

# TYPESCRIPT THAM KHẢO NHANH

Kiểu dữ liệu, interface, generics, utility type

## Kiểu Cơ Bản

### Kiểu Nguyên Thủy

```
let name: string = "Alice";
let age: number = 25;
let active: boolean = true;
let data: null = null;
let x: undefined = undefined;
```

### Kiểu Đặc Biệt

**any** Bỏ qua kiểm tra kiểu  
**unknown** 'any' an toàn kiểu (phải thu hẹp trước khi dùng)  
**void** Không có giá trị trả về  
**never** Hàm không bao giờ trả về (ném lỗi / vô hạn)  
**object** Bất kỳ kiểu không nguyên thủy nào

## Mảng & Tuple

### Mảng

```
let nums: number[] = [1, 2, 3];
let names: Array<string> = ["a", "b"];
let matrix: number[][] = [[1, 2], [3, 4]];
```

### Tuple

```
let pair: [string, number] = ["age", 25];
let rgb: [number, number, number] = [255, 0, 0];

// Named tuples (Labels for readability)
type Point = {x: number, y: number};
```

## Interface

### Định Nghĩa & Sử Dụng

```
interface User {
  name: string;
  age: number;
  email?: string; // optional
  readonly id: number; // immutable
}

const user: User = { name: "Alice", age: 25, id: 1 };
```

### Mở Rộng Interface

```
interface Employee extends User {
  role: string;
  department: string;
}
```

### Index Signature

```
interface StringMap {
  [key: string]: string;
}

const env: StringMap = { NODE_ENV: "prod" };
```

### Type Alias

```
type ID = string | number;
type Point = { x: number; y: number };
type Callback = (data: string) => void;
```

### Interface vs Type

**interface** Có thể mở rộng với `extends`, hỗ trợ declaration merging  
**type** Union, intersection, mapped type, tuple

## Union & Intersection

### Union Type

```
type Status = "loading" | "success" | "error";
type ID = string | number;

function print(val: string | number) {
  if (typeof val === "string") {
    console.log(val.toUpperCase());
  }
}
```

### Intersection Type

```
type Named = { name: string };
type Aged = { age: number };
type Person = Named & Aged;
// Person has both name and age
```

### Discriminated Union

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rect"; w: number; h: number };

function area(s: Shape): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.radius ** 2;
    case "rect": return s.w * s.h;
  }
}
```

## Hàm

### Tham Số & Kiểu Trả Về

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

// Arrow function
const greet = (name: string): string =>
  `Hello, ${name}!`;

// Optional & default params
function log(msg: string, level?: string): void {}
function log(msg: string, level = "info"): void {}
```

### Overload Hàm

```
function parse(input: string): number;
function parse(input: number): string;
function parse(input: string | number) {
  return typeof input === "string"
    ? parseInt(input)
    : input.toString();
}
```

### Tham Số Rest

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

## Generics

### Hàm Generic

```
function identity<T>(value: T): T {
  return value;
}

identity<string>("hello"); // explicit
identity(42); // inferred: number
```

### Interface Generic & Ràng Buộc

```
interface Box<T> {
  value: T;
}

const box: Box<number> = { value: 42 };

// Constraints
function getLen<T extends { length: number }>(
  item: T
): number {
  return item.length;
}
```

## Enum

```
enum Direction { Up, Down, Left, Right }
let d: Direction = Direction.Up; // 0

enum Status {
  Active = "ACTIVE",
  Inactive = "INACTIVE",
}

let s: Status = Status.Active; // "ACTIVE"

// const enum (inlined at compile time)
const enum Color { Red, Green, Blue }
```

## Type Guard

### Guard Dạng Sẵn

```
// typeof
if (typeof x === "string") { /* x: string */ }

// instanceof
if (err instanceof Error) { /* err: Error */ }

// in
if ("name" in obj) { /* obj has name */ }
```

### Type Guard Tùy Chính

```
function isString(val: unknown): val is string {
  return typeof val === "string";
}

if (isString(input)) {
  input.toUpperCase(); // narrowed to string
}
```

## Assertion Function

```
function assertDefined<T>(
  val: T | null
): asserts val is T {
  if (val === null) throw new Error("null");
}
```

## Utility Type

**Partial<T>** Tất cả thuộc tính tùy chọn  
**Required<T>** Tất cả thuộc tính bắt buộc  
**Readonly<T>** Tất cả thuộc tính chỉ đọc  
**Pick<T, K>** Chọn thuộc tính K từ T  
**Omit<T, K>** Loại bỏ thuộc tính K từ T  
**Record<K, V>** Object có key K và value V  
**Exclude<T, U>** Kiểu trong T không có trong U  
**Extract<T, U>** Kiểu trong T cũng có trong U  
**NonNullable<T>** Loại bỏ null và undefined từ T  
**ReturnType<T>** Kiểu trả về của hàm T  
**Parameters<T>** Kiểu tham số của hàm T  
**Awaited<T>** Mở gói kiểu Promise

### Ví Dụ Utility Type

```
interface User { name: string; age: number; email: string }

type UserPreview = Pick<User, "name" | "email">;
type UserUpdate = Partial<User>;
type UserMap = Record<string, User>;
type CreateUser = Omit<User, "id">;
```