Table of Contents

# Bab 7: Decorators di TypeScript

## Penjelasan Materi

Decorators adalah fitur eksperimental di TypeScript yang memungkinkan kita untuk menambahkan metadata dan mengubah perilaku class, method, property, dan parameter. Decorators menggunakan sintaks `@expression`, di mana expression harus mengevaluasi ke function yang akan dipanggil saat runtime dengan informasi tentang dekorasi target. Decorators sangat berguna untuk metaprogramming, di mana kita bisa memodifikasi atau menambahkan perilaku ke kode secara deklaratif.

## Analogi yang Mudah Dipahami

Bayangkan Decorators seperti stiker atau label pada barang: - Class decorator seperti stiker “Fragile” pada kotak - Method decorator seperti label “Harus Dingin” pada makanan - Property decorator seperti tanda “Jangan Sentuh” pada barang - Parameter decorator seperti instruksi khusus pada bahan makanan

Atau seperti instruksi tambahan pada resep: - Class decorator seperti catatan umum tentang cara memasak - Method decorator seperti teknik khusus untuk langkah tertentu - Property decorator seperti spesifikasi bahan - Parameter decorator seperti cara mengolah bahan tertentu

## Point Penting

1. **Class Decorators**
   * Modifikasi konstruktor
   * Menambah property/method
   * Mengubah prototype
   * Inheritance handling
2. **Method Decorators**
   * Modifikasi method
   * Logging dan monitoring
   * Validasi input/output
   * Performance tracking
3. **Property Decorators**
   * Validasi property
   * Computed properties
   * Lazy loading
   * Access control
4. **Parameter Decorators**
   * Validasi parameter
   * Dependency injection
   * Parameter transformation
   * Default values
5. **Decorator Factories**
   * Customizable decorators
   * Parameter passing
   * Composition
   * Reusability

## Contoh Kode dan Penjelasan

```typescript // 1. Class Decorator function Logger(prefix: string) { return function (target: Function) { console.log(`${prefix} ${target.name}`); }; }

// 2. Method Decorator function LogMethod( target: any, propertyKey: string, descriptor: PropertyDescriptor ) { // Simpan method asli const originalMethod = descriptor.value;

// Modifikasi method  
descriptor.value = function (...args: any[]) {  
 console.log(\`Memanggil method \${propertyKey} dengan args: \${JSON.stringify(args)}\`);  
 const result = originalMethod.apply(this, args);  
 console.log(\`Method \${propertyKey} mengembalikan: \${JSON.stringify(result)}\`);  
 return result;  
};

}

// 3. Property Decorator function ValidateNotEmpty(target: any, propertyKey: string) { let value: string;

const getter = function() {  
 return value;  
};  
  
const setter = function(newVal: string) {  
 if (!newVal || newVal.trim().length === 0) {  
 throw new Error(\`\${propertyKey} tidak boleh kosong\`);  
 }  
 value = newVal;  
};  
  
Object.defineProperty(target, propertyKey, {  
 get: getter,  
 set: setter,  
 enumerable: true,  
 configurable: true  
});

}

// 4. Parameter Decorator function Required(target: any, propertyKey: string, parameterIndex: number) { const requiredParams: number[] = Reflect.getMetadata(“required”, target, propertyKey) || []; requiredParams.push(parameterIndex); Reflect.defineMetadata(“required”, requiredParams, target, propertyKey); }

// 5. Decorator Factory dengan Options interface LogOptions { before?: boolean; after?: boolean; }

function Log(options: LogOptions = {}) { return function ( target: any, propertyKey: string, descriptor: PropertyDescriptor ) { const originalMethod = descriptor.value;

descriptor.value = function (...args: any[]) {  
 if (options.before) {  
 console.log(\`Sebelum \${propertyKey}\`);  
 }  
  
 const result = originalMethod.apply(this, args);  
  
 if (options.after) {  
 console.log(\`Setelah \${propertyKey}\`);  
 }  
  
 return result;  
 };  
};

}

// 6. Praktik Penggunaan Decorators @Logger(“Class:”) class Produk { @ValidateNotEmpty nama: string;

constructor(nama: string) {  
 this.nama = nama;  
}  
  
@LogMethod  
@Log({ before: true, after: true })  
hitungHarga(@Required harga: number, @Required jumlah: number): number {  
 return harga \* jumlah;  
}

}

// 7. Advanced Decorator: Dependency Injection interface Type { new (…args: any[]): T; }

class Container { private static services = new Map<string, any>();

static register<T>(token: string, type: Type<T>): void {  
 Container.services.set(token, new type());  
}  
  
static resolve<T>(token: string): T {  
 const service = Container.services.get(token);  
 if (!service) {  
 throw new Error(\`Service \${token} tidak ditemukan\`);  
 }  
 return service;  
}

}

function Inject(token: string) { return function(target: any, propertyKey: string) { Object.defineProperty(target, propertyKey, { get: () => Container.resolve(token), enumerable: true, configurable: true }); }; }

// 8. Real-world Example: API Controller function Controller(prefix: string = ““) { return function(target: Function) { Reflect.defineMetadata(”prefix”, prefix, target); }; }

function Get(path: string = ““) { return function(target: any, propertyKey: string) { const routes = Reflect.getMetadata(”routes”, target.constructor) || []; routes.push({ method: “GET”, path, handler: propertyKey }); Reflect.defineMetadata(“routes”, routes, target.constructor); }; }

@Controller(“/api/products”) class ProductController { @Get(“/”) getAllProducts() { return [{ id: 1, name: “Product 1” }]; }

@Get("/:id")  
getProductById(@Required id: number) {  
 return { id, name: \`Product \${id}\` };  
}

}

// 9. Validation Decorators function MinLength(min: number) { return function(target: any, propertyKey: string) { let value: string;

const getter = function() {  
 return value;  
 };  
   
 const setter = function(newVal: string) {  
 if (newVal.length < min) {  
 throw new Error(  
 \`\${propertyKey} harus memiliki minimal \${min} karakter\`  
 );  
 }  
 value = newVal;  
 };  
   
 Object.defineProperty(target, propertyKey, {  
 get: getter,  
 set: setter,  
 enumerable: true,  
 configurable: true  
 });  
};

}

// 10. Performance Monitoring Decorator function Measure() { return function( target: any, propertyKey: string, descriptor: PropertyDescriptor ) { const originalMethod = descriptor.value;

descriptor.value = async function(...args: any[]) {  
 const start = performance.now();  
 const result = await originalMethod.apply(this, args);  
 const end = performance.now();  
 console.log(  
 \`\${propertyKey} dieksekusi dalam \${end - start}ms\`  
 );  
 return result;  
 };  
};

}

// Penggunaan class UserService { @MinLength(3) username: string;

constructor(username: string) {  
 this.username = username;  
}  
  
@Measure()  
async fetchUserData() {  
 // Simulasi API call  
 await new Promise(resolve => setTimeout(resolve, 1000));  
 return { username: this.username };  
}

} ```

## Cara Kerja TypeScript Decorators

1. **Execution Order**:
   * Parameter Decorators
   * Method Decorators
   * Property Decorators
   * Class Decorators
2. **Metadata Reflection**:
   * Menggunakan reflect-metadata
   * Menyimpan metadata runtime
   * Mengakses informasi tipe
3. **Decorator Evaluation**:
   * Dievaluasi saat deklarasi
   * Dieksekusi saat runtime
   * Dapat dikomposisi

## Tips dan Trik

1. **Decorator Composition**

* // ✅ Gunakan multiple decorators  
  @Logger  
  @Controller("/api")  
  class ApiService {  
   @Validate  
   @Measure()  
   method() {}  
  }

1. **Error Handling**

* // ✅ Tangani error dalam decorator  
  function SafeOperation() {  
   return function(  
   target: any,  
   propertyKey: string,  
   descriptor: PropertyDescriptor  
   ) {  
   const original = descriptor.value;  
   descriptor.value = function(...args: any[]) {  
   try {  
   return original.apply(this, args);  
   } catch (error) {  
   console.error(\`Error in \${propertyKey}:\`, error);  
   throw error;  
   }  
   };  
   };  
  }

1. **Reusable Decorators**

* // ✅ Buat decorator yang bisa digunakan ulang  
  function Memoize() {  
   const cache = new Map();  
    
   return function(  
   target: any,  
   propertyKey: string,  
   descriptor: PropertyDescriptor  
   ) {  
   const original = descriptor.value;  
    
   descriptor.value = function(...args: any[]) {  
   const key = JSON.stringify(args);  
   if (cache.has(key)) {  
   return cache.get(key);  
   }  
   const result = original.apply(this, args);  
   cache.set(key, result);  
   return result;  
   };  
   };  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Mengabaikan Execution Order**

* // ❌ Buruk: Tidak memperhatikan urutan  
  @Decorator2  
  @Decorator1  
  class Example {}  
    
  // ✅ Baik: Memperhatikan urutan eksekusi  
  @InitializeFirst  
  @ProcessSecond  
  class Example {}

1. **Tidak Menangani Error**

* // ❌ Buruk: Tidak ada error handling  
  function Decorator() {  
   return function(target: any) {  
   // Operasi yang bisa error  
   };  
  }  
    
  // ✅ Baik: Dengan error handling  
  function Decorator() {  
   return function(target: any) {  
   try {  
   // Operasi yang bisa error  
   } catch (error) {  
   console.error("Decorator error:", error);  
   }  
   };  
  }

1. **Overuse of Decorators**

* // ❌ Buruk: Terlalu banyak decorator  
  @Validate  
  @Log  
  @Cache  
  @Track  
  @Initialize  
  class TooManyDecorators {}  
    
  // ✅ Baik: Gunakan decorator seperlunya  
  @Validate  
  @Log  
  class JustEnoughDecorators {}

### Solusi:

1. Pahami urutan eksekusi decorator
2. Selalu implementasi error handling
3. Gunakan decorator seperlunya
4. Buat decorator yang reusable
5. Dokumentasikan decorator dengan baik