Chapter 10: Modules và Namespaces - Tổ Chức Code Hiệu Quả

Mô tả tổng quát

Khi dự án TypeScript phát triển lớn hơn, việc tổ chức code thành các phần nhỏ, dễ quản lý và tái sử dụng là yếu tố sống còn. Trong chapter này, chúng ta sẽ khám phá hai cơ chế chính để tổ chức code trong TypeScript: **Modules** (theo chuẩn ES Modules) và **Namespaces**. Bạn sẽ học cách sử dụng `export` và `import` để chia sẻ code giữa các file, cũng như cách sử dụng Namespaces để nhóm code liên quan và tránh xung đột tên trong phạm vi toàn cục (global scope). Chúng ta cũng sẽ so sánh khi nào nên dùng Modules và khi nào nên dùng Namespaces.

Mục tiêu học tập:

- Hiểu được vấn đề của global scope pollution và tại sao cần tổ chức code.
- Thành thạo cú pháp `export` và `import` trong ES Modules.
- Biết cách sử dụng Namespaces để nhóm code logic.
- Phân biệt được khi nào nên sử dụng Modules hay Namespaces trong dự án thực tế.

Tiêu đề Chapter

Tổ Chức Code Hiệu Quả với Modules và Namespaces

Lý thuyết chính

1. Tại sao cần tổ chức code? Vấn đề của Global Scope Pollution

Trong JavaScript truyền thống (trước ES6), các biến, hàm, hoặc class thường được khai báo ở **global scope** (phạm vi toàn cục). Điều này dẫn đến các vấn đề nghiêm trọng:

- **Xung đột tên (Naming Collisions)**: Nếu hai phần code (hoặc hai thư viện) định nghĩa biến hoặc hàm cùng tên trong global scope, chúng sẽ ghi đè lên nhau, gây lỗi khó lường. Ví dụ:

```
// File 1
var counter = 1;

// File 2
var counter = "Hello"; // Ghi đè lên counter của File 1
```

- **Khó quản lý dependency**: Không có cách rõ ràng để biết một đoạn code phụ thuộc vào đoạn code nào khác.

- **Khó bảo trì và mở rộng**: Khi tất cả code nằm trong một file hoặc global scope, việc tìm kiếm, chỉnh sửa, hoặc mở rộng code trở nên phức tạp.

Modules và **Namespaces** giúp giải quyết các vấn đề này bằng cách:

- Tạo ra các **phạm vi riêng biệt** (isolated scopes) cho code.
- Cho phép chia nhỏ code thành các đơn vị độc lập, dễ quản lý và tái sử dụng.

2. Modules trong TypeScript (Chuẩn ES Modules)

TypeScript hỗ trợ đầy đủ **ES Modules** (chuẩn modules của JavaScript từ ES6 trở đi). Một file `.ts` được coi là một module nếu nó chứa ít nhất một câu lệnh `import` hoặc `export` ở cấp độ cao nhất. Các biến, hàm, class, hoặc interface trong module chỉ có phạm vi cục bộ trong file đó, trừ khi được `export` tường minh.

2.1. `export` (Xuất dữ liệu từ Module)

Có ba cách chính để export dữ liệu từ một module:

- **Named Export (Xuất theo tên)**: Cho phép export nhiều thành phần từ một file. Các thành phần này phải được import chính xác bằng tên.

```ts
// File: mathUtils.ts
export const PI = 3.14159;

export function add(a: number, b: number): number {
    return a + b;
}

export class Calculator {
    multiply(a: number, b: number) {
        return a * b;
    }
}
```

Bạn cũng có thể export danh sách ở cuối file:

```ts
// File: stringUtils.ts
function toUpperCase(str: string): string {
    return str.toUpperCase();
}
function toLowerCase(str: string): string {
    return str.toLowerCase();
}
export { toUpperCase, toLowerCase };
```

- **Default Export (Xuất mặc định)**: Mỗi module chỉ có thể có **một** default export. Đây thường là thành phần chính của module, và khi import, bạn có thể đặt tên tùy ý.

```ts
// File: logger.ts
export default class Logger {
    log(message: string) {
        console.log(message);
    }
}
```

- **Re-exporting (Xuất lại)**: Cho phép import từ một module khác và export lại, hoặc export trực tiếp từ module khác.

```
// File: mainExporter.ts
export { add as sumNumbers } from './mathUtils'; // Đổi tên add thành su
export * from './stringUtils'; // Export tất cả named exports từ stringU
export { default as MyLogger } from './logger'; // Export default Logger
```

2.2. `import` (Nhập dữ liệu vào Module)

Tương ứng với các kiểu export, TypeScript hỗ trợ các cách import sau:

- **Named Import (Nhập theo tên)**: Import các thành phần được named export.

```
// File: app.ts
import { PI, add, Calculator } from './mathUtils';
import { toUpperCase } from './stringUtils';

console.log(PI); // 3.14159
const calc = new Calculator();
console.log(calc.multiply(2, 3)); // 6
```

- **Default Import (Nhập mặc định)**: Import default export với tên tùy ý.

```
// File: app.ts
import MyCustomLogger from './logger';
const logger = new MyCustomLogger();
logger.log("App started");
```

- **Namespace Import (Nhập tất cả dưới dạng namespace)**: Import tất cả named exports vào một đối tượng duy nhất.

```
// File: app.ts
import * as MathHelpers from './mathUtils';
console.log(MathHelpers.PI); // 3.14159
console.log(MathHelpers.add(1, 2)); // 3
```

- **Import for side effects only (Nhập chỉ để chạy code)**: Dùng để chạy code trong module (ví dụ: thiết lập polyfills).

```
import './polyfills'; // Chạy code trong polyfills.ts mà không import gì
```

2.3. Dynamic Import (Nhập động)

TypeScript hỗ trợ **dynamic import** để tải module theo cách bất đồng bộ, rất hữu ích cho **code splitting** và **lazy loading** trong các ứng dụng lớn.

```
// File: app.ts
async function loadMathUtils() {
    const { add, PI } = await import('./mathUtils');
    console.log(`PI: ${PI}, Sum: ${add(5, 10)}`);
}
loadMathUtils();
```

Khi nào dùng dynamic import? Khi bạn muốn tải một module chỉ khi cần thiết, giúp giảm thời gian tải ban đầu của ứng dụng.

2.4. Module Resolution Strategies (Chiến lược phân giải Module)

TypeScript cần biết cách tìm file module khi bạn sử dụng `import`. Các chiến lược phân giải module bao gồm:

- `node`: Mô phỏng cách Node.js tìm module, tìm trong thư mục hiện tại và `node_modules`. Đây là chiến lược mặc định cho `module: commonjs`, `es2015`, hoặc cao hơn.

- `node16` : Tương tự `node` nhưng tuân thủ chuẩn module của Node.js từ phiên bản 16, hỗ trợ tốt cho ESM trong Node.js.
- `bundler` : Dành cho các công cụ như Webpack, Vite, hoặc Rollup, nơi module được đóng gói bởi bundler.
- `classic` : Chiến lược cũ, ít dùng, tìm module trong thư mục hiện tại và các thư mục cha.

Cấu hình trong `tsconfig.json` :

```
{
  "compilerOptions": {
    "module": "esnext",
    "moduleResolution": "bundler" // Hoặc "node", "node16"
  }
}
```

Lưu ý: `moduleResolution` nên được đặt thành `bundler` hoặc `node16` cho các dự án hiện đại sử dụng ESM. Nếu làm việc với Node.js, hãy kiểm tra phiên bản Node.js để chọn `node` hoặc `node16` .

2.5. Cấu hình `tsconfig.json` cho Modules

Một số tùy chọn quan trọng trong `tsconfig.json` khi làm việc với Modules:

- `module` : Xác định hệ thống module (`commonjs` , `esnext` , `es2015` , `amd` , `system` , v.v.).
- `baseUrl` : Đường dẫn cơ sở để phân giải các module không tương đối (non-relative imports).
- `paths` : Ánh xạ các alias cho đường dẫn module (rất hữu ích trong dự án lớn).

Ví dụ:

```
{
  "compilerOptions": {
    "module": "esnext",
    "moduleResolution": "bundler",
```

```
    "baseUrl": "src",
    "paths": {
      "@utils/*": ["utils/*"],
      "@services/*": ["services/*"]
    }
  }
}
```

Với cấu hình trên, bạn có thể viết:

```
import { formatDate } from '@utils/formatter'; // Thay vì './utils/formatter
```

3. Namespaces trong TypeScript

Namespaces (trước đây gọi là internal modules) là cách TypeScript tổ chức code trong phạm vi toàn cục mà không cần sử dụng hệ thống file-based modules. Chúng giúp nhóm các thành phần liên quan (hàm, class, interface) vào một đối tượng lớn, tránh xung đột tên trong global scope.

Khi nào dùng Namespaces?

- Trong các dự án nhỏ, không cần thiết lập hệ thống module phức tạp.
- Khi làm việc với các file JavaScript cũ hoặc nhúng trực tiếp vào HTML qua thẻ `<script>`.
- Khi cần nhóm code logic liên quan mà không muốn chia thành nhiều file.

3.1. Khai báo Namespace

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
```

```
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

- Các thành phần bên trong namespace cần được `export` để có thể truy cập từ bên ngoài.
- Namespace tạo ra một đối tượng toàn cục (như `Validation`) chứa tất cả các thành phần được export.

3.2. Sử dụng Namespace

```
let strings = ["Hello", "98052", "101"];
let validators: { [s: string]: Validation.StringValidator } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches"
    }
});
```

3.3. Nested Namespaces (Namespace lồng nhau)

```
namespace Forms {
    export namespace Controls {
        export class Button {
            constructor() {
                console.log("Button created");
            }
        }
        export class TextBox {
            constructor() {
                console.log("TextBox created");
            }
        }
    }
}

let button = new Forms.Controls.Button(); // Button created
```

3.4. Namespace Aliases (Bí danh Namespace)

Nếu namespace lồng sâu, bạn có thể tạo bí danh để truy cập dễ dàng hơn:

```
import FCV = Forms.Controls;
let button = new FCV.Button();
```

3.5. Namespaces trong nhiều file

Nếu namespace được chia thành nhiều file, bạn cần sử dụng `/// <reference path="…">` và biên dịch với tùy chọn `outFile` trong `tsconfig.json` để gộp các file thành một file JavaScript duy nhất.

```
// File: ValidatorInterfaces.ts
namespace Validation {
    export interface StringValidator {
```

```
        isAcceptable(s: string): boolean;
    }
}

// File: LetterValidator.ts
/// <reference path="ValidatorInterfaces.ts" />
namespace Validation {
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return /^[A-Za-z]+$/.test(s);
        }
    }
}
```

Cấu hình `tsconfig.json`:

```
{
  "compilerOptions": {
    "module": "system", // Hoặc "amd"
    "outFile": "app.js"
  }
}
```

Biên dịch: `tsc --outFile app.js ValidatorInterfaces.ts LetterValidator.ts`

Lưu ý: Cách này ít được sử dụng trong các dự án hiện đại vì các công cụ như Webpack hoặc Vite đã thay thế việc gộp file thủ công.

4. So sánh Modules và Namespaces

Tiêu chí	Modules (ES Modules)	Namespaces
Cách hoạt động	Mỗi file `.ts` là một module nếu có `import` / `export`.	Nhóm code trong một phạm vi cục bộ, thường trong global scope.

Tiêu chí	Modules (ES Modules)	Namespaces
Phạm vi	Phạm vi cục bộ trong file, cô lập hoàn toàn.	Phạm vi toàn cục, trừ khi được export.
Quản lý dependency	Rõ ràng qua `import` / `export`.	Dùng `/// <reference>` hoặc gộp file thủ công.
Hỗ trợ công cụ	Hỗ trợ tốt bởi Webpack, Rollup, Vite, Node.js.	Phù hợp cho script nhúng HTML hoặc dự án cũ.
Khả năng tái sử dụng	Cao, dễ chia sẻ giữa các dự án.	Thấp, chủ yếu dùng trong một file hoặc dự án nhỏ.
Khi nào dùng?	Dự án hiện đại, lớn, cần code splitting, lazy loading.	Dự án nhỏ, nhúng script vào HTML, hoặc chuyển đổi code cũ.

Khuyến nghị:

- **Ưu tiên Modules** cho hầu hết các dự án TypeScript hiện đại, đặc biệt khi làm việc với Node.js, React, hoặc các bundler.
- Chỉ dùng **Namespaces** cho các trường hợp đặc biệt, như làm việc với code JavaScript cũ hoặc khi cần một file output duy nhất mà không dùng bundler.
- Tránh trộn lẫn Modules và Namespaces trong cùng dự án để tránh nhầm lẫn.

5. Code ví dụ tổng hợp

5.1. Ví dụ về Modules

```
// File: services/userService.ts
export interface User {
    id: number;
    name: string;
    email: string;
}
```

```ts
export function getUserById(id: number): User | undefined {
    const users: User[] = [
        { id: 1, name: "Alice", email: "alice@example.com" },
        { id: 2, name: "Bob", email: "bob@example.com" }
    ];
    return users.find(user => user.id === id);
}

export default class UserAPI {
    static async fetchAllUsers(): Promise<User[]> {
        console.log("Fetching all users...");
        return [
            { id: 1, name: "Alice", email: "alice@example.com" },
            { id: 2, name: "Bob", email: "bob@example.com" }
        ];
    }
}

// File: utils/formatter.ts
export function formatDate(date: Date): string {
    return date.toLocaleDateString("vi-VN");
}

export const DEFAULT_CURRENCY = "VND";

// File: app.ts
import UserApiService, { getUserById as findUser, type User as UserType } fr
import { formatDate, DEFAULT_CURRENCY } from './utils/formatter';
import * as _ from 'lodash'; // Giả sử đã cài @types/lodash

function displayUser(userId: number) {
    const user = findUser(userId);
    if (user) {
        console.log(`User Found: ${user.name} (${user.email}), Currency: ${D
    } else {
        console.log(`User with ID ${userId} not found.`);
    }
}

displayUser(1);
displayUser(3);
```

```
UserApiService.fetchAllUsers().then(users => {
    console.log("All Users:", users.map(u => u.name).join(", "));
});

console.log("Today's date:", formatDate(new Date()));

const numbers = [1, 5, 2, 8, 3];
console.log("Sorted numbers (lodash):", _.sortBy(numbers));
```

5.2. Ví dụ về Namespaces

```
// File: app.ts
namespace MyCompany.Utilities {
    export function isEmpty(str: string | null | undefined): boolean {
        return !str || str.trim().length === 0;
    }
}

namespace MyCompany.Models {
    export interface Item {
        id: string;
        value: any;
    }
}

function processItem(item: MyCompany.Models.Item): void {
    if (MyCompany.Utilities.isEmpty(item.id)) {
        console.error("Item ID cannot be empty.");
        return;
    }
    console.log(`Processing item ${item.id} with value:`, item.value);
}

let item1: MyCompany.Models.Item = { id: "item-001", value: { data: "sample"
let item2: MyCompany.Models.Item = { id: " ", value: 123 };

processItem(item1);
processItem(item2);
```

Danh sách bài tập

Bài 1: Trắc nghiệm - Hiểu về Default Export

Mô tả: Chọn cú pháp đúng để export một giá trị mặc định từ một module.

Câu hỏi: Trong TypeScript, cách nào sau đây là đúng để export một giá trị mặc định?

- A. `export default const myValue = 10;`
- B. `export { myValue as default };` (với `const myValue = 10;` khai báo trước)
- C. `default export const myValue = 10;`
- D. `export default myValue;` (với `const myValue = 10;` khai báo trước)

Đáp án đúng: D.
Giải thích:

- Cách đúng để export mặc định là `export default expression;` hoặc `export default variable;` sau khi biến đã được khai báo. Ví dụ:

```
const myValue = 10;
export default myValue;
```

- **A** sai vì `export default` không thể kết hợp trực tiếp với khai báo biến (`const` , `let` , `var`).
- **B** đúng nhưng không phải cách phổ biến. Nó hoạt động nếu `myValue` đã được khai báo.
- **C** sai vì `default export` không phải cú pháp hợp lệ trong TypeScript/JavaScript.

Bài 2: Code - Tạo module `logger.ts`

Mô tả:

1. Tạo file `logger.ts` với hai hàm:

- `logInfo(message: string)`: In ra `[INFO] message`.
- `logError(message: string)`: In ra `[ERROR] message`.

2. Tạo file `main.ts` để import và sử dụng hai hàm này.

Giải pháp mẫu:

`logger.ts`:

```
export function logInfo(message: string): void {
    console.log(`[INFO] ${message}`);
}

export function logError(message: string): void {
    console.error(`[ERROR] ${message}`);
}
```

`main.ts`:

```
import { logInfo, logError } from './logger';

logInfo("Application has started successfully.");
logError("A minor issue occurred, but was handled.");
```

Hướng dẫn thực hành:

1. Tạo hai file `logger.ts` và `main.ts` trong cùng thư mục.
2. Biên dịch: `tsc logger.ts main.ts`.
3. Chạy: `node main.js` (nếu dùng `module: commonjs`) hoặc dùng `ts-node main.ts`.

Bài 3: Code - Tạo module `constants.ts` với Default và Named Export

Mô tả:

1. Tạo file `constants.ts`:
    - Export hằng số `MAX_RETRIES = 5` (named export).
    - Export `enum UserRole { ADMIN, EDITOR, GUEST }` (default export).
2. Tạo file `appConfig.ts` để import và sử dụng `MAX_RETRIES` và `UserRole`.

Giải pháp mẫu:

`constants.ts`:

```
export const MAX_RETRIES: number = 5;

enum UserRole {
    ADMIN = "ADMIN",
    EDITOR = "EDITOR",
    GUEST = "GUEST"
}
export default UserRole;
```

`appConfig.ts`:

```
import DefaultUserRole, { MAX_RETRIES } from './constants';

console.log("Maximum Retries Allowed:", MAX_RETRIES);
console.log("Default Admin Role:", DefaultUserRole.ADMIN);
console.log("Guest Role Value:", DefaultUserRole.GUEST);

let currentUserRole: DefaultUserRole = DefaultUserRole.EDITOR;
console.log("Current User Role:", currentUserRole);
```

Hướng dẫn thực hành:

1. Tạo hai file `constants.ts` và `appConfig.ts`.
2. Biên dịch và chạy tương tự Bài 2.

3. Thử thay đổi giá trị của `currentUserRole` để kiểm tra các giá trị khác của `UserRole`.

Bài 4: Code - Tạo namespace `Utils.StringHelper`

Mô tả: Tạo namespace `Utils.StringHelper` với hai hàm:

- `reverse(s: string)`: Đảo ngược chuỗi.
- `countWords(s: string)`: Đếm số từ trong chuỗi (từ được phân cách bởi khoảng trắng). Sử dụng namespace trong cùng file để kiểm tra.

Giải pháp mẫu:

```
namespace Utils {
    export namespace StringHelper {
        export function reverse(s: string): string {
            return s.split("").reverse().join("");
        }

        export function countWords(s: string): number {
            if (!s || s.trim().length === 0) {
                return 0;
            }
            return s.trim().split(/\s+/).length;
        }
    }
}

let originalString = "Hello TypeScript World";
let reversed = Utils.StringHelper.reverse(originalString);
let wordCount = Utils.StringHelper.countWords(originalString);

console.log(`Original: "${originalString}"`);
console.log(`Reversed: "${reversed}"`);
console.log(`Word Count: ${wordCount}`);

let emptyStr = "   ";
console.log(`Word Count for empty string: ${Utils.StringHelper.countWords(em
```

Hướng dẫn thực hành:

1. Tạo file `stringHelper.ts` chứa code trên.
2. Biên dịch: `tsc stringHelper.ts --module system --outFile app.js`.
3. Chạy: `node app.js`.
4. Thử thêm các chuỗi khác nhau để kiểm tra hàm `reverse` và `countWords`.

Bài 5: Code - Chuyển đổi Namespace thành Module

Mô tả: Chuyển namespace `Utils.StringHelper` từ Bài 4 thành module trong file `stringUtilsModule.ts`. Tạo file `mainApp.ts` để import và sử dụng các hàm.

Giải pháp mẫu:

`stringUtilsModule.ts`:

```
export function reverse(s: string): string {
    return s.split("").reverse().join("");
}

export function countWords(s: string): number {
    if (!s || s.trim().length === 0) {
        return 0;
    }
    return s.trim().split(/\s+/).length;
}
```

`mainApp.ts`:

```
import { reverse as reverseString, countWords as countStringWords } from './

let textToProcess = "Learning modules in TypeScript";

let reversedText = reverseString(textToProcess);
let wordsInText = countStringWords(textToProcess);
```

```
console.log(`Original Text: "${textToProcess}"`);
console.log(`Reversed Text: "${reversedText}"`);
console.log(`Number of Words: ${wordsInText}`);
```

Hướng dẫn thực hành:

1. Tạo hai file `stringUtilsModule.ts` và `mainApp.ts`.
2. Biên dịch và chạy như các bài trước.
3. So sánh cách sử dụng module với namespace ở Bài 4. Module có ưu điểm gì trong trường hợp này?

Bài 6: Thực hành nâng cao - Dự án nhỏ với Modules

Mô tả: Tạo một dự án nhỏ với cấu trúc thư mục như sau:

```
src/
  ├── models/
  │   └── user.ts
  ├── utils/
  │   └── formatter.ts
  ├── services/
  │   └── userService.ts
  ├── app.ts
  └── tsconfig.json
```

- **Yêu cầu**:
  1. Trong `user.ts`, định nghĩa interface `User` và export.
  2. Trong `formatter.ts`, tạo hàm `formatUser(user: User)` để định dạng thông tin user thành chuỗi.
  3. Trong `userService.ts`, tạo hàm `getUserById(id: number)` và default export class `UserAPI`.
  4. Trong `app.ts`, import và sử dụng các module trên để hiển thị thông tin user.

5. Cấu hình `tsconfig.json` với `baseUrl` và `paths` để dùng alias `@models`, `@utils`, `@services`.

Giải pháp mẫu:

`src/models/user.ts`:

```ts
export interface User {
    id: number;
    name: string;
    email: string;
}
```

`src/utils/formatter.ts`:

```ts
import { User } from '@models/user';

export function formatUser(user: User): string {
    return `${user.name} <${user.email}> (ID: ${user.id})`;
}
```

`src/services/userService.ts`:

```ts
import { User } from '@models/user';

export function getUserById(id: number): User | undefined {
    const users: User[] = [
        { id: 1, name: "Alice", email: "alice@example.com" },
        { id: 2, name: "Bob", email: "bob@example.com" }
    ];
    return users.find(user => user.id === id);
}

export default class UserAPI {
    static async fetchAllUsers(): Promise<User[]> {
```

```
        return [
            { id: 1, name: "Alice", email: "alice@example.com" },
            { id: 2, name: "Bob", email: "bob@example.com" }
        ];
    }
}
```

`src/app.ts`:

```
import { getUserById } from '@services/userService';
import UserAPI from '@services/userService';
import { formatUser } from '@utils/formatter';

async function main() {
    const user = getUserById(1);
    if (user) {
        console.log("User:", formatUser(user));
    }

    const users = await UserAPI.fetchAllUsers();
    console.log("All Users:", users.map(formatUser).join(", "));
}

main();
```

`src/tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
    "moduleResolution": "bundler",
    "baseUrl": "src",
    "paths": {
      "@models/*": ["models/*"],
      "@utils/*": ["utils/*"],
      "@services/*": ["services/*"]
```

```
    },
    "outDir": "dist",
    "strict": true
  }
}
```

Hướng dẫn thực hành:

1. Tạo cấu trúc thư mục và các file như trên.
2. Chạy `tsc` để biên dịch và kiểm tra file trong thư mục `dist`.
3. Chạy `node dist/app.js` để xem kết quả.
4. Thử thêm một user mới trong `userService.ts` và kiểm tra lại.

Kết luận

- **Modules** là lựa chọn tiêu chuẩn cho các dự án TypeScript hiện đại, với cú pháp `import` / `export` rõ ràng và hỗ trợ tốt từ các công cụ như Vite, Webpack.
- **Namespaces** phù hợp cho các dự án nhỏ hoặc khi làm việc với code JavaScript cũ, nhưng nên hạn chế sử dụng trong các dự án mới.
- Thực hành thường xuyên với các bài tập trên để nắm vững cách tổ chức code hiệu quả trong TypeScript.