Chapter 3: Functions trong TypeScript

Mô tả tổng quát

Chapter này tập trung vào cách **TypeScript** cải thiện việc làm việc với hàm so với JavaScript thuần. Chúng ta sẽ khám phá cách khai báo kiểu (type) cho tham số và giá trị trả về của hàm, tìm hiểu các loại tham số đặc biệt như **tham số tùy chọn**, **tham số mặc định**, và **rest parameters**. Ngoài ra, các khái niệm như **kiểu hàm (function types)** và **nạp chồng hàm (function overloads)** cũng được giới thiệu, giúp bạn viết code hàm an toàn, dễ bảo trì và linh hoạt hơn.

Mục tiêu học tập:

- Hiểu cách sử dụng kiểu dữ liệu trong hàm để tăng tính an toàn và rõ ràng.
- Làm quen với các tính năng đặc biệt của hàm trong TypeScript.
- Áp dụng function overloads và function types vào các tình huống thực tế.

Tiêu đề Chapter học

Functions Nâng Cao: Kiểu Trả Về, Tham Số và Overloads

Tóm tắt lý thuyết chính

1. Định nghĩa kiểu cho tham số và giá trị trả về

TypeScript cho phép khai báo kiểu dữ liệu cụ thể cho **tham số** và **giá trị trả về** của hàm, giúp trình biên dịch phát hiện lỗi ngay trong quá trình phát triển. Điều này làm cho mã nguồn rõ ràng hơn và giảm nguy cơ lỗi runtime.

Ví dụ:

```
function add(x: number, y: number): number {
    return x + y;
}

let sum: number = add(5, 10); // sum = 15
// add(5, "10"); // Lỗi: Argument of type 'string' is not assignable to para
```

Giải thích chi tiết:

- `x: number, y: number` : Khai báo kiểu `number` cho tham số `x` và `y` . Nếu truyền sai kiểu (ví dụ: chuỗi `"10"`), TypeScript sẽ báo lỗi ngay lập tức.
- `: number` sau danh sách tham số chỉ định hàm phải trả về một giá trị kiểu `number` .
- Lợi ích: Đảm bảo hàm chỉ nhận và trả về dữ liệu đúng kiểu, tránh lỗi logic khi chạy chương trình.

2. Kiểu `void` cho hàm không trả về giá trị

Nếu hàm không trả về bất kỳ giá trị nào hoặc chỉ trả về `undefined` một cách ngầm định, bạn nên khai báo kiểu trả về là `void` .

Ví dụ:

```
function logMessage(message: string): void {
    console.log(message);
    // Không có return hoặc chỉ có return; (trả về undefined ngầm định)
}
logMessage("Đây là một thông báo.");
```

Giải thích chi tiết:

- Kiểu `void` biểu thị rằng hàm không trả về giá trị có ý nghĩa nào. Nếu hàm cố gắng trả về một giá trị (ví dụ: `return 42;`), TypeScript sẽ báo lỗi.

- Thường dùng cho các hàm chỉ thực hiện tác vụ như in ra console, cập nhật UI, hoặc gửi request mạng mà không cần trả về dữ liệu.

3. Tham số tùy chọn (`?`) và tham số mặc định

Tham số tùy chọn

Tham số tùy chọn được đánh dấu bằng dấu `?` sau tên tham số, cho phép người gọi hàm bỏ qua tham số đó. Tham số tùy chọn phải nằm **sau** các tham số bắt buộc trong danh sách tham số.

Ví dụ:

```
function greetOptional(name: string, greeting?: string): string {
    if (greeting) {
        return `${greeting}, ${name}!`;
    } else {
        return `Hello, ${name}!`;
    }
}
console.log(greetOptional("Alice")); // "Hello, Alice!"
console.log(greetOptional("Bob", "Hi")); // "Hi, Bob!"
```

Giải thích chi tiết:

- `greeting?: string` : Tham số `greeting` là tùy chọn. Nếu không truyền, giá trị của nó sẽ là `undefined` .
- Trong hàm, cần kiểm tra xem `greeting` có giá trị hay không trước khi sử dụng (thường dùng `if` hoặc toán tử `??`).
- Lưu ý: Nếu đặt tham số tùy chọn trước tham số bắt buộc, TypeScript sẽ báo lỗi.

Tham số mặc định

Tham số mặc định cho phép gán một giá trị mặc định cho tham số nếu người gọi không truyền giá trị hoặc truyền `undefined` . Tham số mặc định cũng được coi là tùy chọn.

Ví dụ:

```
function greetDefault(name: string, greeting: string = "Hello"): string {
    return `${greeting}, ${name}!`;
}
console.log(greetDefault("Charlie")); // "Hello, Charlie!"
console.log(greetDefault("David", "Good morning")); // "Good morning, David!
```

Giải thích chi tiết:

- `greeting: string = "Hello"` : Nếu không truyền `greeting` , nó sẽ lấy giá trị mặc định là `"Hello"` .
- Tham số mặc định giúp giảm bớt sự phức tạp khi gọi hàm, đặc biệt khi một số tham số thường có giá trị giống nhau.
- Lưu ý: Tham số mặc định không cần dấu `?` , vì chúng đã là tùy chọn.

4. Rest Parameters (`...`)

Rest parameters cho phép hàm chấp nhận một số lượng tham số không xác định, được gom lại thành một mảng. Rest parameter phải là tham số cuối cùng và có kiểu là một mảng.

Ví dụ:

```
function sumAllNumbers(...numbers: number[]): number {
    let total = 0;
    for (const num of numbers) {
        total += num;
    }
    return total;
}
console.log(sumAllNumbers(1, 2, 3)); // 6
console.log(sumAllNumbers(10, 20, 30, 40, 50)); // 150
```

Giải thích chi tiết:

- `...numbers: number[]` : Cú pháp `...` (spread operator) thu thập tất cả các tham số được truyền vào thành một mảng `numbers` kiểu `number[]` .
- Rest parameter phải là tham số cuối cùng, vì sau nó không thể có tham số nào khác.
- Ứng dụng: Dùng khi hàm cần xử lý số lượng tham số linh hoạt, như tính tổng, nối chuỗi, hoặc xử lý danh sách.

5. Function Types (Kiểu hàm)

Kiểu hàm (function types) cho phép định nghĩa chữ ký (signature) của một hàm, bao gồm kiểu của tham số và giá trị trả về. Điều này hữu ích khi gán hàm cho biến hoặc truyền hàm làm tham số (callback).

Ví dụ:

```
let multiply: (a: number, b: number) => number;

multiply = function(x: number, y: number): number {
    return x * y;
};
// Hoặc dùng arrow function
// multiply = (x, y) => x * y;

console.log(multiply(5, 4)); // 20
// multiply = function(message: string): void { console.log(message); }; //
```

Giải thích chi tiết:

- `(a: number, b: number) => number` : Định nghĩa kiểu hàm với hai tham số kiểu `number` và trả về giá trị kiểu `number` .
- Biến `multiply` chỉ có thể được gán bởi một hàm có chữ ký phù hợp. Nếu gán hàm không khớp kiểu, TypeScript sẽ báo lỗi.
- Ứng dụng: Dùng trong callback, event handler, hoặc khi cần tái sử dụng hàm với chữ ký cố định.

6. Arrow Functions và kiểu

Arrow functions trong TypeScript có cú pháp giống JavaScript ES6, nhưng TypeScript có thể suy luận kiểu hoặc cho phép khai báo kiểu tường minh.

Ví dụ:

```
const subtract = (x: number, y: number): number => {
    return x - y;
};

// TypeScript suy luận kiểu nếu gán cho biến có kiểu hàm
let divide: (a: number, b: number) => number;
divide = (n1, n2) => n1 / n2; // n1, n2 tự động suy luận là number
console.log(divide(10, 2)); // 5
```

Giải thích chi tiết:

- Arrow function `(x: number, y: number): number => { ... }` có cú pháp ngắn gọn và dễ đọc.
- TypeScript có thể suy luận kiểu của tham số và giá trị trả về nếu biến đã được khai báo với kiểu hàm cụ thể.
- Lưu ý: Nếu không khai báo kiểu rõ ràng, cần đảm bảo ngữ cảnh đủ để TypeScript suy luận đúng.

7. Function Overloads (Nạp chồng hàm)

Function overloads cho phép định nghĩa nhiều chữ ký hàm (overload signatures) cho cùng một tên hàm. Chữ ký thực thi (implementation signature) phải bao quát tất cả các chữ ký nạp chồng. TypeScript sẽ chọn chữ ký phù hợp nhất dựa trên tham số truyền vào.

Ví dụ:

```typescript
// Chữ ký nạp chồng
function processInput(input: string): string;
function processInput(input: number): number;
function processInput(input: string[]): string[];

// Chữ ký thực thi
function processInput(input: string | number | string[]): string | number |
    if (typeof input === "string") {
        return input.toUpperCase();
    } else if (typeof input === "number") {
        return input * 2;
    } else if (Array.isArray(input)) {
        return input.map(item => item.trim());
    }
    throw new Error("Invalid input type");
}

console.log(processInput("hello"));   // "HELLO"
console.log(processInput(10));        // 20
console.log(processInput([" one ", " two "])); // ["one", "two"]
// console.log(processInput(true)); // Lỗi: No overload matches this call.
```

Giải thích chi tiết:

- **Chữ ký nạp chồng**: Mỗi chữ ký định nghĩa một trường hợp cụ thể (ví dụ: nhận `string` trả `string`, nhận `number` trả `number`).
- **Chữ ký thực thi**: Phải đủ tổng quát để bao quát tất cả các chữ ký nạp chồng (dùng `string | number | string[]`).
- TypeScript sử dụng chữ ký nạp chồng để kiểm tra kiểu khi gọi hàm, nhưng mã thực thi nằm trong chữ ký thực thi.
- Ứng dụng: Dùng khi hàm có thể xử lý nhiều kiểu đầu vào và trả về các kiểu khác nhau, ví dụ: xử lý dữ liệu đa dạng trong API.

Code ví dụ chính (Tổng hợp và mở rộng)

```typescript
// Hàm cơ bản với kiểu tham số và trả về
function createGreeting(name: string, age: number): string {
    return `Hello, my name is ${name} and I am ${age} years old.`;
}
console.log(createGreeting("Elena", 28)); // "Hello, my name is Elena and I

// Hàm với tham số tùy chọn và mặc định
function displayProfile(username: string, bio?: string, theme: string = "lig
    console.log(`Username: ${username}`);
    if (bio) {
        console.log(`Bio: ${bio}`);
    }
    console.log(`Theme: ${theme}`);
}
displayProfile("tsDev"); // Username: tsDev, Theme: light
displayProfile("jsFan", "Loves JavaScript and learning TypeScript."); // Use
displayProfile("coderX", "Full-stack developer.", "dark"); // Username: code

// Hàm sử dụng rest parameters
function gatherSkills(primarySkill: string, ...otherSkills: string[]): void
    console.log(`Primary Skill: ${primarySkill}`);
    if (otherSkills.length > 0) {
        console.log(`Other Skills: ${otherSkills.join(", ")}`);
    }
}
gatherSkills("TypeScript", "JavaScript", "React", "Node.js"); // Primary Ski
gatherSkills("Problem Solving"); // Primary Skill: Problem Solving

// Định nghĩa kiểu hàm và sử dụng
type StringManipulator = (input: string) => string;

const toUpperCaseConverter: StringManipulator = (text) => text.toUpperCase()
const toLowerCaseConverter: StringManipulator = (text) => text.toLowerCase()

function applyStringOperation(text: string, manipulator: StringManipulator):
    return manipulator(text);
}
console.log(applyStringOperation("Hello World", toUpperCaseConverter)); // "
console.log(applyStringOperation("Hello World", toLowerCaseConverter)); // "

// Function Overloads ví dụ phức tạp hơn
interface Coordinate { x: number; y: number; }
```

```ts
function parseCoordinate(obj: Coordinate): Coordinate;
function parseCoordinate(x: number, y: number): Coordinate;
function parseCoordinate(arg1: unknown, arg2?: unknown): Coordinate {
    let coord: Coordinate = { x: 0, y: 0 };

    if (typeof arg1 === 'object' && arg1 !== null) {
        coord = { ...(arg1 as Coordinate) }; // Ép kiểu để truy cập thuộc tí
    } else if (typeof arg1 === 'number' && typeof arg2 === 'number') {
        coord = { x: arg1, y: arg2 };
    } else {
        throw new Error("Invalid arguments for parseCoordinate");
    }
    return coord;
}

console.log(parseCoordinate({ x: 10, y: 20 })); // { x: 10, y: 20 }
console.log(parseCoordinate(5, 15)); // { x: 5, y: 15 }
// console.log(parseCoordinate("5", "15")); // Lỗi: No overload matches this
```

Danh sách bài tập

1. Trắc nghiệm: Tham số `…args: string[]` trong một hàm có ý nghĩa gì?

Tiêu đề: Hiểu về Rest Parameters

Mô tả: Chọn mô tả đúng nhất cho cú pháp rest parameters.

Câu hỏi: Trong một hàm TypeScript, khai báo tham số `…args: string[]` có ý nghĩa gì?

- **A.** Hàm chỉ chấp nhận một tham số duy nhất là một mảng các chuỗi.
- **B.** Hàm chấp nhận một số lượng bất kỳ các tham số kiểu chuỗi, và chúng được gom lại thành một mảng tên là `args`.
- **C.** `args` là một tham số tùy chọn kiểu mảng chuỗi.
- **D.** Hàm yêu cầu ít nhất một tham số kiểu chuỗi, các tham số sau đó có thể là bất kỳ kiểu gì.

Đáp án: B

Giải thích chi tiết: Rest parameters (`...args: string[]`) cho phép hàm nhận một số lượng không xác định các tham số kiểu chuỗi, và tất cả chúng được gom vào một mảng tên `args` . Đây không phải là tham số tùy chọn (C), không yêu cầu tham số đầu tiên bắt buộc (D), và không phải chỉ nhận một mảng (A).

2. Code: Viết hàm tính diện tích hình chữ nhật

Tiêu đề: Thực hành định nghĩa kiểu cho hàm

Mô tả: Viết hàm `calculateRectangleArea` nhận vào hai tham số `width` và `height` (kiểu `number`), trả về diện tích hình chữ nhật (kiểu `number`).

Yêu cầu:

- Định nghĩa hàm với kiểu tham số và trả về rõ ràng.
- Gọi hàm với một vài giá trị và in kết quả.

Giải pháp mẫu:

```
function calculateRectangleArea(width: number, height: number): number {
    if (width ⩽ 0 || height ⩽ 0) {
        console.warn("Width and height must be positive numbers.");
        return 0;
    }
    return width * height;
}

console.log("Diện tích (5x10):", calculateRectangleArea(5, 10)); // 50
console.log("Diện tích (7x3):", calculateRectangleArea(7, 3)); // 21
console.log("Diện tích (-2x5):", calculateRectangleArea(-2, 5)); // Cảnh báo
```

Giải thích chi tiết:

- Hàm kiểm tra xem `width` và `height` có hợp lệ (lớn hơn 0) để tránh kết quả không mong muốn.
- Kiểu `number` được khai báo rõ ràng cho cả tham số và giá trị trả về, đảm bảo an toàn kiểu.
- In kết quả giúp kiểm tra hàm hoạt động đúng.

3. Code: Viết hàm tạo thông tin người dùng với tham số mặc định

Tiêu đề: Thực hành với tham số mặc định

Mô tả: Viết hàm `createUserInfo` nhận vào `userId` (number, bắt buộc), `username` (string, bắt buộc) và `role` (string, mặc định là "user"). Hàm trả về chuỗi mô tả thông tin người dùng.

Yêu cầu:

- Định nghĩa hàm với các tham số và kiểu như mô tả.
- Gọi hàm với đủ tham số và không truyền `role` để kiểm tra giá trị mặc định.

Giải pháp mẫu:

```
function createUserInfo(userId: number, username: string, role: string = "us
    return `User ID: ${userId}, Username: ${username}, Role: ${role}`;
}

console.log(createUserInfo(1, "john.doe")); // User ID: 1, Username: john.do
console.log(createUserInfo(2, "jane.admin", "administrator")); // User ID: 2
```

Giải thích chi tiết:

- `role: string = "user"` : Tham số `role` có giá trị mặc định, nên không bắt buộc truyền khi gọi hàm.
- Hàm trả về chuỗi định dạng thông tin người dùng, dễ dàng kiểm tra giá trị mặc định và giá trị truyền vào.

4. Code: Định nghĩa kiểu hàm và sử dụng

Tiêu đề: Thực hành với Function Types

Mô tả:

1. Định nghĩa kiểu hàm `NumberOperation` nhận hai số và trả về một số.
2. Tạo hai biến kiểu `NumberOperation` : `addOperation` (phép cộng) và `subtractOperation` (phép trừ).
3. Gọi và in kết quả của hai biến hàm này.

Giải pháp mẫu:

```
type NumberOperation = (a: number, b: number) => number;

const addOperation: NumberOperation = (x, y) => x + y;
const subtractOperation: NumberOperation = (x, y) => x - y;

console.log("5 + 3 =", addOperation(5, 3)); // 8
console.log("10 - 4 =", subtractOperation(10, 4)); // 6
```

Giải thích chi tiết:

- Kiểu `NumberOperation` định nghĩa chữ ký hàm với hai tham số `number` và trả về `number` .
- Hai arrow function `addOperation` và `subtractOperation` tuân thủ chữ ký này.
- Gọi hàm thông qua biến giúp tái sử dụng và kiểm tra tính đúng đắn của kiểu.

5. Code: Viết hàm với Function Overloads

Tiêu đề: Thực hành với Function Overloads

Mô tả: Viết hàm `formatData` sử dụng function overloads:

- Nếu đầu vào là `string` , trả về chuỗi viết hoa.

- Nếu đầu vào là `number`, trả về chuỗi "Number: [số đó]".
- Nếu đầu vào là `boolean`, trả về chuỗi "Boolean: [true/false]".

Yêu cầu:

- Viết 3 chữ ký nạp chồng cho các trường hợp trên.
- Viết chữ ký thực thi bao quát tất cả.
- Gọi hàm với các kiểu đầu vào khác nhau và in kết quả.

Giải pháp mẫu:

```
function formatData(input: string): string;
function formatData(input: number): string;
function formatData(input: boolean): string;
function formatData(input: string | number | boolean): string {
    if (typeof input === "string") {
        return input.toUpperCase();
    } else if (typeof input === "number") {
        return `Number: ${input}`;
    } else if (typeof input === "boolean") {
        return `Boolean: ${input}`;
    }
    const exhaustiveCheck: never = input;
    return `Unknown type: ${exhaustiveCheck}`;
}

console.log(formatData("typescript")); // "TYPESCRIPT"
console.log(formatData(2024)); // "Number: 2024"
console.log(formatData(true)); // "Boolean: true"
// console.log(formatData({})); // Lỗi: No overload matches this call.
```

Giải thích chi tiết:

- Ba chữ ký nạp chồng định nghĩa rõ các trường hợp đầu vào và đầu ra.
- Chữ ký thực thi sử dụng `string | number | boolean` để bao quát tất cả kiểu đầu vào.

- Biến `exhaustiveCheck: never` giúp đảm bảo không có trường hợp nào bị bỏ sót (TypeScript sẽ báo lỗi nếu có kiểu đầu vào mới không được xử lý).
- Ứng dụng: Xử lý đa dạng kiểu dữ liệu trong một hàm, thường thấy trong các thư viện hoặc API.

Kết luận

Chapter này cung cấp nền tảng vững chắc để làm việc với hàm trong TypeScript, từ việc định nghĩa kiểu cơ bản đến các tính năng nâng cao như function overloads và rest parameters. Các bài tập thực hành giúp bạn củng cố kiến thức và áp dụng vào các tình huống thực tế. Hãy luyện tập nhiều để làm quen với cú pháp và cách TypeScript giúp code an toàn hơn!