2025/02/19

TS Challenge - 6

Keita Kawabata

Problem 1

- Parameters -

Problem 1

Implement the built-in Parameters generic without using it.

```
const foo = (arg1: string, arg2: number): void => {}

type FunctionParamsType = MyParameters<typeof foo> // [arg1: string, arg2: number]
```

Solution

```typescript
type MyParameters<T extends (...args: any[]) => unknown>
  = T extends (...args: infer U) => unknown ? U : false;
```

Prerequisites

1. Function Type
2. Conditional Type
3. `infer`
4. `any` & `unknown` type

Prerequisites

1. **Function Type**
2. Conditional Type
3. `infer`
4. `any` & `unknown` type

Function Type

```
type Increment = (num: number) => number;

// arrow function
const increment: Increment = (num) => num + 1;

// function
const increment: Increment = function (num) { return num + 1; };
```

with Spread Syntax

```typescript
type Add = (...args: number[]) => number;

const add: Add = (...args) => {
  return args.reduce((acc, curr) => acc + curr, 0);
};

// Result
console.log(add(1, 2, 3, 4)); // Output: 10
console.log(add(5, 5));       // Output: 10
console.log(add(10));         // Output: 10
```

Prerequisites

1. Function Type
2. Conditional Type
3. `infer`
4. `any` & `unknown` type

`infer`

- `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
```

Prerequisites

1. Function Type
2. Conditional Type
3. `infer`
4. **`any` & `unknown` type**

`any` & `unknown`

```
// `any`
let anyValue: any = 42;
anyValue = "Hello"; // 🥲 No error
anyValue.toString(); // 🥲 No error, but fail at runtime

// `unknown`
let unknownValue: unknown = 42;
unknownValue = "Hello"; // 🥲 No error

// 😁 Error: Cannot access methods directly
unknownValue.toString(); // Error
unknownValue.toUpperCase(); // Error

// Need "Type guard" to access methods
if (typeof unknownValue === 'string') {..}
```

`any` & `unknown`

- `infer U` works with `any[]`
  - elements have no restrictions
- `unknown[]` blocks inference
  - `unknown` is restrictive
  - TS avoids inference sometimes

```typescript
const foo = (arg1: string, arg2: number): void => {}

// any[]
type MyParameters1<T extends (...args: any[]) => unknown>
  = T extends (...unknown: infer U) => unknown ? U : false

// unknown[]
type MyParameters2<T extends (...args: unknown[]) => unknown>
  = T extends (...unknown: infer U) => unknown ? U : false

type A = MyParameters1<typeof foo> // ✅
type B = MyParameters2<typeof foo> // ❌
```

Problem 1

Implement the built-in Parameters generic without using it.

```
const foo = (arg1: string, arg2: number): void => {}

type FunctionParamsType = MyParameters<typeof foo> // [arg1: string, arg2: number]
```

Solution

```typescript
type MyParameters<T extends (...args: any[]) => unknown>
  = T extends (...args: infer U) => unknown ? U : false;
```

Problem 2

- Includes -

Problem 2

Implement the JS `Array.includes` function in the type system. A type takes the two arguments. The output should be a boolean `true` or `false`.

```
// expected to be `false`
type isPillarMen = Includes<['Kars', 'Esidisi', 'Wamuu', 'Santana'], 'Dio'>
```

Solution - minimum

```typescript
type Includes<T extends unknown[], U> = U extends T[number] ? true : false;
```

Solution - best

```
type IsEqual<X, Y> =
    (<T>() => T extends X ? 1 : 2) extends
    (<T>() => T extends Y ? 1 : 2) ? true : false;


type Includes<T extends readonly unknown[], U> =
  T extends [infer First, ...infer Rest]
    ? IsEqual<First, U> extends true ? true : Includes<Rest, U>
    : false;
```

Prerequisites

1. Conditional Type
2. `T[number]`

Prerequisites

1. Conditional Type
2. `T[number]`

`T[number]`

allows you to extract the union of the types of all elements of the tuple

```
type T = [string, number, boolean];

type TypeAtIndex0 = T[0]; // string
type TypeAtIndex1 = T[1]; // number
type TypeAtIndex2 = T[2]; // boolean

type ElementTypes = T[number]; // string | number | boolean
```

Dive into Best Solution

Dive into Best Solution

- one example where unione one fails;

```
type Includes<T extends unknown[], U> = U extends T[number] ? true : false;

type A = Includes<[{ a: 1 }, { a: 2 }], { a: 1 }>;
type B = Includes<[{}], { a: "A" }> // Expected: false, but actual: true
```

Dive into Best Solution

```
type IsEqual<X, Y> =
    (<T>() => T extends X ? 1 : 2) extends
    (<T>() => T extends Y ? 1 : 2) ? true : false;


type Includes<T extends readonly unknown[], U> =
  T extends [infer First, ...infer Rest]
    ? IsEqual<First, U> extends true ? true : Includes<Rest, U>
    : false;
```

IsEqual<X, Y>

```
type IsEqual<X, Y> =
    (<T>() => T extends X ? 1 : 2) extends
    (<T>() => T extends Y ? 1 : 2) ? true : false;
```

`IsEqual<X, Y>`

this would allow partial matches :(

```
type IsEqual1<X, Y> = X extends Y ? true : false;

type A = IsEqual1<string, string> // ✅ true
type B = IsEqual1<"Hello", string> // ❌ true
```

`IsEqual<X, Y>`

this would not allow partial matches (※ No need to learn that much)

```
type IsEqual<X, Y> =
  (<T>() => T extends X ? 1 : 2) extends
  (<T>() => T extends Y ? 1 : 2) ? true : false;


type C = IsEqual<string, string> // ✅ true
type D = IsEqual<"Hello", string> // ✅ false
```

`Includes<>`

`Includes` recursively until Rest runs out or U is found

```
type Includes<T extends readonly unknown[], U> =
  T extends [infer First, ...infer Rest]
    ? IsEqual<First, U> extends true ? true : Includes<Rest, U>
    : false;
```

Problem 2

Implement the JS `Array.includes` function in the type system. A type takes the two arguments. The output should be a boolean `true` or `false`.

```
// expected to be `false`
type isPillarMen = Includes<['Kars', 'Esidisi', 'Wamuu', 'Santana'], 'Dio'>
```

Solution - minimum

```typescript
type Includes<T extends unknown[], U> = U extends T[number] ? true : false;
```

Outro

- Thank you so much for your participation!
- Next:
  - Scheduled reading group:
    - Software Engineering at Google
  - Starting within the next month.
  - See you soon!