Table of Contents

# Bab 9: Modules dan Namespaces di TypeScript

## Penjelasan Materi

Modules dan Namespaces adalah fitur TypeScript yang memungkinkan kita untuk mengorganisir kode menjadi unit-unit yang terpisah dan dapat digunakan kembali. Modules memungkinkan kita untuk membagi kode ke dalam file-file terpisah dan mengontrol akses ke kode tersebut, sementara Namespaces menyediakan cara untuk mengelompokkan kode terkait dalam satu namespace untuk menghindari konflik nama.

## Analogi yang Mudah Dipahami

Bayangkan Modules seperti ruangan dalam gedung: - Setiap module seperti ruangan terpisah - Export seperti pintu yang bisa diakses dari luar - Import seperti mengambil barang dari ruangan lain - Default export seperti pintu utama - Named exports seperti pintu-pintu khusus

Bayangkan Namespaces seperti departemen dalam perusahaan: - Namespace seperti departemen yang berbeda - Anggota namespace seperti karyawan departemen - Nested namespace seperti sub-departemen - Export namespace seperti kolaborasi antar departemen

## Point Penting

1. **Module Basics**
   * Export/Import syntax
   * Default exports
   * Named exports
   * Module resolution
2. **Namespace Organization**
   * Namespace declaration
   * Nested namespaces
   * Namespace exports
   * Reference tags
3. **Module Loading**
   * AMD/CommonJS/UMD
   * ES Modules
   * Dynamic imports
   * Module bundling
4. **Best Practices**
   * Module organization
   * Circular dependencies
   * Barrel exports
   * Path aliases
5. **Advanced Features**
   * Re-exports
   * Type-only imports
   * Import assertions
   * Module augmentation

## Contoh Kode dan Penjelasan

```typescript // 1. Basic Module Exports (math.ts) export const add = (a: number, b: number): number => a + b; export const subtract = (a: number, b: number): number => a - b;

export default class Calculator { add(a: number, b: number): number { return add(a, b); }

subtract(a: number, b: number): number {  
 return subtract(a, b);  
}

}

// 2. Module Imports (app.ts) import Calculator, { add, subtract } from ‘./math’;

const calc = new Calculator(); console.log(calc.add(5, 3)); // 8 console.log(subtract(10, 4)); // 6

// 3. Namespace Declaration namespace Validation { export interface StringValidator { isValid(s: string): boolean; }

export class EmailValidator implements StringValidator {  
 isValid(email: string): boolean {  
 const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;  
 return emailRegex.test(email);  
 }  
}  
  
export class PhoneValidator implements StringValidator {  
 isValid(phone: string): boolean {  
 const phoneRegex = /^\+?[\d\s-]{10,}$/;  
 return phoneRegex.test(phone);  
 }  
}

}

// 4. Namespace Usage let emailValidator = new Validation.EmailValidator(); let phoneValidator = new Validation.PhoneValidator();

console.log(emailValidator.isValid(“test@email.com”)); // true console.log(phoneValidator.isValid(“+1234567890”)); // true

// 5. Barrel Exports (index.ts) export \* from ‘./models/user’; export \* from ‘./models/product’; export \* from ‘./services/api’;

// 6. Type-only Imports import type { User } from ‘./models/user’;

// 7. Dynamic Imports async function loadModule() { const { default: Module } = await import(‘./dynamic-module’); return new Module(); }

// 8. Module with Types and Implementation // types.ts export interface User { id: number; name: string; email: string; }

export interface UserService { getUser(id: number): Promise; createUser(user: Omit<User, “id”>): Promise; }

// implementation.ts import { User, UserService } from ‘./types’;

export class UserServiceImpl implements UserService { async getUser(id: number): Promise { // Implementation return { id, name: “John”, email: “john@example.com” }; }

async createUser(user: Omit<User, "id">): Promise<User> {  
 // Implementation  
 return { id: 1, ...user };  
}

}

// 9. Module Augmentation // original.ts export interface User { id: number; name: string; }

// augmentation.ts import { User } from ‘./original’;

declare module ‘./original’ { interface User { email: string; } }

// 10. Practical Example: Feature Module // feature/types.ts export interface Feature { id: string; name: string; enabled: boolean; }

export interface FeatureToggle { isEnabled(featureId: string): boolean; enable(featureId: string): void; disable(featureId: string): void; }

// feature/implementation.ts import { Feature, FeatureToggle } from ‘./types’;

export class FeatureService implements FeatureToggle { private features: Map<string, Feature> = new Map();

constructor(features: Feature[]) {  
 features.forEach(feature => {  
 this.features.set(feature.id, feature);  
 });  
}  
  
isEnabled(featureId: string): boolean {  
 return this.features.get(featureId)?.enabled ?? false;  
}  
  
enable(featureId: string): void {  
 const feature = this.features.get(featureId);  
 if (feature) {  
 feature.enabled = true;  
 }  
}  
  
disable(featureId: string): void {  
 const feature = this.features.get(featureId);  
 if (feature) {  
 feature.enabled = false;  
 }  
}

}

// feature/index.ts export \* from ‘./types’; export \* from ‘./implementation’;

// app.ts import { Feature, FeatureService } from ‘./feature’;

const features: Feature[] = [ { id: “dark-mode”, name: “Dark Mode”, enabled: false }, { id: “beta-features”, name: “Beta Features”, enabled: false }];

const featureService = new FeatureService(features);

// Usage console.log(featureService.isEnabled(“dark-mode”)); // false featureService.enable(“dark-mode”); console.log(featureService.isEnabled(“dark-mode”)); // true ```

## Cara Kerja Modules dan Namespaces

1. **Module Resolution**:
   * Node-style resolution
   * Classic resolution
   * Path mapping
   * Base URLs
2. **Namespace Compilation**:
   * Internal module system
   * IIFE generation
   * Reference management
3. **Module Loading**:
   * Synchronous loading
   * Asynchronous loading
   * Bundling optimization

## Tips dan Trik

1. **Barrel Exports**

* // ✅ Gunakan barrel exports untuk API yang bersih  
  // features/index.ts  
  export \* from './user';  
  export \* from './product';  
  export \* from './order';  
    
  // app.ts  
  import { User, Product, Order } from './features';

1. **Path Aliases**

* // ✅ Gunakan path aliases di tsconfig.json  
  {  
   "compilerOptions": {  
   "baseUrl": ".",  
   "paths": {  
   "@/\*": ["src/\*"],  
   "@components/\*": ["src/components/\*"]  
   }  
   }  
  }  
    
  // Penggunaan  
  import { Button } from '@components/Button';

1. **Type-only Imports**

* // ✅ Gunakan type-only imports untuk type definitions  
  import type { User } from './types';  
  import { UserService } from './services';

## Kesalahan yang Sering Dilakukan Pemula

1. **Circular Dependencies**

* // ❌ Buruk: Circular dependency  
  // a.ts  
  import { b } from './b';  
  export const a = () => b();  
    
  // b.ts  
  import { a } from './a';  
  export const b = () => a();  
    
  // ✅ Baik: Refactor untuk menghindari circular dependency  
  // types.ts  
  export interface Common {  
   // shared types  
  }  
    
  // a.ts & b.ts mengimport dari types.ts

1. **Namespace vs Module**

* // ❌ Buruk: Mixing namespace dengan modules  
  namespace Utils {  
   export function helper() {}  
  }  
    
  // ✅ Baik: Gunakan modules  
  export function helper() {}

1. **Default Exports**

* // ❌ Buruk: Overuse of default exports  
  export default class Something {}  
    
  // ✅ Baik: Named exports untuk clarity  
  export class Something {}

### Solusi:

1. Gunakan modules untuk code organization
2. Hindari circular dependencies
3. Manfaatkan barrel exports
4. Gunakan path aliases
5. Pilih named exports daripada default exports