2025/02/11

TS Challenge - 5

Keita Kawabata

Problem 1

- Awaited -

Problem 1

If we have a type which is a wrapped type like `Promise`, how can we get the type which is inside the wrapped type?

```
// For example:
// if we have Promise<ExampleType> how to get ExampleType?

type ExampleType = Promise<string>
type Result = MyAwaited<ExampleType> // string
```

Solution

```ts
type MyAwaited<T> = T extends Promise<infer U> ? U : never;
```

Prerequisites

1. Conditional Type
2. `infer`

Prerequisites

1. Conditional Type
2. `infer`

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
```

Prerequisites

1. Conditional Type
2. `infer`

`infer` - Basic

- `infer` is a type operator used in Conditional Types.
- It means "to infer" and can only be written on the right side of extends

```
const foo = () => "";  // Returns a string
const bar = () => 0;   // Returns a number

type Return<T> = T extends () => infer R ? R : never;

type A = Return<typeof foo>; // string
type B = Return<typeof bar>; // number
```

Problem 1

If we have a type which is a wrapped type like `Promise`, how can we get the type which is inside the wrapped type?

```
// For example:
// if we have Promise<ExampleType> how to get ExampleType?

type ExampleType = Promise<string>
type Result = MyAwaited<ExampleType> // string
```

Solution

```ts
type MyAwaited<T> = T extends Promise<infer U> ? U : never;
```

Problem 2

- First of Array -

Problem 2

Implement a generic `First<T>` that takes an Array T and returns its first element's type.

```
type arr1 = ['a', 'b', 'c']
type arr2 = [3, 2, 1]

type head1 = First<arr1> // expected to be 'a'
type head2 = First<arr2> // expected to be 3
```

Solution

```ts
// 1
type First<T extends any[]> = T extends [] ? never : T[0]

// 2
type First<T extends any[]> = T['length'] extends 0 ? never : T[0]

// 3
type First<T extends unknown[]> = T extends [infer U, ...unknown[]] ? U : never
```

Prerequisites

1. Conditional Type
2. Spread Syntax
3. `infer`

Prerequisites

1. Conditional Type
2. **Spread Syntax**
3. `infer`

Spread Syntax - Basic

A syntax that expands elements of arrays and objects

```
const arr = [1, 2, 3];

const arr2 = [...arr, 4]; // expected to be [1, 2, 3, 4]
const arr3 = [0, ...arr]; // expected to be [0, 1, 2, 3]
const arr4 = [0, ...arr, 4]; // expected to be [0, 1, 2, 3, 4]
```

Spread Syntax - Type

```
type Tuple = [number, boolean]

type ArrayType = [string, ...Tuple]; // [string, number, boolean]
```

Problem 2

Implement a generic `First<T>` that takes an Array T and returns its first element's type.

```
type arr1 = ['a', 'b', 'c']
type arr2 = [3, 2, 1]

type head1 = First<arr1> // expected to be 'a'
type head2 = First<arr2> // expected to be 3
```

Solution

```ts
// 1
type First<T extends any[]> = T extends [] ? never : T[0]

// 2
type First<T extends any[]> = T['length'] extends 0 ? never : T[0]

// 3
type First<T extends unknown[]> = T extends [infer U, ...unknown[]] ? U : never
```

Related Problem

- Last of Array -

Related Problem

Implement a generic `Last<T>` that takes an Array T and returns its last element.

```
type arr1 = ['a', 'b', 'c']
type arr2 = [3, 2, 1]

type tail1 = Last<arr1> // expected to be 'c'
type tail2 = Last<arr2> // expected to be 1
```

Solution

```ts
type Last<T extends unknown[]> = T extends [...unknown[], infer U] ? U : never

// or

type Last<T extends unknown[]> = [unknown, ...T][T["length"]];
```