# Chapter 8: Thao Tác Kiểu Nâng Cao - Mapped Types, Conditional Types và Utility Types

**Mô tả tổng quát**

Chapter này sẽ khám phá các tính năng kiểu nâng cao của TypeScript, giúp bạn thực hiện các thao tác phức tạp với kiểu dữ liệu. Chúng ta sẽ tìm hiểu:

- **Mapped Types**: Tạo kiểu mới bằng cách biến đổi các thuộc tính của một kiểu hiện có.
- **Conditional Types**: Chọn kiểu dựa trên điều kiện logic.
- **Utility Types**: Các kiểu tiện ích tích hợp sẵn trong TypeScript, giúp tiết kiệm thời gian và tăng tính an toàn.

Nắm vững các khái niệm này sẽ giúp bạn viết mã TypeScript linh hoạt, tái sử dụng và an toàn hơn, đặc biệt trong các dự án lớn như ứng dụng React hoặc API Node.js.

## Tóm tắt lý thuyết chính

### 1. Toán tử `keyof`

**Giải thích**:
Toán tử `keyof` lấy tất cả các key (thuộc tính) của một kiểu đối tượng và trả về một **union type** (hợp các chuỗi literal hoặc symbol literal). Đây là nền tảng để làm việc với các kiểu động trong TypeScript, ví dụ như truy cập thuộc tính hoặc lặp qua các key.

**Ví dụ**:

```
interface UserProfile {
    id: number;
    username: string;
    email: string;
    lastLogin: Date;
}
```

```
type UserProfileKeys = keyof UserProfile;
// Kết quả: "id" | "username" | "email" | "lastLogin"

let userKey: UserProfileKeys = "username"; // OK
// let invalidKey: UserProfileKeys = "password"; // Lỗi: "password" không tồ
```

**Ứng dụng thực tế:**

- Dùng `keyof` để tạo hàm generic xử lý thuộc tính động:

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

const user: UserProfile = { id: 1, username: "john", email: "john@example.co
const username = getProperty(user, "username"); // Type: string
```

## 2. Toán tử `typeof` (trong ngữ cảnh type)

**Giải thích:**
Toán tử `typeof` trong TypeScript (khác với JavaScript) lấy kiểu của một giá trị (biến, hằng số, hoặc thuộc tính). Nó rất hữu ích khi bạn muốn tái sử dụng kiểu của một biến mà không cần định nghĩa lại.

**Ví dụ:**

```
let primaryColor = "blue"; // TypeScript suy luận là string
type ColorType = typeof primaryColor; // ColorType là string

const appConfig = {
    version: "1.0.0",
    debugMode: false
};
type AppConfigType = typeof appConfig;
```

```
// AppConfigType: { version: string; debugMode: boolean; }

let currentConfig: AppConfigType = { version: "1.1.0", debugMode: true };
```

**Ứng dụng thực tế:**

- Dùng `typeof` để định nghĩa kiểu cho cấu hình ứng dụng:

```
const DEFAULT_CONFIG = {
    apiUrl: "https://api.example.com",
    timeout: 5000
};
type Config = typeof DEFAULT_CONFIG;

function initializeApp(config: Config) {
    // Xử lý cấu hình
}
```

## 3. Indexed Access Types ( `T[K]` )

**Giải thích:**
Indexed Access Types cho phép lấy kiểu của một thuộc tính cụ thể từ một kiểu đối tượng, tương tự như cách bạn truy cập thuộc tính trong JavaScript ( `obj[key]` ). Nó rất mạnh khi kết hợp với `keyof` để truy cập kiểu động.

**Ví dụ:**

```
interface Product {
    id: number;
    name: string;
    price: number;
    tags: string[];
}

type ProductNameType = Product["name"]; // string
```

```ts
type ProductTagsType = Product["tags"]; // string[]
type ProductIdOrPrice = Product["id" | "price"]; // number

type ProductKeys = keyof Product; // "id" | "name" | "price" | "tags"
type ProductPropertyType<K extends ProductKeys> = Product[K];

let pName: ProductPropertyType<"name"> = "Laptop"; // OK
```

**Ứng dụng thực tế:**

- Dùng để lấy kiểu của một thuộc tính trong hàm generic:

```ts
function updateProductField<K extends keyof Product>(product: Product, key:
    product[key] = value;
}

const product: Product = { id: 1, name: "Phone", price: 500, tags: ["tech"]
updateProductField(product, "price", 600); // OK
// updateProductField(product, "category", "electronics"); // Lỗi
```

## 4. Mapped Types

**Giải thích:**
Mapped Types tạo kiểu mới bằng cách lặp qua các key của một kiểu đối tượng và áp dụng biến đổi cho từng thuộc tính. Cú pháp `[K in keyof T]: NewType` giống như một vòng lặp trong TypeScript, cho phép bạn thay đổi kiểu hoặc thuộc tính.

**Ví dụ:**

```ts
interface Options {
    width: number;
    height: number;
    color: string;
}
```

```
type FeatureFlags<T> = {
    [P in keyof T]: boolean;
};
type OptionFlags = FeatureFlags<Options>;
// OptionFlags: { width: boolean; height: boolean; color: boolean; }

let currentFlags: OptionFlags = { width: true, height: false, color: true };

type StringifiedOptionalOptions<T> = {
    [P in keyof T]?: string;
};
type ConfigStrings = StringifiedOptionalOptions<Options>;
// ConfigStrings: { width?: string; height?: string; color?: string; }

let userConfig: ConfigStrings = { width: "100px", color: "blue" };
```

### Ứng dụng thực tế:

- Tạo kiểu cho form input trong React:

```
type FormValues<T> = {
    [K in keyof T]: string;
};

interface UserForm {
    name: string;
    email: string;
}

const form: FormValues<UserForm> = { name: "John", email: "john@example.com"
```

### Utility Types dựa trên Mapped Types:

- `Readonly<T>` : Làm tất cả thuộc tính thành `readonly` .
- `Partial<T>` : Làm tất cả thuộc tính thành tùy chọn.
- `Required<T>` : Làm tất cả thuộc tính thành bắt buộc.

- `Pick<T, K>`: Chọn một tập hợp thuộc tính từ `T`.
- `Omit<T, K>`: Loại bỏ một tập hợp thuộc tính từ `T`.

**Ví dụ Utility Types:**

```
type ReadonlyOptions = Readonly<Options>;
type PartialOptions = Partial<Options>;
type DimensionOptions = Pick<Options, "width" | "height">;
type StyleOptions = Omit<Options, "width" | "height">;
```

## 5. Conditional Types (`T extends U ? Y : Z`)

**Giải thích:**
Conditional Types chọn kiểu dựa trên một điều kiện logic, giống như câu lệnh `if-else`.
Nó rất mạnh khi kết hợp với từ khóa `infer` để suy luận kiểu từ cấu trúc của kiểu đầu vào.

**Ví dụ:**

```
type IsString<T> = T extends string ? true :  false;

type Result1 = IsString<string>; // true
type Result2 = IsString<number>; // false

type ElementType<T> = T extends (infer U)[] ?  U : T;
type ItemType1 = ElementType<string[]>; // string
type ItemType2 = ElementType<number[]>; // number
type ItemType3 = ElementType<boolean>; // boolean
```

**Ứng dụng thực tế:**

- Kiểm tra kiểu trong hàm generic:

```ts
function unwrap<T>(value: T): T extends Array<infer U> ? U : T {
    if (Array.isArray(value)) {
        return value[0]; // Giả định lấy phần tử đầu tiên
    }
    return value;
}

const str = unwrap(["hello"]); // Type: string
const num = unwrap(42); // Type: number
```

**Utility Types dựa trên Conditional Types:**

- `Exclude<T, U>` : Loại bỏ các kiểu con có thể gán cho `U` .
- `Extract<T, U>` : Trích xuất các kiểu con có thể gán cho `U` .
- `NonNullable<T>` : Loại bỏ `null` và `undefined` .
- `ReturnType<T>` : Lấy kiểu trả về của hàm.
- `Parameters<T>` : Lấy kiểu tham số dưới dạng tuple.
- `InstanceType<T>` : Lấy kiểu của instance từ hàm constructor.

**Ví dụ Utility Types:**

```ts
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T4 = NonNullable<string | null>;
 // string
type T5 = ReturnType<() => string>;
 // string
```

# Code ví dụ chính (Tổng hợp và mở rộng)

```ts
// keyof, typeof, Indexed Access
interface AppSettings {
```

```
    theme: "light" | "dark";
    fontSize: number;
    notifications: {
        enabled: boolean;
        sound: string;
    };
}

type SettingsKeys = keyof AppSettings;
// "theme" | "fontSize" | "notifications"
type NotificationSettingsType = AppSettings["notifications"];
 // { enabled: boolean; sound: string; }

const defaultSettings: AppSettings = {
    theme: "light",
    fontSize: 14,
    notifications: { enabled: true; sound: "default.mp3" }
};
type DefaultSettingsType = typeof defaultSettingsType; // Kiểu của defaultSe

// Mapped Types
type StringifiedReadonly<T> = {
    readonly [P in keyof T]: string;
};
type StringifiedAppSettings = StringifiedReadonly<AppSettings>;

// Conditional Types
type GetDataType<T> = T extends { data: infer D } ? D : never;
type ResponseA = { status: string; data: number[]};
type DataA = { GetDataType<ResponseA>; // number[]
type ResponseB = { status: string; message: string };
type DataB = GetDataType<ResponseB>; // never

// Utility Types
interface User {
    id: number;
    name: string;
    email?: string;
    passwordHash: string;
    createdAt: Date;
    updatedAt?: Date;
}
```

```ts
type UserUpdatePayload = Partial<Pick<User, "name" | "email">>;
type PublicUser = Omit<User, "passwordHash" | "createdAt" | "updatedAt">;

function createUser(name: string, email?: string): User {
    return {
        id: Math.random(),
        name,
        email,
        passwordHash: "hashed",
        createdAt: new Date()
    };
}

type CreateUserParams = Parameters<typeof createUser>; // [string, (string |
type CreatedUserType = ReturnType<typeof createUser>; // User
```

# Danh sách bài tập

## 1. Trắc nghiệm: Utility Type `Pick`

**Mô tả:**
Chọn utility type phù hợp để tạo kiểu mới bằng cách chọn các thuộc tính cụ thể từ một kiểu đối tượng.

**Câu hỏi:**
Utility type nào dùng để tạo một kiểu mới bằng cách chọn một tập hợp các thuộc tính từ một kiểu hiện có?

- A. Partial`
- B. Readonly`
- C. **Pick<T, K>`
- D. Omit<T, K>`

**Đáp án**: C

## 2. Code: Tạo Mapped Type `NullableProps<T>`

**Mô tả:**
Tạo kiểu generic `NullableProps<T>` để tất cả thuộc tính của `T` có thể là `null` hoặc kiểu gốc.

```
type NullableProps<T> = {
    [P in keyof T]: T[P] | null;
};
interface UserDetails {
    username: string;
    email:string;
    lastLogin: Date;
}
type NullableUserDetails = NullableProps<UserDetails>;

let userWithNulls: NullableUserDetails = {
    username: "testuser",
    email: null,
    lastLogin: new Date()
};
```

## 3. Code: Conditional Type `UnwrapPromise<T>`

**Mô tả:**
Tạo `UnwrapPromise<T>` để lấy kiểu giá trị bên trong `Promise`, hoặc giữ nguyên kiểu nếu không phải `Promise`.

```
type UnwrapPromise<T> = T extends Promise<infer U> ? U : T;

type PromiseStringType = Promise<string>;
type UnwrappedString = UnwrapPromise<PromiseStringType>; // string
```

## 4. Code: Sử dụng `Omit<T, K>`

**Mô tả:**
Tạo kiểu `TodoCreationPayload` bằng cách bỏ các thuộc tính `id`, `createdAt`, `updatedAt` từ interface `Todo`.

```typescript
interface Todo {
    id: number;
    text: string;
    completed: boolean;
    createdAt: Date;
    updatedAt: Date;
}

type TodoCreationPayload = Omit<Todo, "id" | "createdAt" | "updatedAt">;

let newTodo: TodoCreationPayload = {
    text: "Học TypeScript nâng cao",
    completed: false
};
```

## 5. Code: Lấy kiểu tham số và trả về của hàm

**Mô tả:**
Sử dụng `Parameters<T>` và `ReturnType<T>` để lấy kiểu tham số và trả về của hàm `processUserData`.

```typescript
function processUserData(id: number,
    data: { name: string; age: number },
    isActive?: boolean
): { success: boolean; message?: string } {
    console.log(`Processing user ${id}, Active: ${isActive ?? false}`);
    if (data.age < 18) {
        return { success: false, message: "User is underage." };
    }
    return { success: true };
```

```
}

type ProcessUserParams = Parameters< typeof processUserData>;
type ProcessUserReturn = ReturnType<ProcessUserData>;

let exampleParams: ProcessUserParams = [101, { name: "John Doe", age: 30 },
let exampleReturnSuccess: ProcessUserReturn = { success: true };
```