Table of Contents

# Bab 6: Generics di TypeScript

## Penjelasan Materi

Generics adalah salah satu fitur paling kuat di TypeScript yang memungkinkan kita untuk menulis kode yang dapat bekerja dengan berbagai tipe data sambil tetap mempertahankan type safety. Dengan generics, kita bisa membuat komponen yang dapat digunakan ulang dan fleksibel tanpa mengorbankan keamanan tipe data. Generics memungkinkan kita untuk menulis fungsi, class, dan interface yang dapat bekerja dengan berbagai tipe data yang berbeda.

## Analogi yang Mudah Dipahami

Bayangkan Generics seperti kotak serbaguna: - Kotak bisa menyimpan berbagai jenis barang - Saat membuat kotak, kita tentukan jenis barang yang akan disimpan - Setelah ditentukan, kotak hanya bisa menyimpan barang jenis itu - Kita bisa membuat kotak baru untuk jenis barang yang berbeda

Atau seperti cetakan kue: - Satu cetakan bisa digunakan untuk berbagai adonan - Bentuk kue akan sama tapi isinya bisa berbeda - Setiap kali membuat, kita tentukan jenis adonannya - Hasilnya tetap terjamin sesuai bentuk cetakan

## Point Penting

1. **Generic Functions**
   * Type parameters
   * Multiple type parameters
   * Generic constraints
   * Default type parameters
2. **Generic Classes**
   * Generic class declaration
   * Generic methods
   * Static members
   * Constructor constraints
3. **Generic Interfaces**
   * Generic interface declaration
   * Generic type aliases
   * Extending generic interfaces
   * Implementation in classes
4. **Generic Constraints**
   * extends keyword
   * Multiple constraints
   * keyof constraint
   * Type parameters in constraints
5. **Best Practices**
   * Naming conventions
   * When to use generics
   * Common patterns
   * Error handling

## Contoh Kode dan Penjelasan

```typescript // 1. Generic Function function identitas(arg: T): T { return arg; }

// Penggunaan let str = identitas(“Hello”); // type: string let num = identitas(42); // type: number (type inference)

// 2. Generic Interface interface Wadah { nilai: T; tambah: (item: T) => void; ambil: () => T; }

// 3. Generic Class class KoleksiData { private items: T[] = [];

tambah(item: T): void {  
 this.items.push(item);  
}  
  
ambil(index: number): T {  
 return this.items[index];  
}  
  
get ukuran(): number {  
 return this.items.length;  
}

}

// 4. Multiple Type Parameters function tukar<T, U>(tuple: [T, U]): [U, T] { return [tuple[1], tuple[0]]; }

// 5. Generic Constraints interface HasLength { length: number; }

function logPanjang(arg: T): number { console.log(arg.length); return arg.length; }

// 6. Generic Class dengan Constraints class DaftarTerbatas { private items: T[] = [];

tambah(item: T): void {  
 if (item.length > 0) {  
 this.items.push(item);  
 }  
}

}

// 7. Generic dengan Default Types interface ResponseAPI<T = any> { data: T; status: number; message: string; }

// 8. Generic Utility Types interface Pengguna { id: number; nama: string; email: string; }

type PartialPengguna = Partial; type ReadonlyPengguna = Readonly; type PenggunaKeys = keyof Pengguna;

// 9. Advanced Generic Patterns class StateManager { private state: S;

constructor(initialState: S) {  
 this.state = initialState;  
}  
  
getState(): S {  
 return this.state;  
}  
  
setState(newState: Partial<S>): void {  
 this.state = { ...this.state, ...newState };  
}

}

// 10. Real-world Example: Generic Repository Pattern interface Entity { id: number; }

class GenericRepository { private items: T[] = [];

create(item: Omit<T, "id">): T {  
 const newItem = {  
 ...item,  
 id: this.generateId()  
 } as T;  
 this.items.push(newItem);  
 return newItem;  
}  
  
read(id: number): T | undefined {  
 return this.items.find(item => item.id === id);  
}  
  
update(id: number, item: Partial<T>): T | undefined {  
 const index = this.items.findIndex(i => i.id === id);  
 if (index !== -1) {  
 this.items[index] = { ...this.items[index], ...item };  
 return this.items[index];  
 }  
 return undefined;  
}  
  
delete(id: number): boolean {  
 const index = this.items.findIndex(i => i.id === id);  
 if (index !== -1) {  
 this.items.splice(index, 1);  
 return true;  
 }  
 return false;  
}  
  
private generateId(): number {  
 return Math.max(0, ...this.items.map(i => i.id)) + 1;  
}

}

// Penggunaan interface Produk extends Entity { nama: string; harga: number; stok: number; }

const produkRepo = new GenericRepository();

const laptop = produkRepo.create({ nama: “Laptop Gaming”, harga: 15000000, stok: 10 });

console.log(laptop); // { id: 1, nama: “Laptop Gaming”, harga: 15000000, stok: 10 }

// 11. Generic Factory Pattern interface Konstruktor { new (…args: any[]): T; }

class Factory { constructor(private type: Konstruktor) {}

create(...args: any[]): T {  
 return new this.type(...args);  
}

}

// Penggunaan Factory class Mobil { constructor( public merek: string, public tahun: number ) {} }

const mobilFactory = new Factory(Mobil); const mobil = mobilFactory.create(“Toyota”, 2023); ```

## Cara Kerja TypeScript Generics

1. **Type Checking**:
   * TypeScript memeriksa tipe saat kompilasi
   * Memastikan konsistensi tipe dalam penggunaan
   * Memberikan type inference yang kuat
2. **Type Erasure**:
   * Informasi generic dihapus saat runtime
   * Kompilasi ke JavaScript regular
   * Tidak ada overhead performa
3. **Constraint Checking**:
   * Memeriksa batasan tipe saat kompilasi
   * Memastikan tipe memenuhi persyaratan
   * Memberikan error yang jelas

## Tips dan Trik

1. **Naming Conventions**

* // ✅ Gunakan nama yang deskriptif untuk type parameters  
  interface Repository<TEntity> {  
   find(id: number): TEntity;  
  }  
    
  // ✅ Gunakan T, U, V untuk parameter sederhana  
  function swap<T, U>(tuple: [T, U]): [U, T] {  
   return [tuple[1], tuple[0]];  
  }

1. **Constraint Best Practices**

* // ✅ Gunakan constraint yang spesifik  
  interface HasId {  
   id: number;  
  }  
    
  class Repository<T extends HasId> {  
   findById(id: number): T | undefined {  
   // Implementation  
   }  
  }

1. **Default Type Parameters**

* // ✅ Sediakan default type yang masuk akal  
  interface Config<T = string> {  
   data: T;  
   timestamp: Date;  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Overuse of Generics**

* // ❌ Buruk: Generic tidak diperlukan  
  function print<T>(value: T): void {  
   console.log(value);  
  }  
    
  // ✅ Baik: Cukup gunakan any atau unknown  
  function print(value: unknown): void {  
   console.log(value);  
  }

1. **Tidak Menggunakan Constraints**

* // ❌ Buruk: Tidak ada constraint  
  function getLength<T>(arg: T): number {  
   return arg.length; // Error: length tidak dijamin ada  
  }  
    
  // ✅ Baik: Menggunakan constraint  
  function getLength<T extends { length: number }>(arg: T): number {  
   return arg.length;  
  }

1. **Mengabaikan Type Inference**

* // ❌ Buruk: Tipe eksplisit tidak perlu  
  let nilai = identitas<string>("hello");  
    
  // ✅ Baik: Biarkan type inference bekerja  
  let nilai = identitas("hello");

### Solusi:

1. Gunakan generics hanya ketika diperlukan
2. Selalu pertimbangkan constraints yang sesuai
3. Manfaatkan type inference TypeScript
4. Dokumentasikan penggunaan generics
5. Gunakan naming convention yang konsisten