Advanced TypeScript Typing: Simplifying Complexity

By Adi Utama

Have you ever seen this kind of code?

```
type ElementType<P = any> =
  {
      [K in keyof JSX.IntrinsicElements]: P extends JSX.IntrinsicElements[K] ? K : never
  }[keyof JSX.IntrinsicElements] |
  ComponentType<P>;

type ComponentType<P = {}> = ComponentClass<P> | FunctionComponent<P>;

type JSXElementConstructor<P> =
  | ((props: P) => ReactElement<any, any> | null)
  | (new (props: P) => Component<any, any>);
```

https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/index.d.ts#L71

What is TypeScript?

> TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

Why TypeScript?

1. Type safety
2. Code readability & maintainability
3. Scalability
4. Better IDE support

Feature	Static Typing	Dynamic Typing
Type Declaration	Required	Optional
Type Checking	Compile-time	Runtime
Error Detection	Can catch errors at compile-time	Can only catch errors at runtime
Flexibility	Less flexible, but more reliable	More flexible, but potentially less reliable
Learning Curve	Steeper learning curve	Easier to learn
Examples	Java, TypeScript, C++	JavaScript, Python, Ruby

Basic TypeScript Types

- Primitive types: `number`, `string`, `boolean`
- Array types
- Function types
- Object types
- Enum types

Pro Tips: Never use `enum`!

Type Operator

- `typeof` : Used to get the type of a value at runtime or the type of a variable or function at compile-time.
- `instanceof` : Used to check if an object is an instance of a specific class at runtime.
- `keyof` : Used to get the keys of an object type as a union of string literal types.
- `in` : Used to check if a property exists on an object.
- `as` : Used for type assertions, which allow you to tell TypeScript that you know the type of a value better than it does.

Advance TypeScript Types

- Literal types
- Indexed Access Types
- Union types
- Intersection types
- Conditional Types
- Mapped Types
- Generic types

Literal Types

> Literal types represent a single value, and can be used to create more specific types for greater type safety.

Example

```
type MyName = 'Adi'
type MyNumber = 62850303030
```

Indexed Access Types

> Indexed access types allow you to access the type of a specific property of an object by its key.

Example

```
type Person = {
  firstName: string
  lastName: string
}

type PersonFirstName = Person['firstName']
type PersonLastName = Person['lastName']
```

Union Types

> Union types in TypeScript allow a variable to have more than one possible type.

Example:

```typescript
type Programmer = {
  job: 'programmer'
  language: ProgrammingLanguage[]
}

type ProgrammingLanguage = 'TypeScript' | 'JavaScript' | 'CoffeeScript' | 'ActionScript'
type ProgrammerFields = keyof Jobs
```

Intersection Types

> Intersection types allow you to combine multiple types into a single type that has all of their properties.

Example:

```
type ProgrammerPerson = Person & Jobs['programmer']
```

Conditional Types

> Conditional types allow you to create types that depend on other types

Example:

```typescript
type MaybeProgrammer = ProgrammerPerson extends Programmer ? Programmer : unknown
```

Mapped Types

> Mapped types allow you to create new types by transforming each property of an existing type.

Example:

```
type ReadonlyPerson = { readonly [Key in keyof Person]: Person[Key] }
type PartialPerson = { [Key in keyof Person]?: Person[Key] }
```

Generic Types

> Generic types in TypeScript allow you to create reusable types that can work with a variety of data types.

Example:

```
type Maybe<T, U> = T extends U ? U : unknown
type Readonly<Target> = { readonly [K in keyof Target]: Target[K] }
type Partial<T extends object> = { readonly [K in keyof T]?: T[K] }
```

Case Studies

Live coding session

Thank you!

> The only way to learn a new programming language is by writing programs in it
>
> ~ Dennis Ritchie

Visit:

http://adiutama.xyz