

## Composing Types (Tổ hợp kiểu dữ liệu)

Trong TypeScript, các kiểu dữ liệu phức tạp có thể được xây dựng bằng cách **kết hợp các kiểu đơn giản**. Hai cơ chế phổ biến và nền tảng nhất để thực hiện điều này là **union types** và **generics**. Phần này tập trung vào **union types**.

## Union Types

**Union type** cho phép mô tả một kiểu dữ liệu có thể nhận **một trong nhiều kiểu khác nhau**. Về mặt hình thức, union được biểu diễn bằng toán tử `|`.

Ví dụ, có thể mô tả một kiểu Boolean bằng cách liệt kê các giá trị hợp lệ của nó:

```
type MyBool = true | false;
```

Mặc dù `MyBool` được định nghĩa từ hai literal type `true` và `false`, nhưng trong hệ thống kiểu của TypeScript, nó vẫn được phân loại là `boolean`. Đây là hệ quả trực tiếp của **Structural Type System**, trong đó TypeScript quan tâm đến cấu trúc và khả năng tương thích kiểu hơn là nguồn gốc danh nghĩa của kiểu.

## Union với literal types

Một trong những ứng dụng phổ biến nhất của union types là **giới hạn tập giá trị hợp lệ** mà một biến có thể nhận, đặc biệt với **string literals** và **number literals**.

Ví dụ:

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Các định nghĩa trên tạo ra những kiểu có miền giá trị hữu hạn, cho phép TypeScript kiểm tra lỗi một cách chính xác tại thời điểm biên dịch nếu một giá trị nằm ngoài tập được cho phép. Cách tiếp cận này thường được sử dụng để mô hình hóa **state**, **mode**, hoặc **enum-like behavior** mà không cần dùng `enum`.

## Union với nhiều kiểu dữ liệu khác nhau

Union types không chỉ giới hạn ở literal types mà còn có thể kết hợp các kiểu dữ liệu hoàn toàn khác nhau. Điều này đặc biệt hữu ích khi một function cần xử lý nhiều dạng dữ liệu đầu vào.

Ví dụ, một function có thể nhận vào một `string` hoặc một `string[]`:

```
function getLength(obj: string | string[]) {  
  return obj.length;  
}
```

Trong trường hợp này, cả `string` và `string[]` đều có property `length`, nên TypeScript cho phép truy cập trực tiếp mà không cần kiểm tra kiểu bổ sung.

## Thu hẹp kiểu bằng typeof và type guard

Khi làm việc với union types, để xác định **kiểu cụ thể tại runtime**, ta sử dụng các **type guard**. Phổ biến nhất là toán tử `typeof` và các hàm kiểm tra cấu trúc như `Array.isArray`.

Bảng ánh xạ giữa kiểu và biểu thức kiểm tra tương ứng:

TYPE	PREDICATE
string	<code>typeof s === "string"</code>
number	<code>typeof n === "number"</code>
boolean	<code>typeof b === "boolean"</code>
undefined	<code>typeof undefined === "undefined"</code>
function	<code>typeof f === "function"</code>
array	<code>Array.isArray(a)</code>

Các biểu thức này cho phép TypeScript **thu hẹp union type (type narrowing)** trong từng nhánh điều kiện.

## Ví dụ: Thu hẹp kiểu trong function

Xét function sau:

```
function wrapInArray(obj: string | string[]) {  
  if (typeof obj === "string") {  
    return [obj];  
  }  
  return obj;  
}
```

Trong nhánh `if`, điều kiện `typeof obj === "string"` khiến TypeScript suy luận rằng `obj` có kiểu `string`. Do đó, việc đặt `obj` vào một array là hợp lệ.

Ở nhánh `else`, TypeScript tự động suy luận rằng `obj` phải là `string[]`. Nhờ cơ chế thu hẹp kiểu này, function có thể xử lý an toàn cả hai trường hợp mà không cần ép kiểu thủ công.

## Tổng kết

Union types cung cấp một cơ chế linh hoạt để:

- Mô hình hóa dữ liệu có nhiều dạng hợp lệ
- Giới hạn miền giá trị của biến một cách chính xác
- Kết hợp nhiều kiểu khác nhau trong cùng một API
- Cho phép TypeScript thực hiện kiểm tra kiểu chặt chẽ thông qua type narrowing

Đây là một trong những thành phần cốt lõi giúp TypeScript mở rộng JavaScript mà vẫn giữ được tính tự nhiên và khả năng biểu đạt cao của ngôn ngữ gốc.