Chapter 6: Classes trong TypeScript

Mô tả tổng quát

Chapter này sẽ giới thiệu về lập trình hướng đối tượng (OOP) trong TypeScript, một cách tiếp cận giúp tổ chức code hiệu quả, dễ tái sử dụng và bảo trì. TypeScript mở rộng khái niệm `class` của JavaScript ES6 với các tính năng mạnh mẽ như **access modifiers** (`public` , `private` , `protected`), **readonly properties**, **static members**, **inheritance**, và **abstract classes**. Mục tiêu là giúp bạn hiểu và áp dụng các khái niệm này vào dự án thực tế.

Tiêu đề Chapter

Lập Trình Hướng Đối Tượng với Classes

Tóm tắt lý thuyết chính

1. Khai báo Class, Properties, Methods và Constructors

- **Class**: Là bản thiết kế để tạo ra các đối tượng (instances). Mỗi đối tượng được tạo từ class sẽ có các thuộc tính (properties) và hành vi (methods) được định nghĩa trong class.
- **Properties**: Là các biến thuộc về class, lưu trữ trạng thái của đối tượng.
- **Methods**: Là các hàm thuộc về class, định nghĩa hành vi của đối tượng.
- **Constructor**: Phương thức đặc biệt, được gọi tự động khi tạo đối tượng bằng từ khóa `new` . Thường dùng để khởi tạo các thuộc tính.

Ví dụ:

```
class Greeter {
    greeting: string; // Property

    constructor(message: string) { // Constructor
```

```
        this.greeting = message;
        console.log("Greeter object created!");
    }

    greet(): string { // Method
        return `Hello, ${this.greeting}`;
    }
}

const greeter = new Greeter("world"); // In: "Greeter object created!"
console.log(greeter.greet()); // In: "Hello, world"
```

Giải thích:

- `greeting` là thuộc tính lưu trữ chuỗi chào hỏi.
- `constructor` nhận tham số `message` và gán nó cho `greeting`.
- `greet()` là phương thức trả về chuỗi chào hỏi.

2. Access Modifiers (Bộ điều chỉnh truy cập)

TypeScript cung cấp 3 access modifier để kiểm soát quyền truy cập vào thuộc tính và phương thức:

- `public`: (Mặc định) Có thể truy cập từ bất kỳ đâu.
- `private`: Chỉ truy cập được trong class khai báo nó. Không thể truy cập từ instance hoặc class con.
- `protected`: Có thể truy cập trong class khai báo và các class con, nhưng không thể từ instance bên ngoài.

Ví dụ:

```
class Animal {
    public name: string;        // Truy cập từ bất kỳ đâu
    private age: number;        // Chỉ trong class Animal
    protected sound: string;    // Trong Animal và class con
```

```
    constructor(name: string, age: number, sound: string) {
        this.name = name;
        this.age = age;
        this.sound = sound;
    }

    public makeSound(): void {
        console.log(`${this.name} makes a ${this.sound} sound.`);
    }

    private getAge(): number {
        return this.age;
    }

    public displayAge(): void {
        console.log(`${this.name} is ${this.getAge()} years old.`);
    }
}

class Cat extends Animal {
    constructor(name: string, age: number) {
        super(name, age, "Meow");
    }

    public purr(): void {
        console.log(`${this.name} is purring. It says ${this.sound}.`); // O
        // console.log(this.age); // Lỗi: age là private
    }
}

const pet = new Animal("Doggy", 5, "Woof");
console.log(pet.name); // OK: "Doggy"
pet.makeSound();       // OK: "Doggy makes a Woof sound."
// console.log(pet.age); // Lỗi: age là private
pet.displayAge();      // OK: "Doggy is 5 years old."

const cat = new Cat("Whiskers", 3);
cat.purr(); // OK: "Whiskers is purring. It says Meow."
```

Giải thích:

- `name` (`public`) có thể truy cập từ bất kỳ đâu, kể cả từ instance (`pet.name`).
- `age` (`private`) chỉ có thể truy cập trong class `Animal`, thông qua phương thức `getAge()` hoặc `displayAge()`.
- `sound` (`protected`) có thể truy cập trong class `Animal` và class con `Cat` (qua `this.sound`).

Lưu ý thực tiễn:

- Sử dụng `private` khi bạn muốn ẩn chi tiết triển khai (encapsulation).
- Sử dụng `protected` khi cần chia sẻ dữ liệu giữa class cha và class con, nhưng không muốn public.

3. Readonly Properties (Thuộc tính chỉ đọc)

Thuộc tính `readonly` chỉ có thể được gán giá trị khi khai báo hoặc trong constructor. Sau đó, giá trị không thể thay đổi.

Ví dụ:

```
class Configuration {
    readonly apiKey: string;
    readonly apiVersion: string = "v1";

    constructor(key: string) {
        this.apiKey = key;
    }

    updateKey(newKey: string) {
        // this.apiKey = newKey; // Lỗi: apiKey là readonly
    }
}

const config = new Configuration("XYZ123");
console.log(config.apiKey); // "XYZ123"
// config.apiVersion = "v2"; // Lỗi: apiVersion là readonly
```

Giải thích:

- `apiKey` được gán trong constructor, sau đó không thể thay đổi.
- `apiVersion` được gán giá trị mặc định ngay khi khai báo.

Lưu ý thực tiễn:

- Sử dụng `readonly` để đảm bảo dữ liệu bất biến (immutable), ví dụ: khóa API hoặc cấu hình cố định.

4. Getters và Setters (Phương thức truy cập)

Getters và setters cho phép kiểm soát việc đọc và ghi giá trị của thuộc tính, thường dùng với thuộc tính `private`.

Ví dụ:

```
class Employee {
    private _fullName: string = "";

    get fullName(): string {
        console.log("Getter called.");
        return this._fullName;
    }

    set fullName(newName: string) {
        console.log("Setter called.");
        if (newName && newName.length > 0) {
            this._fullName = newName;
        } else {
            console.error("Full name cannot be empty.");
        }
    }
}

const emp = new Employee();
emp.fullName = "John Smith"; // Gọi setter
console.log(emp.fullName);   // Gọi getter: "John Smith"
```

```
emp.fullName = "";           // Gọi setter, in lỗi
console.log(emp.fullName);   // Vẫn là "John Smith"
```

Giải thích:

- `_fullName` là thuộc tính private, được quản lý bởi getter (`get fullName`) và setter (`set fullName`).
- Setter kiểm tra `newName` để đảm bảo không rỗng trước khi gán.

Lưu ý thực tiễn:

- Đặt tên thuộc tính private với tiền tố `_` (ví dụ: `_fullName`) là quy ước phổ biến.
- Sử dụng getter/setter khi cần thêm logic kiểm tra hoặc xử lý trước khi đọc/ghi.

5. Static Members (Thành viên tĩnh)

Thành viên `static` thuộc về class, không thuộc về instance. Truy cập trực tiếp qua tên class.

Ví dụ:

```
class MathHelper {
    static readonly PI: number = 3.1415926535;
    static E: number = 2.71828;

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

console.log(MathHelper.PI);         // 3.1415926535
MathHelper.E = 2.718;
console.log(MathHelper.add(5, 7)); // 12
// const helper = new MathHelper();
// console.log(helper.PI); // Lỗi: PI là static
```

Giải thích:

- `PI` là hằng số tĩnh (`static readonly`), không thể thay đổi.
- `E` là biến tĩnh, có thể thay đổi.
- `add` là phương thức tĩnh, gọi trực tiếp qua `MathHelper` .

Lưu ý thực tiễn:

- Sử dụng `static` cho các tiện ích chung (utilities) hoặc dữ liệu toàn cục (global constants).

6. Inheritance (Kế thừa)

Kế thừa cho phép class con (`extends`) tái sử dụng thuộc tính và phương thức của class cha. Class con có thể ghi đè (override) phương thức cha.

Ví dụ:

```
class Shape {
    constructor(public color: string) {}

    draw(): void {
        console.log(`Drawing a shape with color ${this.color}`);
    }
}

class Circle extends Shape {
    constructor(color: string, public radius: number) {
        super(color); // Gọi constructor cha
    }

    draw(): void { // Override
        super.draw(); // Gọi phương thức cha
        console.log(`Drawing a circle with radius ${this.radius}`);
    }

    calculateArea(): number {
        return Math.PI * this.radius * this.radius;
    }
```

```
}

const circle = new Circle("red", 5);
circle.draw();
// In:
// Drawing a shape with color red
// Drawing a circle with radius 5
console.log("Area:", circle.calculateArea());
```

Giải thích:

- `Circle` kế thừa `Shape`, tái sử dụng `color` và phương thức `draw`.
- `super(color)` gọi constructor của `Shape`.
- `draw` được ghi đè để thêm thông tin về `radius`.

Lưu ý thực tiễn:

- Sử dụng `super` để truy cập phương thức hoặc constructor của class cha.
- Kế thừa giúp giảm trùng lặp code, nhưng tránh lạm dụng (ưu tiên composition khi phù hợp).

7. Abstract Classes và Abstract Methods

- **Abstract Class**: Không thể tạo instance trực tiếp, dùng làm class cha để các class con kế thừa.
- **Abstract Method**: Chỉ khai báo, không có triển khai. Class con phải triển khai.

Ví dụ:

```
abstract class Logger {
    abstract log(message: string): void;

    printDate(): void {
        console.log(new Date().toLocaleTimeString());
    }
}
```

```
class ConsoleLogger extends Logger {
    log(message: string): void {
        console.log(`[CONSOLE] ${message}`);
    }
}

const logger = new ConsoleLogger();
logger.printDate();
logger.log("User action performed.");
```

Giải thích:

- `Logger` là abstract class, không thể tạo instance (`new Logger()`).
- `ConsoleLogger` triển khai `log` để in ra console.
- `printDate` là phương thức chung, được kế thừa.

Lưu ý thực tiễn:

- Sử dụng abstract class khi muốn định nghĩa một giao diện chung (interface) nhưng vẫn cung cấp một số triển khai.

8. Shorthand Constructor Initialization

TypeScript cho phép khai báo và khởi tạo thuộc tính ngay trong constructor bằng access modifier.

Ví dụ:

```
class Person {
    constructor(
        public readonly id: number,
        public name: string,
        private age: number
    ) {}

    displayInfo(): void {
```

```
        console.log(`ID: ${this.id}, Name: ${this.name}, Age: ${this.age}`);
    }
}

const person = new Person(1, "Alice", 30);
person.displayInfo(); // In: ID: 1, Name: Alice, Age: 30
console.log(person.name); // OK: Alice
// console.log(person.age); // Lỗi: age là private
```

Giải thích:

- `id`, `name`, `age` được khai báo trực tiếp trong constructor, TypeScript tự động gán chúng vào instance.
- `Person` không cần viết `this.id = id`.

Lưu ý thực tiễn:

- Sử dụng shorthand để giảm code, nhưng tránh lạm dụng khi constructor có logic phức tạp.

Ví dụ tổng hợp

```
abstract class MediaItem {
    constructor(
        public readonly title: string,
        protected duration: number
    ) {}

    abstract play(): void;

    getDetails(): string {
        return `Title: ${this.title}, Duration: ${this.duration}s`;
    }

    static getLibraryName(): string {
        return "My Awesome Media Library";
    }
}
```

```
class Song extends MediaItem {
    constructor(
        title: string,
        duration: number,
        public artist: string,
        private album?: string
    ) {
        super(title, duration);
    }

    play(): void {
        console.log(`Playing song: ${this.title} by ${this.artist} (${this.d
        if (this.album) console.log(`Album: ${this.album}`);
    }
}

class Movie extends MediaItem {
    private _rating: number = 0;

    constructor(
        title: string,
        duration: number,
        public director: string
    ) {
        super(title, duration);
    }

    play(): void {
        console.log(`Playing movie: ${this.title}, Director: ${this.director
    }

    get rating(): number {
        return this._rating;    }

    set rating(newRating: number) {
        if (newRating ⩾ 0 && newRating ⩽ 10) {
            this._rating = newRating;
        } else {
            console.warn("Rating must be between 0 and 10");
        }
    }
}
```

```
console.log("Library Name:", MediaItem.getDetails());getLibraryName());

const song = new Song("Bohemian Rhapsody", 354, "Queen", "A Night at the Ope
const movie = new Movie("Inception", "8880, "Christopher Nolan");

song.play();
console.log(song.getDetails());
movie.play();
movie.rating = 9.5;
console.log("Movie Rating:", movie.rating);
```

Giải thích:

- `MediaItem` là abstract class với thuộc tính `title` (`readonly`) và `duration` (`protected).
- `Song` và `Movie` kế thừa, triển khai `play()` theo cách riêng.
- `static getLibraryName()` cung cấp tên thư viện chung.
- **Ứng dụng thực tế:**
  - Trong React, bạn có thể sử dụng `class` để định nghĩa các service (như `MediaService` để quản lý media) hoặc các model dữ liệu.
  - Trong Node.js, `class` thường dùng để tạo các controller hoặc repository.

Danh sách bài tập

1. Trắc nghiệm: Access Modifiers

Mô tả: Chọn access modifier phù hợp.

Câu hỏi: Trong TypeScript, access modifier nào cho phép truy cập từ trong class và class con, nhưng không từ instance?

- A. public
- B. private

- C. protected
- D. readonly

Đáp án: C

2. Code: Tạo class `Vehicle`

Mô tả: Tạo class `Vehicle` với:

- `make` (`string`, `public`)
- `model` (`string`, `public`)
- `year` (`number`, `public`)
- `_vin` (`string`, `private`)
- Phương thức `displayInfo()` để in thông tin.
- Constructor kiểm tra `_vin` (17 ký tự).

Giải pháp:

```
class Vehicle {
    private _vin: string;

    constructor(
        public make: string,
        public model: string,
        public year: number,
        vin: string
    ) {
        if (vin.length !== 17) {
            throw new Error("VIN phải dài 17 ký tự.");
        }
        this._vin = vin;
    }

    displayInfo(): void {
        console.log(`Vehicle: ${this.year} ${this.make} ${this.make} ${model
    }
```

```
    getVIN(): string {
        return this._vin;
    }
}

try {
    const car = new Vehicle("Honda", "CRV", 2021, "1234567890ABCDEFGH");
    car.displayInfo(); // In: Vehicle: 2021 CRV
    console.log("VIN:", car.getVIN()); // In: VIN:1234567890ABCDEFGH
} catch (error: any) {
    console.error("Error:", error.message);
}
```

3. Code: Kế thừa class `Vehicle`

Mô tả: Tạo class `Car` kế thừa `Vehicle`, thêm:

- `numberOfDoors` (`number`, `public`)
- Override `displayInfo()` để bao gồm `numberOfDoors`.

Giải pháp:

```
class Car extends Vehicle {
    constructor(
        make: string,
        model: string,
        year: number,
        vin: string,
        public numberOfDoors: number
    ) {
        super(make, model, year, vin);
    }

    override displayInfo(): void
 {
        super.displayInfo();
        console.log("Number of Doors:", numberOfDoors);
        console.log(`Number of doors: ${this.numberOfDoors}`);
```

```
    }
}

try {
    const sedan = new Car("Toyota", "Camry", 2023, "ABC123DEF456GHI789", 4);
    sedan.displayInfo();
    // In:
    // Vehicle: 2023
    // Number of Doors: 4
    console.log("VIN:", sedan.getVIN());
} catch (error: any) {
    console.error("Error:", error.message);
}
```

4. Code: Getters và Setters cho `_vin`

Mô tả: Thay `getVIN()` bằng getter/setter cho `vin`, kiểm tra độ dài 17 ký tự.

Giải pháp:

```
class Vehicle {
    private _vin: string;

    constructor(
        public make: string,
        public model: string,
        public year: number,
        initialVin: string
    ) {
        this.vin = initialVin; // Dùng setter
    }

    displayInfo(): void {
        console.log(`Vehicle:${this.year} ${this.year} ${this.make} ${this.m
    }

    get vin(): string
 {
        console.log("Accessing VIN getter");
```

```
        return this._vin;
    }

    set vin(newVin: string
    ) {
        console.log("Setting VIN to:", newVin);
        if (newVin.length === 17) {
            this._vin = newVin;
        } else {
            console.error("Invalid VIN: Must be 17 characters.");
        }
    }
}

try {
    const truck = new Vehicle("Ford", "F-150", 2022, "TRUCK456789123000");
    console.log("VIN:", truck.vin); // Gọi getter
    truck.vin = "NEWVIN12345678900"; // Gọi setter
    console.log("New VIN:", truck.vin);
    truck.vin = "SHORT"; // In lỗi
    console.log("VIN after invalid:", truck.vin); // Vẫn là giá trị cũ
}
catch (error: any) {
    console.error("Error:", error.message);
}
```

5. Code: Abstract Class `DataProcessor`

Mô tả:

- Tạo `abstract class DataProcessor<TInput, TOutput>` với phương thức abstract `process`.
- Thêm `logStart()` và `logEnd()`.
- Tạo 2 class con:
  - `StringToNumberProcessor`: Chuyển chuỗi sang số.
  - `ArraySumProcessor`: Tính tổng mảng số.

Giải pháp:

```typescript
abstract class DataProcessor<TInput, TOutput> {
    abstract process(data: TInput): TOutput;

    logStart(): void {
        console.log("Data processing started.");
    }

    logEnd(): void {
        console.log("Data processing finished.");
    }

    execute(data: TInput): TOutput {
        this.logStart();
        const result = this.process(data);
        this.logEnd();
        return result;
    }
}

class StringToNumberProcessor extends DataProcessor<string, number> {
    process(data: string): number {
        const num = parseInt(data, 10);
        if (isNaN(num)) {
            throw new Error(`"${data}" is not a valid number.`);
        }
        return num;
    }
}

class ArraySumProcessor extends DataProcessor<number[], number> {
    process(data: number[]): number {
        return data.reduce((sum, num) => sum + num, 0);
    }
}

try {
    const parser = new StringToNumberProcessor();
    console.log("Result:", parser.execute("123")); // In: Data processing st
    // parser.execute("abc"); // Throw error

    const summer = new ArraySumProcessor();
    console.log("Sum:", summer.execute([1, 2, 3])); // In: Sum: 6
```

```
} catch (error: any) {
    console.error("Error:", error.message);
}
```

Ứng dụng thực tế

- **Trong React**: Sử dụng `class` để tạo các **service classes** (ví dụ: `AuthService` để quản lý đăng nhập/đăng xuất).
- **Node.js**: Dùng `class` để tạo các **controller** hoặc **repository** trong mô hình MVC.
- **Best Practices**:
  - Luôn khai báo kiểu dữ liệu cho thuộc tính và phương thức.
  - Sử dụng `private` và `protected` để bảo vệ dữ liệu.
  - Ưu tiên `readonly` cho các thuộc tính không đổi.
  - Tránh lạm dụng kế thừa, sử dụng **composition** khi cần thiết.

Kết luận

Chapter này đã trang bị cho bạn kiến thức về OOP trong TypeScript. Hãy thực hành các bài tập và áp dụng vào dự án thực tế để nắm vững các khái niệm!