2025/02/06

TS Challenge - 4

Keita Kawabata

Problem 1

- If -

Problem 1

Implement the util type `If<C, T, F>` which accepts condition `C`, a truthy value `T`, and a falsy value `F`.

```
// C is expected to be either true or false
// while T and F can be any type.

type A = If<true, 'a', 'b'>  // expected to be 'a'
type B = If<false, 'a', 'b'> // expected to be 'b'
```

Solution

```
type If<C extends boolean, T, F> = C extends true ? T : F;
```

Prerequisites

1. Generics
2. Conditional Type

Prerequisites

1. Generics
2. Conditional Type

Conditional Type - Basic

Allow you to define types dynamically based on conditions.

```
// T extends U ? X : Y
type IsString<T> = T extends string ? true : false;

type a = IsString<"a"> // true
type b = IsString<0> // false
```

Problem 1

Implement the util type `If<C, T, F>` which accepts condition `C`, a truthy value `T`, and a falsy value `F`.

```
// C is expected to be either true or false
// while T and F can be any type.

type A = If<true, 'a', 'b'>  // expected to be 'a'
type B = If<false, 'a', 'b'> // expected to be 'b'
```

Solution

```
type If<C extends boolean, T, F> = C extends true ? T : F;
```

Problem 2

- Exclude -

Problem 2

Implement the built-in `Exclude<T, U>`

```
// Exclude from T those types that are assignable to U

type Result = MyExclude<'a' | 'b' | 'c', 'a'> // 'b' | 'c'
```

Solution

```
type MyExclude<T, U> =  T extends U ? never : T;
```

Prerequisites

1. Generics
2. `never` type
3. Distributive Conditional Type

Prerequisites

1. Generics
2. `never` type
3. Distributive Conditional Type

`never` type

- `never` is an empty type — it represents nothing.

```ts
const foo: never = 1; ❌

const any: any = 1;
const bar: never = any; // ❌ Even if `any` type

const nev: never = 1 as never;
const str: string = nev; // ✅ can be assigned to any type.

// Usecase: A function that always throws an error
function throwError(): never {
  throw new Error();
}
```

Prerequisites

1. Generics
2. `never` type
3. **Distributive Conditional Type**

Distributive Conditional Type

`T extends U ? X : Y`

- if `T` is a union type, the condition is applied to each member of the union separately.

```ts
type IsString<T> = T extends string ? true : false;


type c = IsString<string | number>;
// Equivalent to: IsString<string> | IsString<number>
// Resolves to: true | false
```

Distributive Conditional Type

- with `never` type

```ts
type MyExtract<T, U> = T extends U ? T : never;

// `never` in a union gets removed automatically
type A = MyExtract<"a" | "b" | "c", "a" | "c">;  // "a" | "c"
```

Problem 2

Implement the built-in `Exclude<T, U>`

```
// Exclude from T those types that are assignable to U

type Result = MyExclude<'a' | 'b' | 'c', 'a'> // 'b' | 'c'
```

Solution

```
type MyExclude<T, U> =  T extends U ? never : T;
```