Defining Types (Định nghĩa kiểu dữ liệu)

JavaScript cho phép áp dụng nhiều **design pattern** khác nhau trong quá trình thiết kế và tổ chức mã nguồn. Tuy nhiên, một số design pattern—đặc biệt là các pattern dựa trên hành vi động, chẳng hạn như **dynamic programming** hoặc thao tác cấu trúc dữ liệu thay đổi tại runtime—khiến cho việc **suy luận kiểu tự động (type inference)** trở nên khó khăn hoặc không khả thi.

Để giải quyết các trường hợp này, TypeScript mở rộng cú pháp của JavaScript bằng cách cung cấp các vị trí rõ ràng, nơi lập trình viên có thể **khai báo tường minh kiểu dữ liệu**, giúp hệ thống type system hoạt động chính xác và chặt chẽ hơn.

Suy luận kiểu từ object literal

Khi khởi tạo một object literal, TypeScript có thể tự động suy luận **shape (hình dạng)** của object dựa trên các property và giá trị ban đầu.

Ví dụ:

```
const user = {
  name: "Hayes",
  id: 0,
};
```

Trong trường hợp này, TypeScript suy luận rằng `user` là một object có:

- `name` thuộc kiểu `string`
- `id` thuộc kiểu `number`

Mô tả shape bằng interface

Để mô tả tường minh cấu trúc của object, TypeScript cung cấp **interface declaration**. Interface dùng để định nghĩa chính xác tập hợp các property và kiểu dữ liệu tương ứng mà một object phải tuân theo.

Ví dụ:

```
interface User {
  name: string;
  id: number;
}
```

Interface trên định nghĩa một contract: bất kỳ object nào được xem là `User` đều phải có đúng hai property `name` và `id` với kiểu tương ứng.

Gán object cho interface

Sau khi khai báo interface, ta có thể chỉ định rằng một biến phải tuân theo interface đó bằng cách sử dụng cú pháp `: TypeName` sau tên biến:

```
const user: User = {
  name: "Hayes",
  id: 0,
};
```

Tại đây, TypeScript sẽ kiểm tra **tính tương thích cấu trúc (structural compatibility)** giữa object literal và interface `User`.

Nếu object không khớp với interface đã khai báo, TypeScript sẽ phát hiện và cảnh báo lỗi tại thời điểm biên dịch.

Ví dụ:

```
interface User {
  name: string;
  id: number;
}

const user: User = {
  username: "Hayes",
  id: 0,
};
```

Lỗi phát sinh do property `username` không tồn tại trong interface `User`. Điều này phản ánh nguyên tắc **excess property checking** của TypeScript đối với object literal.

Interface và lập trình hướng đối tượng

Do JavaScript hỗ trợ **class** và mô hình lập trình hướng đối tượng, TypeScript cho phép interface được sử dụng như một hợp đồng cho class.

Ví dụ:

```
interface User {
  name: string;
  id: number;
}

class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

Trong trường hợp này, class `UserAccount` được xem là **tương thích với interface** **`User`** vì nó cung cấp đầy đủ các property yêu cầu, bất kể class không khai báo từ khóa `implements`.

Interface trong hàm

Interface cũng có thể được sử dụng để **chú thích kiểu cho tham số và giá trị trả về của function**, giúp tăng tính rõ ràng và an toàn kiểu.

Ví dụ:

```
function deleteUser(user: User) {
  // ...
}

function getAdminUser(): User {
```

```
  // ...
}
```

Cách tiếp cận này đảm bảo rằng dữ liệu truyền vào và trả ra tuân thủ đúng cấu trúc đã được định nghĩa.

Các kiểu dữ liệu cơ bản và mở rộng

JavaScript cung cấp một tập nhỏ các **primitive types**, bao gồm:

- `boolean`
- `bigint`
- `null`
- `number`
- `string`
- `symbol`
- `undefined`

Các kiểu này có thể được sử dụng trực tiếp trong interface và type annotation.

TypeScript mở rộng hệ thống kiểu này bằng cách bổ sung thêm một số kiểu đặc biệt:

- `any`: cho phép mọi giá trị, bỏ qua kiểm tra kiểu
- `unknown`: yêu cầu phải kiểm tra hoặc xác định kiểu trước khi sử dụng
- `never`: biểu thị trường hợp không thể xảy ra
- `void`: thường dùng cho function không trả về giá trị (hoặc trả về `undefined`)

Interface và Type alias

TypeScript cung cấp **hai cú pháp chính để xây dựng kiểu dữ liệu**:

- `interface`
- `type`

Trong thực hành chuẩn, **interface nên được ưu tiên sử dụng** do khả năng mở rộng, hợp nhất khai báo (declaration merging) và phù hợp với mô hình thiết kế hướng đối tượng.
`type` chỉ nên sử dụng khi cần các tính năng đặc thù mà interface không hỗ trợ, chẳng hạn như union type, intersection type, hoặc mapped type phức tạp.

Cách phân biệt này giúp duy trì tính nhất quán, khả năng mở rộng và độ rõ ràng của hệ thống kiểu trong các hệ thống TypeScript quy mô lớn.