trực tiếp vấn đề gặp phải trên từng video Youtube , hoặc

Đăng câu hỏi lên Group cộng đồng của Hỏi Dân IT:
https://www.facebook.com/groups/hoidanit

để được hỗ trợ và giải đáp thắc mắc nhé !

#1. What is TypeScript

1.1 Giới thiệu về TypeScript

TypeScript (viết tắt là TS) là một "phong cách mới" để code Javascript (JS)

Browser chỉ có thể hiểu HTML/CSS và Javascript thuần, **KHÔNG HIỂU typescript.**

Vậy làm sao để chạy được 1 project typescript ?

Typescript => TypeScript Compiler => Javascript

---

with developer:

- TypeScript sử dụng "cú pháp của Javascript", và bổ sung thêm các cú pháp mới để hỗ trợ "Types" (các kiểu dữ liệu).

Những kiến thức đã biết về Javascript đều có thể áp dụng trong Typescript.

- Để viết Typescript, chúng ta định nghĩa file cuối **extension là .ts,** thay vì .js (javascript).
=> có thể viết code js bên trong ts. và có thể rename .js thành .ts để viết code TypeScript

1.2 Tại sao lại dùng Typescript.

- Ép kiểu dữ liệu chặt chẽ hơn (giảm sự freedom so với javascript).
Sử dụng Type để định nghĩa dữ liệu, tương tự Java, C#...

- Giúp hạn chế bugs, đặc biệt trong các dự án có khối lượng codebase (dòng code) lớn.

//ví dụ của lesson 1 với JS và TS

#2. TypeScript Setup

Cài đặt thư viện:
npm install -g typescript@4.8.3 ts-node@10.9.1

check version typescript compiler:
tsc -v //Version 4.8.3

- Nếu có lỗi:
'tsc' is not recognized as an internal or external command,
operable program or batch file.

+ dùng GitBash
+ tắt CMD, rồi mở lại
+ setup env cho máy tính: https://stackoverflow.com/a/40616601

#3. TypeScript "Hello, World!"

1. Coding

Download file: ....

B1: Tạo file lesson3.ts, nội dung như sau:

```
let message: string = 'Hello, World!';
console.log(message);
```

B2: Mở terminal của VScode, sử dụng phím tắt Ctrl+` or từ menu Terminal > New Terminal

B3: Chạy câu lệnh:
tsc lesson3.ts

- hiện lỗi File 'lesson3.ts' not found.

=> cần vào đúng thư mục

có 2 cách làm:

Cách 1:
Dùng full đường dẫn tới file cần dịch: (gõ và nhấn phím Tab để được gợi ý)
tsc .\projects\lesson3\lesson3.ts

Cách 2:
cd .\projects\lesson3\

chạy lại câu lệnh trên: tsc lesson3.ts

Nếu chạy thành công, sẽ xuất hiện file "đã được dịch sang js" là: lesson3.js

Quá trình chuyển từ file .ts sang file .js gọi là quá trình compile (dịch code). Ở đây, thư viện tsc đã làm điều này.

2. Chạy file Javascript

Để chạy file js, có 2 cách: (Lưu ý là chạy file .js, không phải .ts)

Cách 1: sử dụng nodejs

chạy file js: **node file_js_cần_run**

node .\projects\lesson3\lesson3.js

Nếu sử dụng thư viện ts-node, dịch code + run trong 1 câu lệnh:

(lưu ý: sử dụng file .ts, thay vì js ở trên)
ts-node .\projects\lesson3\lesson3.ts

=> với câu lệnh này không tạo ra file .js :D

Cách 2: sử dụng browser (với file .js)

#4. Why TypeScript

TS với hệ thống Types => hạn chế bugs khi khối lượng codebase lớn (các project lớn, nhiều members)

1. Dynamic types với javascript

Javascript là ngôn ngữ với 'type' động, có nghĩa là chúng ta có thể thay đổi "datatype" - kiểu dữ liệu

ví dụ:
let name = 'abc'; typeof === string
name = 35; typeof === number
name = false; typeof === boolean

Dynamic types (type động) mang tới sự freedom, tuy nhiên, tự do quá thì...

2. Vấn đề với Dynamic types

- Cho ra sai kết quả (nếu truyền vào sai type) => cần code bổ sung validate
=> typescript sinh ra để khắc phục nhược điểm trên

Chapter 2: Basic Types
#5. TypeScript Types

1. Type là gì ?

Type gồm 2 loại: **keyword 'type' và data-types** (kiểu dữ liệu)
Ở đây, chúng ta sẽ đề cập tới 'data-types: kiểu dữ liệu

Việc định nghĩa kiểu dữ liệu (data-types), sẽ nói cho chúng ta biết,
biến số (variable) có những thuộc tính và functions nào

Ví dụ:
const name = 'Hello'.

Khi nhìn vào value 'Hello', chúng ta sẽ nghĩ nó là string. (data-type === String)

Một vài thuộc tính và function của string:

console.log('Hello'.length) //5 => property: length (có thuộc tính length)

console.log('Hello'.toUpperCase()); // HELLO => có functions

=> dựa vào type (data-types), trình compiler sẽ biết được 1 tham số (variable/value)
sẽ có những thuộc tính (property) và phương thức (functions) nào.

2. Các loại data-types với Typescript

Tương tự như javascript, TS datatype bao gồm:

- Dữ liệu nguyên thủy
- Dữ liệu tham chiếu

+ Primitive data- types:
String, number, boolean, null, undefined, symbol, bigint

+ References dataTypes: Objects, Array, Functions..

#6. Type Annotations - Keyword Type

TypeScript sử dụng cú pháp:

:type sau khi định nghĩa biến. Một khi đã khai báo type, chúng ta sẽ không thể thay đổi nó (static type)

Ví dụ:
let variableName: type; => khai báo 1 biến, định nghĩa type, nhưng chưa khởi tạo giá trị
let variableName: type = value; => khai báo 1 biến, định nghĩa type, và gán giá trị khởi tạo

const constantName: type = value;

Với kiểu dữ liệu nguyên thủy (primitive data-type)

let count: number;
count = 1; //ok

count = 'name'; //error

let count: number = 1; //sử dụng khai báo type và init value

Dữ liệu tham chiếu - References data types:
let arrayName: type[ ];

ví dụ:
let names: string[ ] = ['a', 'b', 'c'] // array chỉ bao gồm string
names.push(1) //error

#7. Type Inference - Tự động gán type

Tự động gán type
Khi khai báo 1 biến với typescript, chúng ta 'thường' gán type cho nó
vd:
let count: number; // khởi tạo biến count, kiểu 'number', và không có giá trị khởi tạo

let count: number = 1: // khởi tạo biến count, kiểu 'number', và giá trị khởi tạo = 1;

Đôi khi chúng ta code javascript:

let count = 0; //ok ? => typescript sẽ 'suy luận' và tự động gán kiểu 'number ' cho biến khởi tạo

Type inference vs. Type annotations

Type inference: TS sẽ "đoán" kiểu dữ liệu dựa vào "thuật toán của nó"

Type annotations: định nghĩa "ép buộc chính xác" kiểu dữ liệu

Khi nào sử dụng Type inference (Tự động gán type): khi nào muốn ?

Sử dụng Type annotations (định nghĩa type) khi:
- Khai báo 1 biến và gán giá trị cho nó ngay sau đó
- Khai báo 1 biến => ép kiểu (không muốn được tự động gán)
- Muốn ép kiểu trả về giá trị mong muốn

Lưu ý về viết chữ Hoa/Thường khi định nghĩa type:

https://stackoverflow.com/questions/14727044/what-is-the-difference-between-types-string-and-string

#8. Number Type
Tương tự javascript, typescript kế thừa lại kiểu number, tức là:

Number sẽ bao gồm: số thực (float), số nguyên (integer), hoặc thậm chí là bigint

Lưu ý:
let a = 9.69; typeof === number
và let b = 10; typeof === number

Typescript (hay JS) không có khái niệm kiểu dữ liệu (datatype) là float hay int, double... chỉ có kiểu Number (hiển thị số)

let price: number; // khai báo biến price, kiểu number, và không khởi tạo giá trị ban đầu

let price: number = 10;

let price = 6.69;

#9. String Type

Tương tự Javascript, chúng ta có thể dùng double quotes (") hoặc single quotes (') để khai báo string.

```
let firstName: string = 'ABC';
let fullName: string;
fullName = "DEF";
```

Ngoài ra, chúng ta cũng có thể sử dụng dấu backtick (`)

```
let test = ` abc
def `;

let mine: string = ` my name is ${fullName} `;
```

#10. Boolean Type
Tương tự JS, kiểu dữ liệu boolean chỉ có 2 giá trị: true hoặc false.

```
let pending: boolean;

pending = true;//ok

//do something

pending = false; //ok
```

#11. Object Type

Kiểu dữ liệu object
Kiểu dữ liệu object của Typescript, tượng trưng cho tất cả dữ liệu **KHÔNG** là kiểu dữ liệu nguyên thủy,
bao gồm: string, number, boolean, null, undefined, bigint, symbol.

```
let employee: object;

employee = {
   firstName: 'John',
   lastName: 'Doe',
   age: 25,
   jobTitle: 'Web Developer'
};

console.log(employee);


employee = "Jane"; //error
console.log(employee.hireDate); //error
---
let employee: {
   firstName: string;
   lastName: string;
   age: number;
   jobTitle: string;
};

employee = {
   firstName: 'John',
   lastName: 'Doe',
   age: 25,
   jobTitle: 'Web Developer'
};
```

or shorter:

```
let employee: {
    firstName: string;
    lastName: string;
    age: number;
    jobTitle: string;
} = {
    firstName: 'John',
    lastName: 'Doe',
    age: 25,
    jobTitle: 'Web Developer'
};
```

Lưu ý về object vs Object vs { }

https://stackoverflow.com/a/49465172

- Sử dụng object (viết thường chữ o)

#12. Array Type

Định nghĩa array type
Để định nghĩa "array" type, sử dụng cú pháp:
let arrayName: type[ ];

vd:
let skills: string[ ]; // định nghĩa array "strings"

skills[0] = "ABC";
skills[1] = 69; //error
skills.push("DEF"); //ok

//với js
let skills = ['Problem Solving','Software Design','Programming'];

=> ts sẽ "tự đoán và định nghĩa kiểu type"

let skills: string [ ] = ['Problem Solving','Software Design','Programming'];

skills.push(96); //error

Lưu trữ "mixed types"

let scores = ['Programming', 5, 'Software Design', 4];

=> TS sẽ đoán gồm type string hoặc number

let scores : (string | number)[ ] = ['Programming', 5, 'Software Design', 4];

#13. Tuple Types
"Tuple": a structure of data that has several parts

Định nghĩa
Hoạt động giống như Array, tuy nhiên, có thêm một vài quy định bắt buộc:
- Số phần tử của tuple cần được khai báo trước (cố định)
- Type của từng phần tử trong array cần được khai báo trước, và không nhất thiết phải giống nhau

ví dụ: let skills = ['Languages', 5];

=>

let skills : [string, number] = ['Languages', 5];

- Với tuple, thứ tự của phần tử rất QUAN TRỌNG.

let skills : [string, number];
skills = [5, 'Languages']; //error

vd trong thực tế:
let color: [number, number, number] = [255, 0, 0];

Optional Tuple Elements
let bgColor, headerColor: [number, number, number, number?];
bgColor = [0, 255, 255, 0.5];
headerColor = [0, 255, 255];

#14. Enum Types

1. Enum là gì

Enum (enumerated : liệt kê, one by one) là 1 nhóm các giá trị hằng sô.

ví dụ: STATUS = [PENDING, SUCCESS, FAILED]

Để định nghĩa enum:
- Sử dụng keyword enum, và 'tên' cho enum
- định nghĩa các giá trị hằng số

enum name { constant1, constant2,...};

2. String enum

https://www.typescriptlang.org/docs/handbook/enums.html

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

#15. TypeScript any Type

Giới thiệu về any type

Đôi khi chúng ta dùng biến (variables) để lưu giá trị, nhưng lại không chắc chắn về kiểu dữ liệu của biến đấy (data types).

Trong trường hợp này, chúng ta muốn không bị 'compiler' của typescript báo lỗi khi dịch code (do gán data-type lung tung :D)
=> sử dụng 'any' type

Khi nào sử dụng any type

- Với any type, chúng ta báo hiệu cho compiler 'không check' kiểu giá trị (data-type)

- Migrate từ 1 project Javascript sang Typescript

#16. TypeScript void Type

Giới thiệu về void Type

- Nếu như 'any' Type (không cần biết phải trả về gì, sao cũng được), thì 'void' type sẽ không trả về giá trị gì.

- Thường dùng để ám chỉ '1 function' không trả về giá trị

```
function test(message): void {
  console.log(message);
}
```

Lợi ích của việc dùng void Type:

- Giúp code trong function rõ ràng hơn, khi chúng ta không cần 'soi tìm bằng được' keyword return :v

#17. TypeScript Data Type - Never

1.Giới thiệu về Never

Any: trả về bất cứ thứ gì (áp dụng cho function và variable)
Void: không cần trả ra dữ liệu (thực chất vẫn trả ra :v), không cần keyword return (áp dụng chủ yếu cho function)
Never: 'không bao giờ' trả ra giá trị. promise !

Kiểu dữ liệu 'never' được sử dụng khi chúng ta 'chắc chắn' rằng một điều gì đấy không bao giờ sẽ xảy ra. (không cần trả ra kết quả, giống void ???)

Ví dụ:
- 1 function handle exception (xử lý lỗi):

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

- 1 function chạy không có điểm dừng :v

```
function doingForever() : never {
        while(true){
          console.log("I'm always running...");
        }

}
```

=> **never type được dùng để 'ám chỉ':**
một giá trị sẽ không bao giờ xảy ra (ví dụ về exception error)
hoặc
một giá trị sẽ không bao giờ được trả về (ví dụ về loop infinity)

2.Phân biệt Never và Void

khi sử dụng void cho function, thực ra nó trả ra 'undefined' :v

```
function logging (): void {
        console.log("Server logging...");
}

let check: void = logging();
console.log(">>> check: ", check);//undefined
```

#18. TypeScript union Type

1. Giới thiệu về union Type

```
//union ~ join
EU: European Union

function addNumberOrString(a: any, b: any) {
   if (typeof a === 'number' && typeof b === 'number') {
      return a + b;
   }
   if (typeof a === 'string' && typeof b === 'string') {
      return a.concat(b);
   }
   throw new Error('Parameters must be numbers or strings');
}
```

Nếu a,b đều là number => tính tổng 2 số a và b
Nếu a, b đều là string => cộng gộp chuỗi (nối chuỗi)
Nếu a, b không là number hoặc string => báo lỗi

Nếu chúng ta truyền:
addNumberOrString(true, [1,2,3])
=> typescript dịch code không lỗi (do kiểu dữ liệu any)

=> chạy code thì sẽ có lỗi (throw error)

=> Cách giải quyết: không dùng kiểu dữ liệu any

```
let a: string | number; //union type
a = 10;//ok
a = 'abc';//ok
a = true; //error

----
update functions:
function add(a: number | string, b: number | string) {
   if (typeof a === 'number' && typeof b === 'number') {
      return a + b;
   }
```

```
    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    }
    throw new Error('Parameters must be numbers or strings');
}
```

#19. TypeScript Type Aliases

Type aliases cho phép chúng ta tạo mới 1 kiểu type từ những type đã tồn tại.

Cú pháp: type alias = existingType;

existingType là tất cả những type 'hợp lệ' của TypeScript, như là string, number, boolean...

Ví dụ:

```
type ky_tu = string;
let message: ky_tu;// same as string type
```

Sử dụng aliases rất hữu ích với union type:

before:

```
let myVar: string|number|boolean = 'just a test';
myVar = { abc: 'def' }; //error
```

after:

```
type superType = string|number|boolean;
let myVar: superType = 'just a test';
```

Chapter 3: Control Flow Statements

#20. TypeScript if else

1.TypeScript if statement

```
if(condition) {
  // if-statement
}
```

Khối code if sẽ được thực thi dựa vào điều kiện. Nếu điều kiện là 'true', nó sẽ được thực thi.

vd:

```
let age = 25;

if(age > 18) {
 console.log("You can watch JAV");
}
```

```
if(condition)// truyền vào 1 biến thì sao ?
{
  // if-statement
}
```

khi condition là 1 biến, TS/JS tự động convert nó sang boolean
vd:

```
let name = 'abc';
if(name){ console.log(`your name 's ${name}`);
```

2.TypeScript if...else statement

```
if(condition) {
  // if-statements
} else {
 // else statements;
}
```

Điều kiện else sẽ xảy ra khi condition = false

if(age> 18) {} else {}

if else mới giải quyết được 2 điều kiện, vậy >2 điều kiện thì sao ?

3. if if if if...

```
let discount: number;
let itemCount = 11;

if (itemCount > 0 && itemCount <= 5) {
   discount = 5;  // 5% discount
} else if (itemCount > 5 && itemCount <= 10) {
   discount = 10; // 10% discount
} else {
   discount = 15; // 15%
}

console.log(`You got ${discount}% discount. `);
```

#21. TypeScript switch case

1. Cú pháp:

```
switch ( expression ) {
  case value1:
     // statement 1
     break;
  case value2:
     // statement 2
     break;
  case valueN:
     // statement N
     break;
  default:
     //
     break;
}
```

switch... case sẽ tính toán giá trị của 'expression'.
Nó sẽ tìm 'case'(trường hợp) đầu tiên thỏa mãn expression (value1, value2,..., valueN)

switch...case sẽ thực thi "only" - mình case, mà giá trị của nó thỏa mãn
trong trường hợp không có case thỏa mãn => giá trị default sẽ được sử dụng

keyword break đảm bảo rằng 'chỉ có duy nhất' 1 case được thực hiện

2. ví dụ

```
let targetId = 'btnDelete';

switch (targetId) {
   case 'btnUpdate':
      console.log('Update');
      break;
   case 'btnDelete':
      console.log('Delete');
      break;
   case 'btnNew':
      console.log('New');
      break;
}
```

ví dụ về group case: (waterfall)

```
// change the month and year
let month = 2,
    year = 2020;

let day = 0;
switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        day = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        day = 30;
        break;
    case 2:
        // leap year
        if (((year % 4 == 0) &&
            !(year % 100 == 0))
            || (year % 400 == 0))
            day = 29;
        else
            day = 28;
        break;
    default:
        throw Error('Invalid month');
}

console.log(`The month ${month} in ${year} has ${day} days`);
```

#22. TypeScript for

1.Cú pháp for loop

Chúng ta có thể dùng if/else or switch/case, tuy nhiên, với vòng lặp, nó giúp code ngắn hơn và thực thi nhiều lần

```
for(initialization; condition; expression) {
   // statement
}
```

initialization: giá trị khởi tạo (input đầu vào) của vòng lặp
condition: điều kiện để thực thi vòng lặp. Nếu điều kiện là true, khối code sẽ được thực thi. Ngược lại khi condition = false, vòng lặp sẽ dùng (không chạy code nữa)

expression: sử dụng để update vòng lặp. Mỗi 1 lần vòng lặp được chạy, khối code trong body được thực thi đến cuối, thì expression sẽ được chạy để chuẩn bị cho lần chạy kế tiếp

2. Example

```
for(initialization; condition; expression);
```

```
for (let i = 0; i < 10; i++) {
   console.log(">> i= ", i);
}
```

initialization: i = 0
condition: i< 10
expression: i++

- Chúng ta khởi tạo biến i với giá trị = 0: i = 0
Sau đấy, kiểm tra điều kiện: i < 10 hay không.

Nếu i < 10, thì console.log giá trị
đồng thời, thực thi expression: i++ để tăng i lên 1 đơn vị

- Thực thi cho tới khi điều kiện không còn thỏa mãn

3. optional block

- Lưu ý không nên sử dụng việc viết tắt, đừng viết kiểu 'đố bạn mình viết gì'.
Có chắc gì 'bây giờ bạn viết code bạn hiểu, 1 tháng sau quay lại, liệu bạn có hiểu :v'

```
let i = 0;
for (; i < 10; i++) {
   console.log(i);
}


for (let i = 0; ; i++) {
   console.log(i);
   if (i > 9) break; // cần có if/break để không khiến vòng lặp chạy vô hạn
}


let i = 0;
for (; ;) {
   console.log(i);
   i++;
   if (i > 9) break;
}
```

#23. TypeScript while

1. Giới thiệu về vòng lặp while

- Tương tự for, vòng lặp while giúp chúng ta có thể viết code khác đi 'if/else', switch/case

- Nếu như với vòng lặp for, chúng ta sẽ biết 'nên chạy' vòng lặp bao nhiêu lần (1 con số cụ thể), thì với while, chúng ta có thể chạy với số lần không cần biết trước

```
while(condition) {
   // do something
}
```

Nếu như điều kiện = true => thực thi khối code, else.. do nothing

- Để thoát khỏi vòng lặp (vì mặc định vòng lặp sẽ chạy không giới hạn), cần dùng keyword break;

```
while(condition) {
   // do something
   // ...

   if(anotherCondition)
      break;
}
```

2. example

```
let counter = 0;

while (counter < 5) {
   console.log(counter);
   counter++;
}
```

```
let counter = 0;

while (counter < 5) {
   console.log(counter);
   if(counter % 2 === 0) break;
   counter++;
}
```

#24. TypeScript do...while

1.Cú pháp

```
do {
   // do something
} while(condition);
```

```
let i = 0;

do {
   console.log(i);
   i++
} while (i < 10);
```

2. Sự khác biệt giữa do...while và while

- do...while luôn chạy ít nhất 1 lần

```
let i = 20;

do {
   console.log(">>> i= ", i);
   i++;
} while (i < 10);
```

#25. TypeScript break

1. Sử dụng TypeScript break để kết thúc vòng lặp

Chúng ta có thể sử dụng keyword break bên trong vòng lặp for, while và do..while

```
let products = [
   { name: 'phone', price: 700 },
   { name: 'tablet', price: 900 },
   { name: 'laptop', price: 1200 }
];

for (let i = 0; i < products.length; i++) {
   if (products[i].price == 900)
      break;
}

// show the products
console.log(products[i]);
```

2. Sử dụng break bên trong switch...case

```
let products = [
   { name: 'phone', price: 700 },
   { name: 'tablet', price: 900 },
   { name: 'laptop', price: 1200 }
];

let discount = 0;
let product = products[1];
switch (product.name) {
   case 'phone':
      discount = 5;
      break;
   case 'tablet':
      discount = 10;
      break;
   case 'laptop':
      discount = 15;
      break;
}
console.log(`There is a ${discount}% on ${product.name}.`);
```

#26. TypeScript continue

Tương tự break, continue được sử dụng trong các vòng lặp for, while và do...while

Nếu như break được sử dụng để kết thúc 'nguyên/cả' vòng lặp, thì continue báo hiệu bỏ quả 'mình lần chạy' body của vòng lặp . vòng lặp không thực thi lần chạy đấy, mà tiếp tục chạy lần tiếp theo.

```
for (let index = 0; index < 9; index++) {

   // if index is odd, skip it
   if (index % 2)
      continue;

   // the following code will be skipped for odd numbers
   console.log(index);
}
=========

let index = -1;

while (index < 9) {
   index = index + 1;
   if (index % 2)
      continue;
   console.log(index);
}

============
let index = 9;
let count = 0;

do {
   index += 1;

   if (index % 2)
      continue;
   count += 1;
} while (index < 99);
console.log(count); // 45
```

Chapter 4: Functions

#27. TypeScript Functions

1. Giới thiệu về functions

Function chính là tập hợp của 1 khối code có thể: đọc/dùng và tái sử dụng được.

Tương tự như javascript, điểm khác biệt của TS là nó quy định (ép buộc)
của type của biến (variable) và function

before:

```
function name(parameter1, parameter2,...){
  // do something
}
```

after

```
function name(parameter1: type, parameter2:type,...): returnType {
  // do something
}
```

ví dụ:

```
function add(a: number, b: number): number {
   return a + b;
}

let sum = add('10', '20'); //error
```

2. Sử dụng với arrow function

```
let sum = (x: number, y: number): number => {
   return x + y;
}

sum(10, 20); //returns 30
```

#28. TypeScript Function Types

1. syntax

(parameter1: **type**, parameter2:**type**,...) => **type**

1 Function Type gồm 2 phần: Type của tham số (parameters) và kiểu giá trị/type trả về của hàm (return type)

let add: (x: number, y: number) => number;

- Type của tham số: x và y là number
- Return type: number => hàm này sẽ trả ra number

2. Inferring function types

let add: (x: number, y: number) => number;

TS tự đoán type ???

#29. TypeScript Optional Parameters

Với Javascript, chúng ta có thể gọi 1 functions mà không cần truyền vào input đầu vào, mặc dù functions đấy cần tham số đầu vào.
VD:

```
function test(x,y,z) {
        console.log("x = ", x , " y = ", y , "z = ", z);
}

test() // run ok without any errors
```

Với Typescript, nếu làm giống ví dụ trên sẽ lỗi vì: TS dịch code và:
- Kiểm tra số lượng tham số cần truyền vào cho 1 functions
- Kiểu dữ liệu (types) truyền vào từng tham số

Vậy làm sao để có 'fix' trường hợp trên ???

Optional Parameters:
Để sử dụng, chúng ta dùng dấu ? sau khi khai báo tên tham số

```
function multiply(a: number, b: number, c?: number): number {

   if (typeof c !== 'undefined') {
      return a * b * c;
   }
   return a * b;
}
```

=> trường hợp không truyền vào tham số => JS/TS sẽ gán giá trị 'undefined'

```
function multiply(a: number, b?: number, c: number): number {

   if (typeof c !== 'undefined') {
      return a * b * c;
   }
   return a * b;
}
```
=> lưu ý khi check code

#30. TypeScript Default Parameters

Nếu như optional Parameter cho phép chúng 'truyền' hoặc 'không truyền' input đầu vào cho functions,

và trong trường hợp 'không truyền', giá trị chúng ta nhận được là undefined

=> đôi khi chúng ta muốn 1 giá trị mặc định (khác undefined) trong trường hợp đấy :v

Với javascript (từ ES6)

```
function name(parameter1=defaultValue1,...) {
  // do something
}

function applyDiscount(price, discount = 0.05) {
   return price * (1 - discount);
}

console.log(applyDiscount(100)); // 95
console.log(applyDiscount(100, 0.99)); //
```

với TypeScript:

```
function name(parameter1:type=defaultvalue1, parameter2:type=defaultvalue2,...) {
  //
}

function applyDiscount(price: number, discount: number = 0.05): number {
   return price * (1 - discount);
}

console.log(applyDiscount(100)); // 95
```

Link stackoverflow:
https://stackoverflow.com/questions/53627516/typescript-cant-set-default-parameter-value-as-false

#31. TypeScript Rest Parameters

Rest: phần còn lại

Với TypeScript, Rest parameters có các rules sau:

- 1 function chỉ có **1 tham số duy nhất** rest parameter
- phải là tham số **cuối cùng** trong danh sách tham số
- phải sử dụng với **array type**

cú pháp:

```
function fn(...rest: type[ ]) {
  //...
}
```

```
function getTotal(...numbers: number[ ]): number {
   let total = 0;
   numbers.forEach((num) => total += num);
   return total;
}

console.log(getTotal()); // 0
console.log(getTotal(10, 20)); // 30
console.log(getTotal(10, 20, 30)); // 60
```

```
function multiply(n: number, ...m: number[ ]) {
 return m.map((x) => n * x);
}
// 'a' gets value [10, 20, 30, 40]
const a = multiply(10, 1, 2, 3, 4);
```

```
function Greet(greeting: string, ...names: string[]) {
   return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```

#32. TypeScript Function Overloading

- Lưu ý: Overloadings của Typescript khác với các ngôn ngữ lập trình khác,ví dụ C# hay java

```
function addNumbers(a: number, b: number): number {
   return a + b;
}

function addStrings(a: string, b: string): string {
   return a + b;
}

// 2 functions làm nhiệm vụ tương tự nhau => có thể gồm thành 1 với union type
function add(a: number | string, b: number | string): number | string {
   if (typeof a === 'number' && typeof b === 'number')
      return a + b;

   if (typeof a === 'string' && typeof b === 'string')
      return a + b;
}
```

Với union type, chúng ta không thấy được 'mối quan hệ' giữa các loại datatype, đồng thời vẫn bị 'lặp' code

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
  return a + b;
}
```

Lưu ý cách hoạt động của TS overloading không giống các ngôn ngữ khác, ví dụ:

```
class MethodOverloading {
   private static void display(int a){
      System.out.println("Arguments: " + a);
   }

   private static void display(int a, int b){
      System.out.println("Arguments: " + a + " and " + b);
   }

   public static void main(String[] args) {
      display(1);
      display(1, 4);
   }
}
```

---

method overloading:

```
class Counter {
   private current: number = 0;
   count(): number;
   count(target: number): number[];
   count(target?: number): number | number[] {
      if (target) {
         let values: number[] = [];
         for (let start = this.current; start <= target; start++) {
            values.push(start);
         }
         return values;
      }
      return ++this.current;
   }
}

let counter111 = new Counter();

console.log(counter111.count()); // return a number
console.log(counter111.count(20)); // return an array
```

Chapter 5: Classes

#33. Classes

Với javascript, khái niệm 'class' nó không giống hoàn toàn với các ngôn ngữ khác, ví dụ C# và Java

Để tạo 1 class, chúng ta có thể dùng cách sau: (với version trước ES6)

```
function Person(ssn, firstName, lastName) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.getFullName = function () {
    return `${this.firstName} ${this.lastName}`;
}
```

https://www.w3schools.com/js/js_object_prototypes.asp

--

để sử dụng :

```
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

với ES6, chúng ta có thể dùng keyword 'class'

```
class Person {
    ssn;
    firstName;
    lastName;

    constructor(ssn, firstName, lastName) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }
```

```
}
//sử dụng
let person = new Person('171-28-0926','John','Doe');
console.log(person.getFullName());
```

Sử dụng với typescript

```
class Person {
    ssn: string;
    firstName: string;
    lastName: string;

    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

#34. TypeScript Access Modifiers

Access modifiers cung cấp công cụ về quyền truy cập và sử dụng biến/functions với class.
Typescript cung cấp 3 Access modifiers:
- public
- private
- protected

1. Public modifier

Default, tất cả mọi thứ (variables/functions) bên trong class là public. chúng ta có thể truy cập chúng mà không bị giới hạn gì

=> không cần khai báo keyword public

```
class Employee {
    public empCode: string;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123;
emp.empName = "Swati";
```

empCode and empName được khai báo public.
vì vậy, chúng ta có thể truy cập chúng bên ngoài class khởi tạo ban đầu.

2. Private

- giới hạn quyền truy cập biến/function ở trong cùng class. Bên ngoài class => error

```
//khai báo
class Person {
    private ssn: string;
    private firstName: string;
    private lastName: string;
    // ...
}
```

```
class Person {
    private ssn: string;
    private firstName: string;
    private lastName: string;

    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}

let person = new Person('153-07-3130', 'John', 'Doe');
console.log(person.ssn); // compile error
```

3. protected

Giống private, tuy nhiên, lớp con (kế thừa) sẽ không thể truy cập được

```
class Employee {
    public empName: string;
    protected empCode: number;

    constructor(name: string, code: number){
        this.empName = name;
        this.empCode = code;
    }
}
```

```
class SalesEmployee extends Employee{
    private department: string;

    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.department = department;
    }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
emp.empCode; //Compiler Error
```

#35. TypeScript readonly

readonly => chỉ đọc => KHÔNG MODIFY (UPDATE/DELETE)

Typescript cung cấp công cụ 'readonly' để đánh dấu 1 một thuộc tính **'immutable'**

```
class Person {
   readonly birthDate: Date;

   constructor(birthDate: Date) {
      this.birthDate = birthDate;
   }
}

let person = new Person(new Date(1990, 12, 25));
person.birthDate = new Date(1991, 12, 25); // Compile error
```

2. Readonly vs const

readonly: sử dụng cho thuộc tính của class
const: sử dụng cho variables

#36. Typescript Getters and Setters

```
class Person {
   public age: number;
   public firstName: string;
   public lastName: string;
}

let person = new Person();
person.age = 26;
---
Nếu như cần validate biến age , thì làm thế nào ?
if( inputAge > 0 && inputAge < 200 ) {
   person.age = inputAge;
}
- với kiểu dữ liệu private => không thể truy cập

1. Setter và Getter
class Person {
   private _age: number;
   private _firstName: string;
   private _lastName: string;

   public get age() {
      return this._age;
   }

   public set age(theAge: number) {
      if (theAge <= 0 || theAge >= 200) {
         throw new Error('The age is invalid');
      }
      this._age = theAge;
   }
   public getFullName(): string {
      return `${this._firstName} ${this._lastName}`;
   }
}

let person = new Person();
person.age = 10; //setter
```

Ví dụ:

```
class Person {
    private _age: number;
    private _firstName: string;
    private _lastName: string;

    public get age() {
        return this._age;
    }

    public set age(theAge: number) {
        if (theAge <= 0 || theAge >= 200) {
            throw new Error('The age is invalid');
        }
        this._age = theAge;
    }
    public get firstName() {
        return this._firstName;
    }
    public set firstName(theFirstName: string) {
        if (!theFirstName) {
            throw new Error('Invalid first name.');
        }
        this._firstName = theFirstName;
    }

    public get lastName() {
        return this._lastName;
    }

    public set lastName(theLastName: string) {
        if (!theLastName) {
            throw new Error('Invalid last name.');
        }
        this._lastName = theLastName;
    }

    public getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    } }
```

#37 TypeScript Inheritance

1. Giới thiệu về Typescript inheritance

Một class có thể tái sử dụng lại thuộc tính (properties) và method của class khác.
Đây gọi là **kế thừa**.
Tương tự việc 'con cái' thừa hưởng lại tài sản (properties/method) 'cha mẹ' để lại.

Bằng cách sử dụng tính năng kế thừa, chúng ta không cần phải 'code lại' class đã có sẵn

```
class Person3 {
    constructor(private firstName: string, private lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
    describe(): string {
        return `This is ${this.firstName} ${this.lastName}.`;
    }
}

//để kế thừa 1 class, chúng ta sử dụng keyword extends

class Employee1 extends Person3 {
    private jobTitle;
    constructor(
        firstName: string,
        lastName: string,
        jobTitle: string) {

        // call the constructor of the Person class:
        super(firstName, lastName);
        this.jobTitle = jobTitle;
    }
}

// let employee = new Employee('John','Doe','Front-end Developer');

//Employee kết thừa lại person => dùng đc method của parent
```

```
let employee = new Employee1('Hoi Dan IT', 'Eric', 'Web Developer');

console.log(employee.getFullName());
console.log(employee.describe());
```

2. Method overriding

- định nghĩa method trong class child => ghi đè lại cha
- nếu muốn sử dụng class cha => super

```
class Employee extends Person {
    constructor(
        firstName: string,
        lastName: string,
        private jobTitle: string) {

        super(firstName, lastName);
    }

    describe(): string {
        return super.describe() + `I'm a ${this.jobTitle}.`;
    }
}

let employee = new Employee('John', 'Doe', 'Web Developer');
console.log(employee.describe());
```

#38. TypeScript Static Methods and Properties

Để truy cập 1 thuộc tính static: <ClassName>.< StaticMember>
chúng ta sử dụng class name và dot notation, mà không cần 'tạo mới' 1 object

```
class Circle {
   static pi: number = 3.14;

   static calculateArea(radius:number) {
      return this.pi * radius * radius;
   }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5

//static and non static
class Circle {
   static pi = 3.14;
   pi = 3;
}

Circle.pi; // returns 3.14

let circleObj = new Circle();
circleObj.pi; // returns 3



class Circle {
   static pi = 3.14;

   static calculateArea(radius:number) {
      return this.pi * radius * radius;
   }

   calculateCircumference(radius:number):number {
      return 2 * Circle.pi * radius;
   }
}
```

```
Circle.calculateArea(5); // returns 78.5

let circleObj = new Circle();
circleObj.calculateCircumference(5) // returns 31.4000000
//circleObj.calculateArea(); <-- cannot call this
```

#39 TypeScript abstract classes
abstract: trừu tượng

Cú pháp:
abstract class Employee { //...}

1 abstract class có thể chứa:
- 1 hoặc nhiều 'normal' method
- 1 hoặc nhiều **abstract method:** đối với loại này, không chứa phần body, chỉ định nghĩa tên method

```
abstract class Employee {
   constructor(private firstName: string, private lastName: string) {
   }
   abstract getSalary(): number ; //abstract method

   //normal method
   get fullName(): string {
      return `${this.firstName} ${this.lastName}`;
   }
   compensationStatement(): string {
      return `${this.fullName} makes ${this.getSalary()} a month.`;
   }
}
```

Với abstract class, chúng ta không thể tạo mới 1 object với nó

```
const e = new Employee("Eric", "Hỏi Dân IT");// Error
```

=> dùng kế thừa để sử dụng abstract class.

```
class FullTimeEmployee extends Employee {
   constructor(firstName: string, lastName: string, private salary: number) {
      super(firstName, lastName); //dùng super để kế thừa lại cha
   }

  // cần viết method này, vì nó được khai báo abstract ở trên
   getSalary(): number {
      return this.salary;
   }
}



class Contractor extends Employee {
   constructor(firstName: string, lastName: string, private rate: number, private hours:
number) {
      super(firstName, lastName);
   }
   getSalary(): number {
      return this.rate * this.hours;
   }
}

let e1 = new FullTimeEmployee('Eric', 'Hoi Dan IT', 12000);
let e2 = new Contractor('Eric', 'Hoi Dan IT', 100, 160);

console.log(john.compensationStatement());
console.log(jane.compensationStatement());
```

Chapter 6: Interfaces

#40. Interfaces

1. Giới thiệu về interfaces

TypeScript Interfaces 'định nghĩa' cách chúng ta viết code, bằng cách cung cấp các công cụ để có thể kiểm soát chặt chẽ việc khai báo 'type'

Ví dụ:

```
function getFullName(person: {
    firstName: string;
    lastName: string
}) {
    return `${person.firstName} ${person.lastName}`;
}

let person = {
    firstName: 'Eric',
    lastName: 'Hoi Dan IT'
};

console.log(getFullName(person)); // Eric Hoi Dan IT
```

Trong ví dụ trên check các tham số truyền vào hàm getFullName(). Nếu các thuộc tính truyền vào khác type 'string', TS compiler sẽ báo lỗi.

Với cách viết ở trên : code sẽ dài và khó đọc nếu như truyền nhiều tham số (person)/object person có nhiều thuộc tính ?

Để giải quyết vấn đề trên, **TS cung cấp type annotation với tên gọi Interface**

2. Cú pháp.

Để định nghĩa object Person có 2 thuộc tính ở trên:

```
interface Person {
    firstName: string;
    lastName: string;
}
```

- Convention:
+ Luôn bắt đầu = chữ hoa, và sử dụng camelcase, tức là luôn viết hoa chữ cái đầu tiên của từ
Ví dụ: Person, UserProfile...

+ Ngoài ra, có thể bắt đầu bằng chữ I (viết tắt của Person) để cho tường minh (Recommend)
ví dụ: IPerson, IUserProfile

sau khi định nghĩa interface:
```
function getFullName(person: Person) {
    return `${person.firstName} ${person.lastName}`;
}

let john = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(john));
```

Nếu khai báo thừa thuộc tính thì sau, ví dụ:
```
let jane = {
  firstName: 'Jane',
  middleName: 'K.'
  lastName: 'Doe',
  age: 22
};
```
=> không báo lỗi, chẳng qua không gợi ý code :v

3. Optional properties
Sử dụng dấu ? cuối tên của thuộc tính để báo hiệu 1 thuộc tính không bắt buộc cần khai báo.

```
interface Person {
    firstName: string;
    middleName?: string; //optional
    lastName: string;
}
```

```
--update
function getFullName(person: Person) {
    if (person.middleName) {
        return `${person.firstName} ${person.middleName} ${person.lastName}`;
    }
    return `${person.firstName} ${person.lastName}`;
}
```

4. Readonly properties

readonly giúp chúng ta báo hiệu 1 thuộc sẽ không thể 'sửa đổi' sau khi đã được khai báo (immutable)

```
interface Person {
    readonly ssn: string;
    firstName: string;
    lastName: string;
}

let person: Person;
person = {
    ssn: '171-28-0926',
    firstName: 'John',
    lastName: 'Doe'
}

person.ssn = '171-28-0000'; //error
```

So sanh type và interface:
https://stackoverflow.com/questions/37233735/interfaces-vs-types-in-typescript

#41.How to extend interfaces in Typescript

1. Sử dụng với class

Extends : class extends class, interface extends interface

Implements: class **i**mplements **i**nterface

--

Class kế thừa lại interface để có những thuộc tính và functions 'bắt buộc' phải có, ngoài ra có thể định nghĩa thêm các class/method riêng của nó.

```
interface IEmployee {
   empCode: number;
   name: string;
   getSalary:(empCode: number) => number;
}

class Employee implements IEmployee {
   empCode: number;
   name: string;

   constructor(code: number, name: string) {
      this.empCode = code;
      this.name = name;
   }

   getSalary(empCode:number):number {
      return 20000;
   }
}

let emp = new Employee(1, "Steve");
```

2. Interfaces extendings 1 interface

Ví dụ chúng ta có 1 interface, tên là 'Mailable', gồm có 2 method: send() và queue()

```
interface Mailable {
        send(email: string): boolean,
        queue(email: string): string
//coding ….
}
```

và chúng ta đã định nghĩa 'rất nhiều classes' kế thừa lại interface đấy

```
class A implements Mailable {...}
class B implements Mailable {...}

Class D
```

Yêu cầu phát sinh: chúng ta thêm mới method later() vào interface Mailable => gây ra lỗi, vì cần định nghĩa method này ở 'tất cả' những class kế thừa nó.

Để giải quyết vấn đề trên, chúng ta có thể tạo mới 1 interface, kế thừa lại interface đã có

--- các interface có thể kế thừa lại lẫn nhau => cộng gộp data

```
interface FutureMailable extends Mailable {
   later(email: string, after: number): boolean
}
```

Sử dụng keyword 'extends' để kế thừa lại.

Như vậy, với bài toán ban đầu của chúng ta, tất cả những class nào cần sử dụng 'method' mới (later()),
thì chỉ cần sửa code từ implements interface Mailable sang interface FutureMailable

3. interfaces extending multiple interfaces

1 interface có thể kế nhiều interface khác, tạo ra 1 tổ hợp của các interfaces

```
interface B {
   b(): void
}

interface C {
   c(): void
}

interface D extends B, C {
   d(): void
         b(): void
         c(): void

}
```

Ở ví dụ này, interface D kế thừa lại interface B và C => D có tất cả method của B và C

Chapter 7: Advanced Types

#43. Intersection Types

1. Giới thiệu về intersection types

Với union type:(toán tử or)

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

Idea:

- Universe values:
number + string
=> tất cả các phần tử thỏa mãn 1 trong 2 điều kiện trên đều ok => cộng gộp giá trị

Dựa vào giá trị, Typescript sẽ 'gợi ý' code như thế nào

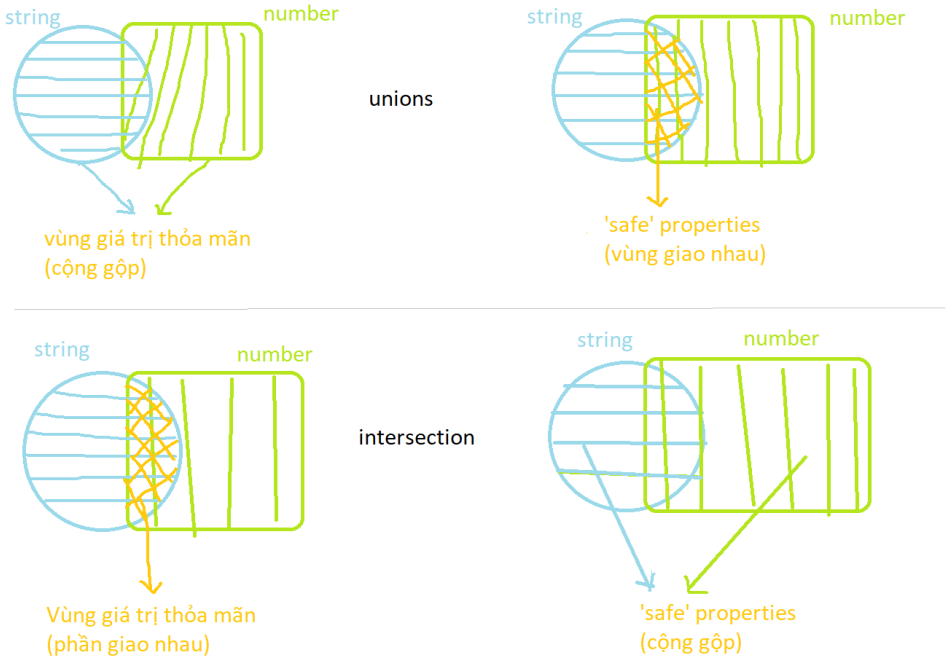
- Safe properties:
Typescript chỉ có thể 'gợi ý' những giá trị nào, thuộc cả 2 loại trên (phần giao nhau)

Nếu var có datatype là number => không có các method của string
Nếu var có datatype là string => không có các method của number

=> Cần có thêm 'gợi ý' cho typescript biết chúng ta đang sử dụng loại datatype nào (ví dụng sử dụng typeof)

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
// khi sử dụng thuộc tính 'id', typescript không thể gợi ý, vì không biết là 'number' hay string
// ts chỉ biết, khi và chỉ khi:
// 1. dựa vào giá trị input truyền vào. ví dụ: printId(10) => suy luận: 10 là number -> biết các method của number
// 2. chúng ta nói cho nó biết :v

//update code
if (typeof id === 'number) {//ts dựa vào đây để biết id là number -> gợi ý method number}
if(typeof id === 'string'){//...}

// ví dụ về union => gợi ý 'safe' properties
function doSth(val: string | number) {
    console.log("the primitive value of a string = ", val.valueOf())
}

doSth(10)
doSth("just a text");

----
Với intersections: (toán tử &)

interface Person {
firstName: string,
lastName: string
}

interface Printable {
 printFullName: () => void;
}

const person: Person & Printable; //intersections


- Universe Values:
-> chỉ allow các giá trị thuộc Person và Printable (điều kiện và &)

- Safe properties => all values
```

```
let mine: Person & Printable = {
firstName: "abc",
lastName: "def",
printFullName: ()=> { `console.log(${this.firstName} - ${this.lastName})`}
}


let myFunc = ( mine: Person & Printable) => {
        mine. //show all the properties from all interfaces
}
```



--- Type Order
khi sử dụng intersections, thứ tự type không quan trọng
type typeAB = typeA & typeB;
type typeBA = typeB & typeA;

=> typeAB và typeBA có các thuộc tính giống hệt nhau