# LOẠI CƠ BẢN

## Typescript là gì?

Là ngôn ngữ lập trình mở rộng của JavaScript, bổ xung các tính năng kiểu tĩnh và nhiều tính năng nâng cao giúp kiểm soát lỗi và phát triển ứng dụng dễ dàng hơn.

Lợi ích:

* Phát hiện lỗi ngay từ khi viết code thay vì đợi đến lúc chạy.
* Cải thiện đọc hiểu và bảo trì code.

## Kiểu chú thích

Chỉ định kiểu dữ liệu của biến, hàm, đối tượng.

Cú pháp:

: kiểu dữ liệu

VD1:

let counter: number;

counter = 1; // đúng

counter = "Hello"; // sai, biên dịch lỗi

VD2: Mảng.

let names: string[] = ['John', 'Jane', 'Peter', 'David', 'Mary'];

VD3: Đối tượng.

let person: {

name: string;

age: number;

};

person = {

name: 'John',

age: 25,

}; // valid

VD4: Hàm.

const sayHello = (name: string): string => {

return `Hello, ${name}!`

}

Có thể quy định 1 biến có 2 hoặc nhiều kiểu dữ liệu ngăn cách nhau bởi |

const userId = string | number.

## Kiểu suy luận

VD:

let counter = 0;

Tương đương với:

let counter: number = 0;

Đối với hàm nếu không có return sẽ mặc định quy định kiểu hàm là void, còn nếu có trả về thì nó sẽ dựa trên kiểu dữ liệu của return mà quy định kiểu trả về.

Đối với kiểu suy luận về cơ bản giống javascript, nó vẫn chạy được trong hầu hết các trường hợp, tuy nhiên đối với dự án lớn, đòi hỏi mức độ chính xác cao cần nên chỉ định kiểu.

## Kiểu số

Dùng kiểu : number cho tất cả các kiểu số, ngoài ra đối với số lớn hơn 253 – 1 thì dùng : bigint.

let price: number;

let big: bigint = 9007199254740991n;

## Kiểu chuỗi

Dùng kiểu : string, cho kiểu dữ liệu chuỗi.

let firstName: string = 'John';

let title: string = "Web Developer";

let profile: string = `I'm ${firstName}`;

## Kiểu đúng sai

Dùng kiểu : Boolean.

let completed: boolean = true;

## Kiểu đối tượng

Là kiểu đại diện cho tất cả giá trị không thuộc loại kiểu nguyên thủy.

Kiểu nguyên thủy: number, bigint, string, boolean, null, undefined, symbol

VD:

let employee: object;

employee = {

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

};

## Kiểu mảng

Cú pháp:

let arrayName: type[];

VD:

let skills: string[] = [];

let scores : (string | number)[];

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

## Kiểu Tuple

Hoạt động giống như mảng nhưng sẽ có thêm vài điều kiện.

VD:

let skill: [string, number];

skill = ['Programming', 5];

Nếu thay đổi thứ tự, chương trình sẽ lỗi.

Khai báo biến tùy chọn, đôi khi bạn không chắc chắn hoặc tham số đó có thể có hoặc không bạn có thể dùng ? để quy định nó:

let skill: [string, number?];

skill = ['Programming', 5]; // đúng

skill = ['Programming']; // đúng

// Tham số thứ 2 bắt buộc là kiểu number và có giá trị hoặc không đều không lỗi.

## Kiểu enum

Là đặt tên của một nhóm các giá trị hằng số (viết tắt của enumerated).

Cú pháp:

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

VD:

enum Month {

Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,

}

console.log(Month.Jan);

// 0

enum Status {

Pending = 1, // Bắt đầu từ 1

InProgress = 3,

Completed = 5

}

console.log(Status.Pending); // 1

console.log(Status.Completed); // 5

Tuy nhiên enum thường không được sử dụng mấy trong dự án thực tế vì nó có khá nhiều nhược điểm:

* Một tập hợp nhỏ có các giá trị cố định.
* Giá trị này không được thay đổi

Nên sử dụng type thay vì enum.

## Kiểu dữ liệu any

Là kiểu dữ liệu bất kỳ, typescript sẽ bỏ qua bắt lỗi đối với biến này, tức là biến này có thể gán bất kì giá trị nào, chỉ nên sử dụng trong trường hợp không biết trước hoặc chấp nhận mọi giá trị chấp nhận rủi ro.

VD:

let variable: any;

let variable; // mặc định không gán kiểu hoặc giá trị sẽ là any

## Kiểu unknown

Cũng giống như kiểu any, bạn có thể gắn bất kì giá trị gì cho nó, ví dụ bạn không thể gọi một hàm hoặc toán tử trên giá trị unknown.

VD:

let result: unknown;

result = [1,2,3];

const total = result.reduce((a: number, b:number ) => a + b, 0);

console.log(total);

Vì nó không phải mảng number nên bạn không thể sử dụng hàm reduce hay bất kì hàm nào bạn có thể ép kiểu.

VD:

let result: unknown;

result = [1, 2, 3];

const total = (result as number[]).reduce((a: number, b: number) => a + b, 0);

console.log(total); // 6

So sánh sự khác biệt giữa any và unknown.

|  |  |  |
| --- | --- | --- |
| Tính năng | any | unknown |
| Mức độ an toàn | Không an toàn | An toàn |
| Thực thi | Thực thi không cần kiểm tra | Cần phải xác nhận loại trước khi thực thi |
| Trường hợp sử dụng | Sử dụng cho các giá trị động nhưng không an toàn | Sử dụng cho các giá trị động và an toàn nhưng cần xác thực trước khi sử dụng |
| Kiểm soát lỗi | Không | Có |
| Trường hợp sử dụng | Khi chuyển basecode từ JS sang TS | Khi lấy dữ liệu từ nguồn ngoài (gọi API, …) |

## Kiểu void

Dùng để khai báo cho hàm việc không trả về bất kì giá trị gì, hàm đơn giản chỉ thực thi và kết thúc.

function log(message): void {

console.log(messsage);

}

Tuy nhiên nó gán cho nó return cũng sẽ không gây lỗi nhưng có thể tiềm ẩn rủi ro không kiểm soát lỗi được.

## Kiểu never

Kiểu never là chỉ những giá trị không bao giờ xảy ra, một hàm có kiểu trả về là nerver sẽ không kết thúc bình thường.

Trường hợp sử dụng:

* Một hàm xác định luôn throw lỗi.

function throwError(message: string): never {

throw new Error(message);

}

* Hàm là vòng lặp vô hạn.

function infiniteLoop(): never {

while (true) {

console.log("Running forever...");

}

}

## Kiểu union

Là kiểu có thể nhận nhiều kiểu dữ liệu khác nhau bằng việc sử dụng dấu gạch đứng | .

let id: number | string;

id = 101; // Hợp lệ

id = "USER101"; // Hợp lệ

id = true; // Lỗi: boolean không nằm trong union type

## Kiểu dữ liệu chuỗi kí tự

Là kiểu chỉ gán duy nhất cho 1 chuỗi kí tự.

VD:

let click: 'click';

click = 'click'; // Hợp lệ

click = 'dblclick'; // Không hợp lệ

## Kiểu biệt danh

Cho phép bạn tạo một tên mới dựa kiểu dữ liệu đã tồn tại. Thường dùng trong những hệ thống phức tạp nhiều loại trùng tên nhau nhưng khác nội dung nên có thể đặt tên mới để phân biệt chúng giúp dễ đọc hơn và tránh xung đột.

* Kiểu nguyên thuỷ.

type Name: string;

let firstName: Name;

let lastName: Name;

* Kiểu đối tượng.

type Person = {

name: string;

age: number;

};

let person: Person = {

name: 'John',

age: 25

};

* Kiểu union.

type alphanumeric = string | number;

let input: alphanumeric;

input = 100; // valid

input = 'Hi'; // valid

input = false; // Compiler error

* Kiểu giao nhau.

type Personal = {

name: string;

age: number;

};

type Contact = {

email: string;

phone: string;

};

type Candidate = Personal & Contact;

let candidate: Candidate = {

name: "Joe",

age: 25,

email: "joe@example.com",

phone: "(408)-123-4567"

};

# KIỂM SOÁT LUỒNG TRẠNG THÁI

Phần này tương tự như trong Javascript.

* if else
* switch case
* for
* while
* do while
* break
* continue

# HÀM

## Hàm

Hàm là khối code có thể đọc được, bảo trì và tái sử dụng, cũng giống như javascript, tuy nhiên typescript sẽ yêu cầu kiểu dữ liệu.

VD: Trong Javascript:

function sum(a, b) {

return a + b;

}

Còn trong Typescript

function sum(a: number, b: number): number {

return a + b;

}

## Kiểu dữ liệu hàm

Là quy định kiểu trả về của hàm.

VD: : number

function sum(a: number, b: number): number {

return a + b;

}

Hàm sẽ có kiểu trả về là number.

## Tham số tuỳ chọn

Đối với tham số có thể có hoặc không, ta dùng dấu ? để xác định nó có thể có hoặc không.

VD:

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

if (typeof c !== 'undefined') {

return a \* b \* c;

}

return a \* b;

}

## Tham số mặc định

Khi bạn muốn khi có tham số thì lấy giá trị đó, còn khi không có thì tham số đó mặc định là 1 giá trị nào đó:

VD:

function sum(a: number, b: number = 2): number {

return a + b;

}

console.log(sum(2)); // 4

console.log(sum(2,3)); // 5

## Tham số rest

Typescript cho phép một hàm chấp nhận không hoặc nhiều đối số của kiểu dữ liệu cụ thể. Yêu cầu:

* Một hàm chỉ nhận một tham số rest.
* Tham số res xuất hiện cuối cùng trong danh sách tham số.
* Kiểu dữ liệu của rest là mảng kiểu dữ liệu.

VD:

function getTotal(...numbers: number[]): number {

let total = 0;

numbers.forEach((num) => total += num);

return total;

}

getTotal(2, 3) // 5

getTotal(2, 3, 5) // 10

rest có thể là mảng chứa nhiều kiểu dữ liệu

## Hàm quá tải

Nó tương tự như tính đa hình trong lập trình hướng đối tượng (nhưng khác chỗ typescript/javascript chỉ cho tạo 1 hàm có 1 tên duy nhất).

VD:

// Các chữ ký hàm (Overload Signatures)

function double(value: number): number;

function double(value: string): string;

// Hàm thực tế (Implementation)

function double(value: any): any {

if (typeof value === "number") {

return value \* 2;

} else if (typeof value === "string") {

return value + value;

}

}

console.log(double(10)); // Output: 20

console.log(double("Hi")); // Output: "HiHi"

console.log(double(true)); // Lỗi: Không có overload nào cho kiểu Boolean

# LỚP

## Lớp

Dùng để tạo đối tượng theo mô hình hướng đối tượng (OOP). Sử dụng từ khoá class để định nghĩa một lớp.

VD:

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}`;

}

}

## Trình sửa đổi truy cập

Typescript cung cấp 3 kiểu truy cập sửa đổi.

* public: thuộc tính hoặc phương thức có thể truy cập vào đối tượng để lấy giá trị (mặc định).
* private: thuộc tính hoặc phương thức chỉ có thể được sử dụng trong class, ngoài phạm vi không thể truy cập được.
* protect: thuộc tính hoặc phương thức chỉ có thể sử dụng trong class hoặc lớp kế thừa.

## Công cụ sửa đổi readonly

Từ khoá readonly dùng để đánh dấu thuộc tính của class là bất biến, tức là không thể chỉnh sửa.

VD:

class Person {

readonly birthDate: Date;

constructor(birthDate: Date) {

this.birthDate = birthDate;

}

}

So sánh readonly với const.

* readonly: chỉ sử dụng với class, có thể khởi tạo trong hàm khởi tạo của class
* const: sử dụng với biến, khởi tạo khi khai báo

## Getters & Setters

Getter trả về giá trị của giá trị thuộc tính

Setter cập nhật giá trị của thuộc tính

VD:

class Person {

private \_age: number;

private \_firstName: string;

private \_lastName: string;

constructor(age: number, firstName: string, lastName: string) {

this.\_age = age;

this.\_firstName = firstName;

this.\_lastName = lastName;

}

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}`;

}

}

## Kế thừa

Kế thừa là một tính năng quan trọng trong lập trình hướng đối tượng (OOP), cho phép một class có thể tái sử dụng các thuộc tính và phương thức của một class khác. Điều này giúp giảm sự lặp lại mã và tăng tính linh hoạt trong thiết kế phần mềm.

Lớp con (Child Class) kế thừa từ lớp cha (Parent Class) bằng cách sử dụng từ khóa extends. Lớp con có thể sử dụng lại tất cả các thuộc tính và phương thức của lớp cha.

VD:

class Person {

constructor(

private firstName: string,

private lastName: string,

) {}

getFullName(): string {

return `${this.firstName} ${this.lastName}`;

}

describe(): string {

return `This is ${this.firstName} ${this.lastName}.`;

}

}

class Employee extends Person {

constructor(

firstName: string,

lastName: string,

private jobTitle: string,

) {

super(firstName, lastName); // Gọi constructor của lớp cha

}

}

let employee = new Employee('John', 'Doe', 'Web Developer');

console.log(employee.getFullName()); // Output: "John Doe"

console.log(employee.describe()); // Output: "This is John Doe."

Nếu trong lớp con có khai báo thuộc tính của lớp cha thì khi tạo lớp con sẽ ghi đè thuộc tính đó của lớp cha.

## Phương thức tĩnh và thuộc tính

**Thuộc tính tĩnh (static properties)**

* Được chia sẻ giữa tất cả các instance của class.
* Khai báo bằng từ khóa static.
* Truy cập bằng cú pháp: ClassName.propertyName.

**Phương thức tĩnh (static method)**

* Cũng được chia sẻ giữa tất cả các instance.
* Không thể truy cập this, chỉ có thể truy cập các thành viên tĩnh khác của class.
* Truy cập bằng cú pháp: ClassName.methodName().

## Lớp trừu tượng

* Abstract class là một lớp trừu tượng, không thể khởi tạo trực tiếp.
* Dùng để định nghĩa hành vi chung cho các lớp con kế thừa.
* Khai báo bằng từ khóa abstract.

VD:

abstract class Employee {

constructor(private firstName: string, private lastName: string) {}

abstract getSalary(): number; // Phương thức trừu tượng (không có phần thân)

get fullName(): string {

return `${this.firstName} ${this.lastName}`;

}

compensationStatement(): string {

return `${this.fullName} makes ${this.getSalary()} a month.`;

}

}

Kế thừa

* Một class kế thừa phải triển khai tất cả các phương thức trừu tượng.
* Dùng từ khóa extends để kế thừa.
* Hữu ích khi muốn chia sẻ code chung giữa các lớp liên quan.

# INTERFACE

## Interface

Là sự định nghĩa một giao ước với code, cung cấp các tên rõ rang cho loại kiểm tra. Tuân theo quy tắc PascalCase (viết hoa các chữ cái đầu của từ).

VD:

interface Person {

firstName: string;

lastName: string;

}

Ngoài ra với những thuộc tính tuỳ chọn, bạn có thể đánh dấu ? khi khao báo:

interface Person {

firstName: string;

middleName?: string;

lastName: string;

}

Có thể khai báo từ khoá readonly cho thuộc tính nếu muốn nó chỉ được gán khi khởi tạo.

**Kiểu hàm:** Ngoài việc mô tả một đối tượng có thuộc tính, interface còn cho phép đại diện kiểu cho một hàm.

Có thể định nghĩa kiểu hàm trong một interface bằng cách sử dụng chữ ký hàm

interface StringFormat {

(str: string, isUpper: boolean): string;

}

Sau khi có interface, bạn có thể khai báo một biến có kiểu StringFormat và gán một hàm có cùng kiểu với nó

let format: StringFormat;

format = function (str: string, isUpper: boolean) {

return isUpper ? str.toUpperCase() : str.toLowerCase();

};

console.log(format('hi', true)); // Output: "HI"

## Interface mở rộng

Khi bạn muốn mở rộng một interface khác có các thuộc tính của interface khác nhưng bổ xung thêm thuộc tính:

VD:

interface Animal {

id: string;

name: string;

}

interface Cat extends Animal {

sound: string;

}

Lúc này Cat sẽ có: id, name, sound được kế thừa từ Animal mà không cần phải khai báo lại id và name

# LOẠI NÂNG CAO

## Kiểu kết hợp

* Intersection Type giúp kết hợp nhiều kiểu dữ liệu thành một kiểu mới.
* Kiểu mới sẽ chứa tất cả thuộc tính của các kiểu đã kết hợp.
* Dùng toán tử & để kết hợp kiểu:

VD:

interface BusinessPartner {

email: string;

phone: string;

}

interface Identity {

id: number;

name: string;

}

type Employee = Identity & BusinessPartner;

let e: Employee = {

id: 100,

name: 'John Doe',

email: 'john.doe@example.com',

phone: '(408)-897-5684'

};

## Kiểu bảo vệ

Giúp thu hẹp kiểu dữ liệu trong 1 khối điều kiện

* typeof: kiểm tra kiểu nguyên thuỷ

VD: const a: string = "Hello"

console.log(typeof(a)); // string

* instanceof: kiểm tra kiểu đối tượng dùng để kiểm tra đối tượng thuộc class nào

VD: partner instanceof Customer; // true nếu partner có class Customer

* in: Kiểm tra thuộc tính có tồn tại không

VD: 'isCreditAllowed' in partner; // true nếu partner có thuộc tính/class là isCreditAllowed

## Kiểu xác nhận

Hay còn gọi là ép kiểu, nếu TS không thể suy luận được mà bạn biết chính xác nó là gì, bạn có thể ép kiểu để TS hiểu được.

Dùng as hoặc <>

VD:

let el = document.querySelector('input[type="text"]');

let input = el as HTMLInputElement;

console.log(input.value); // Không lỗi

Hoặc (nhưng không dùng được cho JSX

let input = <HTMLInputElement>document.querySelector('input[type="text"]');

console.log(input.value);

Lưu ý: khi ép kiểu giữa 2 loại không tương thích, hãy dùng kiểu thứ 3 như unknown

VD:

let price = '9.99';

let netPrice = price as number; // Lỗi: Không thể ép string thành number

Hãy dùng

let price: unknown = '9.99';

let netPrice = price as number; // Không lỗi

Nếu như ép kiểu mà bạn ép kiểu sai, không đúng kiểu dữ liệu của nó, chương trình sẽ lỗi runtime

# GENERIC

## Giới thiệu về Typescript Generics

* Generics giúp bạn viết code linh hoạt, tái sử dụng được nhưng vẫn đảm bảo an toàn kiểu dữ liệu (type-safe).
* Cho phép sử dụng kiểu dữ liệu như một tham số khi định nghĩa hàm, class hoặc interface.

VD1: Xem xét ví dụ sau

function getRandomNumberElement(items: number[]): number {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

VD2: Xem xét ví dụ sau

function getRandomStringElement(items: string[]): string {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

Nhận thấy cả 2 hàm đều khá giống nhau chỉ khác kiểu dữ liệu.

Giải pháp là tạo một hàm chung cho mọi kiểu dữ liệu

function getRandomElement<T>(items: T[]): T {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

Với T là tham số kiểu dữ liệu

Có thể sử dụng kiểu tường minh:

let numbers = [1, 5, 7, 4, 2, 9];

let randomNum = getRandomElement<number>(numbers);

console.log(randomNum);

Hoặc để TS tự suy luận:

let colors = ['red', 'green', 'blue'];

let randomColor = getRandomElement(colors); // TS tự hiểu T = string

console.log(randomColor);

Lợi ích: An toàn, tái sử dụng code, giảm việc phải ép kiểu thủ công

## Hằng Generics

* Constraints (ràng buộc) giúp bạn giới hạn kiểu dữ liệu mà Generics có thể nhận.
* Dùng extends để đặt giới hạn cho kiểu dữ liệu của tham số Generic.
* Tránh lỗi khi truyền vào giá trị không phù hợp.

VD1: Tạo 1 hàm nhận 2 object và gộp 2 object này lại thành 1 object mới

function merge<U, V>(obj1: U, obj2: V) {

return { ...obj1, ...obj2 };

}

Nhược điểm: nếu tham số không phải object sẽ không gộp giá trị đó, và không báo lỗi

Giải pháp, thêm ràng buộc extends object:

function merge<U extends object, V extends object>(obj1: U, obj2: V) {

return { ...obj1, ...obj2 };

}

Lúc này nếu truyền tham số không phải object sẽ báo lỗi

VD2: Đối với ràng buộc khoá của object

function prop<T, K>(obj: T, key: K) {

return obj[key]; // Lỗi: TypeScript không biết K có phải key của obj hay không

}

Vì TS không biết key có phải khoá của object hay không nên sẽ báo lỗi

function prop<T, K extends keyof T>(obj: T, key: K) {

return obj[key];

}

Khi khai báo K là extends keyof T thì TS xác định K là kiểu dữ liệu thuộc object T nên sẽ không báo lỗi khi runtime, tuy nhiên nếu truyền key không thuộc khoá của obj vẫn sẽ lỗi.

## Lớp Generics

VD1:

class ClassName<K, T> {

// Code...

}

VD2:

class ClassName<T extends TypeA> {

// Code...

}

Thường được ứng dụng vào Stack (LIFO – Last In First Out)

## Giao diện Generics

**Interface Generic** giúp định nghĩa interface có thể làm việc với nhiều kiểu dữ liệu khác nhau mà vẫn đảm bảo **type safety**.

VD1: Dùng nhiều Gerneric

interface InterfaceName<U, V> {

// Code...

}

VD2: Mô tả cặp khoá key – value

interface Pair<K, V> {

key: K;

value: V;

}

// Sử dụng với kiểu dữ liệu cụ thể

let month: Pair<string, number> = {

key: 'Jan',

value: 1

};

console.log(month);

# MODULES

## Typescript Modules

* Module trong TypeScript tương tự ES6 Modules, giúp tổ chức mã nguồn dễ đọc, dễ quản lý.
* Module có phạm vi riêng, không ảnh hưởng đến global scope.
* Sử dụng export để xuất dữ liệu ra ngoài module.
* Sử dụng import để nhập dữ liệu từ module khác.

**Export một module**

interface Validator {

isValid(s: string): boolean;

}

export { Validator };

Hoặc viết gộp:

export interface Validator {

isValid(s: string): boolean;

}

Nếu không dùng export thì module mặc định là private và chỉ sử dụng ở phạm vị khối hoặc riêng chỉ file đó.

Có thể đổi tên export:

export { Validator as StringValidator };

**Import một module**

Import thông thường:

import { Validator } from './Validator';

Đổi tên khi import:

import { Validator as StringValidator } from './Validator';

Import tất cả nội dung:

import \* as Validators from './Validator';

Import type, Cách này giúp TypeScript loại bỏ import không cần thiết trong mã JavaScript sau khi biên dịch.:

import type { alphanumeric } from './Types';

**Export Default**:

* Không cần {} khi import default export.
* 1 file chỉ có duy nhất 1 export default.

VD:

import { Validator } from './Validator';

export default class ZipCodeValidator implements Validator {

isValid(s: string): boolean {

const numberRegexp = /^[0-9]+$/;

return s.length === 5 && numberRegexp.test(s);

}

}

// ====

import ZipCodeValidator from './ZipCodeValidator';

# BỔ XUNG

## Loại bỏ thuộc tính khỏi kiểu (Omit)

Trái ngược với extend, Omit sẽ loại bỏ đi bớt các thuộc tính của interface

VD:

interface User {

id: number;

name: string;

age: number;

email: string;

}

// Loại bỏ thuộc tính "email" và "age"

type UserWithoutEmail = Omit<User, "email" | "age">;

const user: UserWithoutEmail = {

id: 1,

name: "John Doe"

};

Cách 2: Có thể dùng interface extends, vẫn giữ được đặc điểm là interface và có thể mở rộng thêm các thuộc tính khác.

interface UserWithoutEmail extends Omit<User, "email" | "age"> {}

So sánh:

|  |  |  |
| --- | --- | --- |
| Tính năng | interface | type |
| Kế thừa | Có | Không |
| Kết hợp | extends | & |
| Hợp nhất | Không | Có |
| Dùng với Utility Types (Omit, Pick, Partial...) | Khó | Linh hoạt |

## Chọn thuộc tính cụ thể (Pick)

Pick chỉ lấy một số thuộc tính cụ thể

type UserSummary = Pick<User, "id" | "name">;

## Chuyển tất cả thuộc tính thành tùy chọn (Partial)

Biến tất cả thuộc tính trong kiểu thành **optional (?)**.

type PartialUser = Partial<User>;

const user: PartialUser = {

name: "Charlie"

}; // Không cần tất cả thuộc tính

## Chuyển tất cả thuộc tính thành bắt buộc (Required)

Ngược lại với Partial, **bắt buộc** phải khai báo đầy đủ tất cả thuộc tính.

type RequiredUser = Required<PartialUser>;

const user: RequiredUser = {

id: 1,

name: "David",

age: 25,

email: "david@example.com"

};

## Chuyển tất cả thuộc tính thành chỉ đọc (Readonly)

Biến tất cả thuộc tính thành **chỉ đọc (readonly)**.

type ReadonlyUser = Readonly<User>;

const user: ReadonlyUser = {

id: 1,

name: "Eve",

age: 30,

email: "eve@example.com"

};

// user.name = "New Name"; Lỗi: name là readonly