Table of Contents

# Bab 11: Advanced Types di TypeScript

## Penjelasan Materi

Advanced Types di TypeScript menyediakan fitur-fitur type system yang lebih canggih untuk menangani kasus-kasus kompleks dalam pengembangan aplikasi. Fitur-fitur ini memungkinkan kita untuk membuat tipe yang lebih ekspresif, aman, dan fleksibel, serta membantu dalam pembuatan abstraksi yang lebih baik.

## Analogi yang Mudah Dipahami

Bayangkan Advanced Types seperti alat-alat canggih dalam bengkel: - Union Types seperti obeng multi-mata yang bisa digunakan untuk berbagai jenis sekrup - Intersection Types seperti alat kombinasi yang menggabungkan fungsi palu dan tang - Conditional Types seperti mesin yang menghasilkan output berbeda berdasarkan input - Mapped Types seperti mesin duplikator yang membuat salinan dengan modifikasi - Type Guards seperti alat pengukur yang memastikan ketepatan penggunaan

## Point Penting

1. **Union dan Intersection Types**
   * Union Types (|)
   * Intersection Types (&)
   * Type Guards
   * Discriminated Unions
2. **Conditional Types**
   * extends keyword
   * infer keyword
   * Distributive Conditional Types
   * Nested Conditional Types
3. **Mapped Types**
   * Property Mapping
   * Key Remapping
   * Modifiers
   * Template Literal Types
4. **Type Inference**
   * Type Widening
   * Type Narrowing
   * Control Flow Analysis
   * Type Predicates
5. **Advanced Features**
   * Index Types
   * Recursive Types
   * Variadic Tuple Types
   * Template Literal Types

## Contoh Kode dan Penjelasan

```typescript // 1. Union dan Intersection Types type StringOrNumber = string | number; type NumberAndString = { num: number } & { str: string };

// Discriminated Unions type Shape = | { kind: “circle”; radius: number } | { kind: “rectangle”; width: number; height: number } | { kind: “triangle”; base: number; height: number };

function calculateArea(shape: Shape): number { switch (shape.kind) { case “circle”: return Math.PI \* shape.radius \*\* 2; case “rectangle”: return shape.width \* shape.height; case “triangle”: return (shape.base \* shape.height) / 2; } }

// 2. Conditional Types type IsString = T extends string ? true : false; type A = IsString; // true type B = IsString; // false

// Inferring within Conditional Types type UnwrapPromise = T extends Promise ? U : T; type PromiseString = Promise; type Result = UnwrapPromise; // string

// 3. Mapped Types dengan Modifiers type Optional = { [P in keyof T]?: T[P]; };

type Nullable = { [P in keyof T]: T[P] | null; };

// Key Remapping type Getters = { [P in keyof T as `get${Capitalize<string & P>}`]: () => T[P]; };

interface Person { name: string; age: number; }

type PersonGetters = Getters; // { // getName: () => string; // getAge: () => number; // }

// 4. Advanced Type Guards function isString(value: unknown): value is string { return typeof value === “string”; }

function processValue(value: unknown) { if (isString(value)) { // value is string here console.log(value.toUpperCase()); } }

// 5. Recursive Types type JSONValue = | string | number | boolean | null | JSONValue[] | { [key: string]: JSONValue };

// 6. Template Literal Types type EventName = “click” | “focus” | “blur”; type EventHandler = `on${Capitalize}`; type EventHandlers = { [E in EventName as EventHandler]: () => void; };

// 7. Advanced Example: API Type Safety type HTTPMethod = “GET” | “POST” | “PUT” | “DELETE”;

type Endpoint = “/users” | “/posts” | “/comments”;

type APIResponse = { data: T; status: number; message: string; };

type User = { id: number; name: string; email: string; };

type Post = { id: number; title: string; content: string; userId: number; };

// Mapped type untuk endpoint responses type EndpointResponse = { “/users”: User[]; “/posts”: Post[]; “/comments”: Comment[]; };

// Type-safe API client class APIClient { async request( method: HTTPMethod, endpoint: E ): Promise<APIResponse<EndpointResponse[E]>> { // Implementation return {} as any; } }

// 8. Advanced Pattern Matching type Pattern = T extends any[] ? “array” : T extends Function ? “function” : T extends object ? “object” : “primitive”;

type P1 = Pattern<string[]>; // “array” type P2 = Pattern<() => void>; // “function” type P3 = Pattern<{ a: 1 }>; // “object” type P4 = Pattern; // “primitive”

// 9. Advanced Inference with Tuples type Head<T extends any[]> = T extends [infer H, …any[]] ? H : never; type Tail<T extends any[]> = T extends [any, …infer R] ? R : never;

type H1 = Head<[1, 2, 3]>; // 1 type T1 = Tail<[1, 2, 3]>; // [2, 3]

// 10. Practical Example: Form Validation System type ValidationRule = { validate: (value: T) => boolean; message: string; };

type FieldValidation = { [P in keyof T]: ValidationRule<T[P]>[]; };

interface LoginForm { username: string; password: string; rememberMe: boolean; }

const loginValidation: FieldValidation = { username: [ { validate: (value) => value.length >= 3, message: “Username must be at least 3 characters” } ], password: [ { validate: (value) => value.length >= 8, message: “Password must be at least 8 characters” }, { validate: (value) => /[A-Z]/.test(value), message: “Password must contain at least one uppercase letter” } ], rememberMe: [] };

function validate(obj: T, validation: FieldValidation): string[] { const errors: string[] = [];

for (const key in validation) {  
 const rules = validation[key];  
 const value = obj[key];  
   
 rules.forEach(rule => {  
 if (!rule.validate(value)) {  
 errors.push(rule.message);  
 }  
 });  
}  
  
return errors;

} ```

## Cara Kerja Advanced Types

1. **Type Resolution**:
   * Type inference
   * Type compatibility
   * Structural typing
   * Nominal typing hints
2. **Type Manipulation**:
   * Type transformation
   * Type composition
   * Type distribution
   * Type inference
3. **Type Guards**:
   * Runtime checks
   * Type narrowing
   * Control flow analysis
   * Type predicates

## Tips dan Trik

1. **Type Guards yang Efektif**

* // ✅ Gunakan type predicates untuk custom type guards  
  function isError(value: unknown): value is Error {  
   return value instanceof Error;  
  }  
    
  // ✅ Gunakan type guards dengan union types  
  function processResult(result: number | Error) {  
   if (isError(result)) {  
   console.error(result.message);  
   return;  
   }  
   console.log(result.toFixed(2));  
  }

1. **Conditional Types yang Bersih**

* // ✅ Gunakan helper types untuk conditional types yang kompleks  
  type If<C extends boolean, T, F> = C extends true ? T : F;  
  type NonEmptyArray<T> = T[] & { 0: T };  
    
  // Penggunaan  
  type Result = If<true, string, number>; // string

1. **Mapped Types yang Efisien**

* // ✅ Gunakan template literals dengan mapped types  
  type Setters<T> = {  
   [P in keyof T as \`set\${Capitalize<string & P>}\`]: (value: T[P]) => void;  
  };

## Kesalahan yang Sering Dilakukan Pemula

1. **Overcomplicating Types**

* // ❌ Buruk: Terlalu kompleks  
  type ComplexType<T> = T extends any  
   ? T extends string  
   ? string  
   : T extends number  
   ? number  
   : never  
   : never;  
    
  // ✅ Baik: Lebih sederhana dan jelas  
  type SimpleType<T> = T extends string | number ? T : never;

1. **Incorrect Type Guards**

* // ❌ Buruk: Type guard yang tidak tepat  
  function isUser(obj: any): obj is User {  
   return obj.name !== undefined;  
  }  
    
  // ✅ Baik: Type guard yang lebih robust  
  function isUser(obj: unknown): obj is User {  
   return (  
   typeof obj === "object" &&  
   obj !== null &&  
   "name" in obj &&  
   typeof obj.name === "string"  
   );  
  }

1. **Misusing Conditional Types**

* // ❌ Buruk: Conditional type yang tidak perlu  
  type ToString<T> = T extends any ? string : never;  
    
  // ✅ Baik: Langsung gunakan type  
  type ToString = string;

### Solusi:

1. Gunakan type guards untuk type safety
2. Buat helper types untuk reusability
3. Manfaatkan conditional types dengan bijak
4. Hindari kompleksitas yang tidak perlu
5. Dokumentasikan tipe-tipe kompleks