TypeScript Utility Types

By Web Dev Simplified https://courses.webdevsimplified.com

`Pick<Obj, Keys>`

Create a new object type that contains only the specified keys from the original object.

```
type User = { id: number; name: string; age: number; }
type UserFormFields = Pick<User, 'name' | 'age'>
//   ^= { name: string; age: number; }
```

Useful for creating subsets of object types without needing to redefine every property. Use `Pick` whenever your type is derived from the existing object type.

Opposite of `Omit`.

`Omit<Obj, Keys>`

Create a new object type that contains all properties except the specified keys from the original object.

```
type User = { id: number; name: string; age: number; }
type UserWithoutAgeOrName = Omit<User, 'age' | 'name'>
//   ^= { id: number; }
```

Useful for creating subsets of object types without needing to redefine every property. Use `Omit` whenever your type is derived from the existing object type.

Opposite of `Pick`.

`Record<Keys, Values>`

Create an object type with specific keys that all have the same value type.

```
type UserRole = 'admin' | 'user' | 'guest'
type Permissions = Record<UserRole, string[]>
//   ^= { admin: string[]; user: string[]; guest: string[]; }
```

Perfect for creating dictionaries or maps with known keys. Commonly used for configurations, mappings, and lookup tables.

`Partial<Obj>`

Create a new object type where all properties of the original object are optional.

```
type User = { id: number; name: string; age: number; }
type PartialUser = Partial<User>
//   ^= { id?: number; name?: string; age?: number; }
```

Useful for update operations where you only want to modify some properties. Commonly used in forms, API updates, and configuration objects.

Opposite of `Required`.

`Required<Obj>`

Create a new object type where all properties of the original object are required.

```
type PartialUser = { id?: number; name?: string; age?: number; }
type RequiredUser = Required<PartialUser>
//   ^= { id: number; name: string; age: number; }
```

Useful when you need to ensure all properties are defined. This is often used in forms and database inserts.

Opposite of `Partial`.

`Readonly<Obj>`

Create a new object type where all properties of the original object are readonly.

```
type User = { id: number; name: string; }
type ReadonlyUser = Readonly<User>
//   ^= { readonly id: number; readonly name: string; }

type NamesArray = string[]
type ReadonlyNames = Readonly<NamesArray>
//   ^= readonly string[]
```

Useful for creating immutable data structures and ensuring data integrity.

Can also be used with arrays to create immutable arrays.

Parameters<Func>

Extract the parameter types from a function type as a tuple.

```
type GreetFunction = (name: string, age: number) => string
type GreetParams = Parameters<GreetFunction>
//   ^= [string, number]
```

Useful for creating wrapper functions, currying, or when you need to access the type of individual parameters from libraries that don't expose those types.

ReturnType<Func>

Extract the return type from a function type.

```
function getUser(id: number) {
  return { id, name: "John" }
}
type User = ReturnType<typeof getUser>
//   ^= { id: number; name: string; }
```

Perfect when you need to work with the return type of a function that uses implicit types. Commonly used with libraries that don't expose these return types.

ConstructorParameters<Class>

Extract the parameter types from a constructor as a tuple.

```
class Database {
  constructor(host: string, port: number, ssl: boolean) {}
}
type DatabaseParams = ConstructorParameters<typeof Database>
//   ^= [string, number, boolean]
```

Useful for factory functions, dependency injection, or when you need to work with constructor arguments programmatically.

Similar to `Parameters` but specifically for constructors.

`Extract<Union, Members>`

Create a new union type that contains only the specified members from the original union.

```
type Colors = 'red' | 'green' | 'blue' | 'yellow' | 'orange'
type RedishColors = Extract<Colors, "red" | "yellow" | "orange">
//   ^= 'red' | 'yellow' | 'orange'
```

Useful for creating subsets of union types without needing to redefine every member. Use `Extract` whenever your union is derived from the existing union type.

Opposite of `Exclude`, and similar to `Pick`.

`Exclude<Union, Members>`

Create a new union type that contains all members except the specified members from the original union.

```
type Colors = 'red' | 'green' | 'blue' | 'yellow' | 'orange'
type RedishColors = Exclude<Colors, "blue" | "green">
//   ^= 'red' | 'yellow' | 'orange'
```

Useful for creating subsets of union types without needing to redefine every member. Use `Exclude` whenever your union is derived from the existing union type.

Opposite of `Extract`, and similar to `Omit`.

`NonNullable<T>`

Remove null and undefined from the type.

```
type MaybeString = string | null | undefined
type DefiniteString = NonNullable<MaybeString>
//   ^= string
```

Useful for type narrowing. Commonly used in optional chaining and null checking scenarios.

This is similar to `Required`, but it removes `null` and `undefined` from the type while `Required` just makes all properties required (even if the value of those properties can still be `null` or `undefined`).

`Awaited<PromiseType>`

Get the type that a Promise resolves to, recursively unwrapping nested Promises.

```
type NestedPromise = Promise<Promise<number>>
type UnwrappedNumber = Awaited<NestedPromise>
//   ^= number
```

Perfect for working with async functions and it handles nested Promises automatically.

`Uppercase<StringType>`

Convert each character in the string type to uppercase.

```
type Greeting = 'Hello World'
type LoudGreeting = Uppercase<Greeting>
//   ^= 'HELLO WORLD'
type WithUnion = Uppercase<'Union' | 'Types'>
//   ^= 'UNION' | 'TYPES'
```

Useful for creating consistent casing in APIs, constants, and configuration objects. Can be used with strings or unions of strings.

Opposite of `Lowercase`.

`Lowercase<StringType>`

Convert each character in the string type to lowercase.

```
type Greeting = 'Hello World'
type QuietGreeting = Lowercase<Greeting>
//   ^= 'hello world'
type WithUnion = Lowercase<'Union' | 'Types'>
//   ^= 'union' | 'types'
```

Perfect for normalizing strings to lowercase for consistent comparisons and data processing. Can be used with strings or unions of strings.

Opposite of `Uppercase`.

`Capitalize<StringType>`

Convert the first character of the string type to uppercase.

```
type Greeting = 'hello WORLD'
type CapitalGreeting = Capitalize<Greeting>
//   ^= 'Hello WORLD'
type Animal = Capitalize<'cat' | 'dog' | 'bird'>
//   ^= 'Cat' | 'Dog' | 'Bird'
```

Commonly used for converting camelCase to PascalCase for type definitions. Can be used with strings or unions of strings.

Opposite of `Uncapitalize`.

`Uncapitalize<StringType>`

Convert the first character of the string type to lowercase.

```
type Greeting = 'Hello WORLD'
type CapitalGreeting = Uncapitalize<Greeting>
//   ^= 'hello WORLD'
type InstanceName = Uncapitalize<'UserService' | 'DataProcessor'>
//   ^= 'userService' | 'dataProcessor'
```

Useful for converting PascalCase to camelCase, and creating instance names from class names. Can be used with strings or unions of strings.

Opposite of `Capitalize`.

`InstanceType<Class>`

Extract the instance type from a constructor.

```
class User {
  constructor(public name: string, public age: number) {}
}
type UserInstance = InstanceType<typeof User>
//   ^= User
```

Don't use this utility type with classes since you can use the type from the class instead.