Chapter 4: Định Hình Dữ Liệu: Interfaces và Type Aliases

Mô tả tổng quát Chapter này giới thiệu hai công cụ cốt lõi trong TypeScript để định nghĩa **hình dạng** (shape) của dữ liệu: `Interface` và `Type Alias`. Cả hai giúp bạn tạo ra các **hợp đồng** (contracts) để đảm bảo dữ liệu tuân theo cấu trúc cụ thể, áp dụng cho đối tượng, hàm, và cả class. Bạn sẽ học cách sử dụng chúng, so sánh sự khác biệt, và chọn công cụ phù hợp cho từng tình huống.

Mục tiêu học tập:

- Hiểu cách sử dụng `Interface` và `Type Alias` để định nghĩa kiểu dữ liệu.
- Nắm được sự khác biệt giữa `Interface` và `Type Alias`, bao gồm các trường hợp sử dụng cụ thể.
- Thực hành tạo và sử dụng `Interface`, `Type Alias` trong các ví dụ thực tế.

Tóm tắt lý thuyết chính

1. Interfaces

`Interface` là một cách mạnh mẽ để định nghĩa cấu trúc của đối tượng hoặc class. Nó giống như một bản thiết kế (blueprint), yêu cầu các thực thể tuân theo phải có đủ các thuộc tính/phương thức được khai báo.

Định nghĩa Interface cho đối tượng

```
interface User {
    id: number;
    username: string;
    email: string;
    isActive: boolean;
```

```
}

let user1: User = {
    id: 1,
    username: "typescriptFan",
    email: "fan@example.com",
    isActive: true
};
```

Giải thích:

- `interface User` định nghĩa một đối tượng phải có 4 thuộc tính: `id` (số), `username` (chuỗi), `email` (chuỗi), `isActive` (boolean).
- Nếu thiếu thuộc tính (như `email` hoặc `isActive`) hoặc sai kiểu dữ liệu, TypeScript sẽ báo lỗi ngay lập tức.
- **Ứng dụng thực tế**: Dùng để kiểm tra dữ liệu người dùng từ API hoặc form nhập liệu trong ứng dụng web.

Thuộc tính tùy chọn (`?`)

```
interface Profile {
    userId: number;
    bio?: string; // Tùy chọn
    websiteUrl?: string; // Tùy chọn
}

let profile1: Profile = { userId: 101 }; // OK
let profile2: Profile = { userId: 102, bio: "Loves coding!", websiteUrl: "ht
```

Giải thích:

- Dấu `?` chỉ định thuộc tính không bắt buộc. Điều này hữu ích khi dữ liệu có thể thiếu (ví dụ: người dùng chưa điền bio).
- **Ứng dụng thực tế**: Dùng cho các form tùy chỉnh, nơi một số trường có thể để trống.

Thuộc tính chỉ đọc (`readonly`)

```
interface Point {
    readonly x: number;
    readonly y: number;
}

let p1: Point = { x: 10, y: 20 };
// p1.x = 5; // Lỗi: Cannot assign to 'x' because it is a read-only property
```

Giải thích:

- `readonly` ngăn việc thay đổi giá trị thuộc tính sau khi khởi tạo.
- **Ứng dụng thực tế**: Dùng để bảo vệ dữ liệu bất biến, như tọa độ của một điểm hoặc ID của đối tượng.

Index Signatures (Chữ ký chỉ mục)

```
interface StringValueDictionary {
    [key: string]: string; // Key là chuỗi, giá trị là chuỗi
}

let myDict: StringValueDictionary = {
    prop1: "value1",
    prop2: "value2"
};
// myDict.prop3 = 123; // Lỗi: Type 'number' is not assignable to type 'stri

interface MixedDictionary {
    [index: string]: string | number; // Key là chuỗi, giá trị là chuỗi hoặc
    length: number; // Tương thích
    name: string;  // Tương thích
}

let config: MixedDictionary = { length: 10, name: "AppConfig", version: "1.0
```

Giải thích:

- Chữ ký chỉ mục cho phép định nghĩa kiểu cho các thuộc tính động (không biết trước tên).
- Lưu ý: Các thuộc tính cụ thể (như `length`, `name`) phải có kiểu tương thích với chữ ký chỉ mục.
- **Ứng dụng thực tế**: Dùng để mô tả các cấu trúc dữ liệu động, như cấu hình ứng dụng hoặc dữ liệu từ API không cố định.

Lưu ý tránh lỗi:

- Nếu chữ ký chỉ mục khai báo `[key: string]: string`, thì tất cả giá trị phải là chuỗi. Thêm `length: number` sẽ gây lỗi, trừ khi mở rộng chữ ký chỉ mục thành `[key: string]: string | number`.

Function Types trong Interfaces

```
interface SearchFunction {
    (source: string, subString: string): boolean;
}

let mySearch: SearchFunction = (src, sub) => src.search(sub) > -1;
console.log(mySearch("hello world", "world")); // true
```

Giải thích:

- Interface có thể định nghĩa chữ ký hàm, chỉ định kiểu tham số và giá trị trả về.
- **Ứng dụng thực tế**: Dùng để kiểm tra kiểu của các callback trong hàm xử lý sự kiện hoặc API.

Extending Interfaces (Kế thừa Interface)

```
interface Shape {
    color: string;
}
```

```
interface Square extends Shape {
    sideLength: number;
}

let mySquare: Square = { color: "blue", sideLength: 10 };
```

Giải thích:

- `extends` cho phép kế thừa từ interface khác, mở rộng cấu trúc dữ liệu.
- **Ứng dụng thực tế**: Dùng để xây dựng hệ thống phân cấp kiểu, như mô tả các loại sản phẩm (sản phẩm cơ bản, sản phẩm cao cấp).

Class Implementing Interface

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date): void;
}

class DigitalClock implements ClockInterface {
    currentTime: Date = new Date();
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { /* ... */ }
}
```

Giải thích:

- Class phải triển khai tất cả thuộc tính/phương thức của interface.
- **Ứng dụng thực tế**: Dùng để đảm bảo các class (như `DigitalClock`) tuân theo hợp đồng, ví dụ trong các ứng dụng quản lý thời gian.

2. Type Aliases

`Type Alias` tạo tên mới cho một kiểu dữ liệu, áp dụng cho cả kiểu nguyên thủy, union types, intersection types, tuples, và các kiểu phức tạp.

Type Alias cho kiểu nguyên thủy

```
type MyString = string;
let pageTitle: MyString = "Homepage";
```

Giải thích:

- Đơn giản nhưng hữu ích để tăng tính rõ ràng khi đặt tên kiểu.
- **Ứng dụng thực tế**: Dùng để đặt tên ngữ nghĩa cho các kiểu đơn giản, như `UserId` thay vì `string | number`.

Type Alias cho Union Types

```
type StringOrNumber = string | number;
let input: StringOrNumber = "test";
input = 123; // OK
```

Giải thích:

- Union type cho phép giá trị thuộc một trong các kiểu được liệt kê.
- **Ứng dụng thực tế**: Dùng để xử lý dữ liệu đa dạng, như input từ người dùng có thể là chuỗi hoặc số.

Type Alias cho kiểu đối tượng

```
type Point2D = {
    x: number;
    y: number;
```

```
    label?: string;
};

let p2: Point2D = { x: 5, y: 15, label: "Center" };
```

Giải thích:

- Tương tự `interface`, nhưng cú pháp ngắn gọn hơn.
- **Ứng dụng thực tế**: Dùng để mô tả dữ liệu từ API hoặc cấu trúc dữ liệu trong React state.

Type Alias cho kiểu hàm

```
type GreetHandler = (name: string) => void;
const greetUser: GreetHandler = (name) => console.log(`Hello, ${name}!`);
greetUser("TypeScript User");
```

Giải thích:

- Định nghĩa chữ ký hàm tương tự interface.
- **Ứng dụng thực tế**: Dùng trong các component React để định nghĩa kiểu cho props là hàm.

Type Alias với Generics

```
type Container<T> = { value: T };
let numberContainer: Container<number> = { value: 100 };
```

Giải thích:

- Generic cho phép định nghĩa kiểu linh hoạt, tái sử dụng.
- **Ứng dụng thực tế**: Dùng trong các thư viện hoặc hàm xử lý dữ liệu đa dạng, như Redux store.

3. Sự khác biệt giữa Interfaces và Type Aliases

Tiêu chí	Interface	Type Alias
Declaration Merging	Hỗ trợ (có thể khai báo lại để hợp nhất)	Không hỗ trợ (lỗi nếu khai báo lại)
Extending	Sử dụng `extends` (rõ ràng, trực quan)	Sử dụng `&` (intersection type, ít trực quan)
Class Implements	Hỗ trợ trực tiếp	Chỉ hỗ trợ với union/intersection của class
Kiểu phức tạp	Hạn chế với union, intersection, mapped types	Linh hoạt hơn, hỗ trợ mọi kiểu dữ liệu

Khi nào dùng cái nào?

- **Dùng** `Interface` :
  - Khi định nghĩa hình dạng đối tượng hoặc class.
  - Khi cần declaration merging (ví dụ: mở rộng thư viện bên thứ ba).
  - Khi làm việc với lập trình hướng đối tượng.
- **Dùng** `Type Alias` :
  - Khi cần định nghĩa union types, intersection types, hoặc kiểu phức tạp (mapped types, conditional types).
  - Khi đặt tên cho kiểu nguyên thủy hoặc tuples.
- **Lưu ý**: Trong nhiều trường hợp, cả hai đều hoạt động. Ưu tiên `interface` cho đối tượng vì tính phổ biến và dễ bảo trì.

Ví dụ minh họa:

```
interface AnimalInterface {
    name: string;
```

```
    makeSound(): void;
}

type AnimalType = {
    name: string;
    makeSound(): void;
};
```

4. Code ví dụ tổng hợp

```
// ---- Interfaces ----
interface Vehicle {
    readonly id: string;
    brand: string;
    model: string;
    year: number;
    start(): void;
    stop(): void;
    getDetails(): string;
    serviceDate?: Date;
}

interface ElectricVehicle extends Vehicle {
    batteryLevel: number;
    charge(): void;
}

class Car implements Vehicle {
    readonly id: string;
    brand: string;
    model: string;
    year: number;
    serviceDate?: Date;

    constructor(id: string, brand: string, model: string, year: number) {
        this.id = id;
        this.brand = brand;
        this.model = model;
        this.year = year;
```

```
    }

    start(): void {
        console.log(`${this.brand} ${this.model} started.`);
    }

    stop(): void {
        console.log(`${this.brand} ${this.model} stopped.`);
    }

    getDetails(): string {
        return `ID: ${this.id}, Brand: ${this.brand}, Model: ${this.model},
    }
}

let myCar = new Car("CAR-001", "Toyota", "Corolla", 2022);
console.log(myCar.getDetails());
myCar.start();

let myTesla: ElectricVehicle = {
    id: "EV-002",
    brand: "Tesla",
    model: "Model S",
    year: 2023,
    batteryLevel: 85,
    start: () => console.log("Tesla Model S started silently."),
    stop: () => console.log("Tesla Model S stopped."),
    getDetails: function() { return `EV: ${this.brand} ${this.model}, Batter
    charge: () => console.log("Charging Tesla Model S...")
};
console.log(myTesla.getDetails());
myTesla.charge();

// ---- Type Aliases ----
type UserID = string | number;
type ProductStatus = "Available" | "OutOfStock" | "Discontinued";

type Product = {
    sku: string;
    name: string;
    price: number;
    status: ProductStatus;
    tags?: string[];
```

```
};

let product1: Product = {
    sku: "SKU123",
    name: "Awesome Gadget",
    price: 99.99,
    status: "Available",
    tags: ["tech", "gadget"]
};

function displayProduct(product: Product): void {
    console.log(`Product: ${product.name} (SKU: ${product.sku})`);
    console.log(`Price: $${product.price}, Status: ${product.status}`);
    if (product.tags) {
        console.log(`Tags: ${product.tags.join(", ")}`);
    }
}

displayProduct(product1);
```

Giải thích:

- Mã trên minh họa cách sử dụng `interface` và `type alias` trong các tình huống thực tế, như mô tả xe cộ (`Vehicle` , `ElectricVehicle`) và sản phẩm (`Product`).
- **Ứng dụng thực tế**: Có thể dùng trong ứng dụng quản lý kho hoặc giao diện hiển thị danh sách sản phẩm.

Danh sách bài tập

1. Trắc nghiệm: Kiến thức về Interfaces

Câu hỏi: Phát biểu nào sau đây là ĐÚNG về Interfaces trong TypeScript?

- A. Interface chỉ có thể được sử dụng để mô tả hình dạng của đối tượng, không dùng cho class.

- B. Interface có thể kế thừa từ nhiều interface khác.
- C. Interface không hỗ trợ thuộc tính tùy chọn (`?`).
- D. Interface không thể được hợp nhất (declaration merging).

Đáp án: B
Giải thích:

- A sai vì interface có thể dùng cho class (`implements`).
- B đúng vì interface hỗ trợ kế thừa nhiều interface bằng `extends` .
- C sai vì interface hỗ trợ thuộc tính tùy chọn (`?`).
- D sai vì interface hỗ trợ declaration merging.

2. Code: Tạo interface `Product`

Mô tả: Tạo `interface Product` với các thuộc tính:

- `id` (number, readonly)
- `name` (string)
- `price` (number)
- `description` (string, tùy chọn)
- `inStock` (boolean)

Tạo đối tượng `sampleProduct` tuân theo interface này.

Giải pháp mẫu:

```
interface Product {
    readonly id: number;
    name: string;
    price: number;
    description?: string;
    inStock: boolean;
}
```

```
let sampleProduct: Product = {
    id: 1,
    name: "Tai nghe Bluetooth XYZ",
    price: 1200000,
    description: "Tai nghe chống ồn, pin trâu",
    inStock: true
};

console.log("Sản phẩm mẫu:", sampleProduct);
```

Giải thích:

- Interface đảm bảo đối tượng có đúng cấu trúc và kiểu dữ liệu.
- Thuộc tính `description` là tùy chọn, nên có thể bỏ qua.

3. Code: Tạo type alias `HttpCode`

Mô tả: Tạo `type alias HttpCode` cho kiểu `number`. Khai báo biến `responseStatus` và gán giá trị mã HTTP (200, 404, 500).

Giải pháp mẫu:

```
type HttpCode = number;

let responseStatus: HttpCode = 200;
console.log("Response Status:", responseStatus);

responseStatus = 404; // OK
console.log("New Response Status:", responseStatus);
```

Giải thích:

- `HttpCode` là bí danh cho `number`, giúp mã rõ ràng hơn khi làm việc với mã trạng thái HTTP.

4. Code: Kế thừa interface

Mô tả: Tạo `interface BookProduct` kế thừa từ `Product`, thêm:

- `author` (string)
- `pages` (number)

Tạo đối tượng `programmingBook` tuân theo `BookProduct`.

Giải pháp mẫu:

```
interface Product {
    readonly id: number;
    name: string;
    price: number;
    description?: string;
    inStock: boolean;
}

interface BookProduct extends Product {
    author: string;
    pages: number;
}

let programmingBook: BookProduct = {
    id: 10,
    name: "Effective TypeScript",
    price: 750000,
    inStock: true,
    author: "Dan Vanderkam",
    pages: 322
};

console.log("Sách lập trình:", programmingBook);
```

Giải thích:

- `BookProduct` kế thừa tất cả thuộc tính của `Product` và thêm `author`, `pages`.

- **Ứng dụng thực tế**: Dùng để mô tả sản phẩm sách trong ứng dụng bán hàng.

5. Code: Type alias cho API Response

Mô tả: Tạo `type alias ApiResponse<TData>` với:

- `success` (boolean)
- `data` (TData)
- `message` (string, tùy chọn)
- `errorCode` (number, tùy chọn)

Tạo `successResponse` và `errorResponse`.

Giải pháp mẫu:

```
type ApiResponse<TData> = {
    success: boolean;
    data: TData;
    message?: string;
    errorCode?: number;
};

type UserData = { userId: string; username: string };

let successUserResponse: ApiResponse<UserData> = {
    success: true,
    data: { userId: "user123", username: "ts_pro" },
    message: "User data fetched successfully."
};

let errorUserResponse: ApiResponse<null> = {
    success: false,
    data: null,
    message: "User not found.",
    errorCode: 404
};
```

```
console.log("Success Response:", successUserResponse);
console.log("Error Response:", errorUserResponse);
```

Giải thích:

- Generic `TData` cho phép `data` có kiểu linh hoạt.
- **Ứng dụng thực tế**: Dùng để mô tả phản hồi từ API trong ứng dụng web.

Bài tập bổ sung

6. Code: Intersection Types với Type Alias

Mô tả: Tạo hai `type alias`:

- `BasicInfo`: `{ name: string; age: number }`
- `ContactInfo`: `{ email: string; phone?: string }`

Tạo `type alias Person` là intersection của `BasicInfo` và `ContactInfo`. Tạo đối tượng `person` tuân theo `Person`.

Giải pháp mẫu:

```
type BasicInfo = {
    name: string;
    age: number;
};

type ContactInfo = {
    email: string;
    phone?: string;
};

type Person = BasicInfo & ContactInfo;
```

```
let person: Person = {
    name: "Nguyen Van A",
    age: 25,
    email: "nva@example.com",
    phone: "0123456789"
};

console.log("Person:", person);
```

Giải thích:

- Intersection type (`&`) kết hợp các thuộc tính của cả hai type alias.
- **Ứng dụng thực tế**: Dùng để mô tả đối tượng có nhiều nhóm thông tin, như hồ sơ người dùng.