2025/01/30

TS Challenge - 3

Keita Kawabata

Problem 1

- Push -

Problem 1

Implement the generic version of `Array.push`

```
type Result = Push<[1, 2], '3'> // [1, 2, '3']
```

Solution

```ts
type Push<T extends unknown[], U> = [...T, U];

// or

type Push<T extends readonly unknown[], U> = [...T, U];
```

Prerequisites

1. Generics
2. `any` / `unknown` type
3. Spread syntax

Prerequisites

1. Generics
2. `any` / `unknown` type
3. Spread syntax

Generics - Basic

A way to create reusable code that works with multiple types

```
type Foo<T> = {
  bar: T
};

type FooString = Foo<string>;
type FooNumber = Foo<number>;
```

Generics - Extends

Extends allows you to limit a generic type to a specific type

```
type Foo<T extends string> = {
  bar: T
};

type FooString = Foo<string>; // OK
type FooNumber = Foo<number>; // NG
```

Prerequisites

1. Generics
2. `any` / `unknown` type
3. Spread syntax

any

Allow all types, any operation

```
let value: any = "hello"; // ✅

value = 42; // ✅: Any type can be assigned to 'any'.

const bool: boolean = value; // ✅: 'any' can be assigned to any type without
error.

value.toUpperCase(); // ✅: No type checking
```

unknown

"Type-safe counterpart of any"

```
let value: unknown = "hello"; // ✅
value = 42; // ✅: Any type can be assigned to 'unknown'.

const num: number = value; // ❌: Cannot assign 'unknown' to 'number' directly.

value.toString(); // ❌: Property 'toString' does not exist on type 'unknown'

if (typeof value === "number") {
  console.log(value.toString()); // ✅: with TypeGuard
}
```

Prerequisites

1. Generics
2. `any` / `unknown` type
3. Spread syntax

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 1

Implement the generic version of `Array.push`

```
type Result = Push<[1, 2], '3'> // [1, 2, '3']
```

Solution

```ts
type Push<T extends unknown[], U> = [...T, U];

// or

type Push<T extends readonly unknown[], U> = [...T, U];
```

Problem 2

- Unshift -

Problem 2

Implement the type version of Array.unshift

```
type Result = Unshift<[1, 2], 0> // [0, 1, 2]
```

Solution

```ts
type Unshift<T extends unknown[], U> = [U, ...T];

// or

type Unshift<T extends readonly unknown[], U> = [U, ...T];
```

Problem 3

- Concat -

Problem 3

Implement the JavaScript `Array.concat` function in the type system.

```
// A type takes the two arguments.
// The output should be a new array that includes inputs in ltr order

type Result = Concat<[1], [2]> // expected to be [1, 2]
```

Solution

```ts
type Concat<T extends unknown[], U extends unknown[]>
= [...T,...U]

// or

type Concat<T extends readonly unknown[], U extends readonly unknown[]>
= [...T,...U]
```

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

- `[unknown, ...T]`
  - `T = [3, 2, 1]` becomes `[unknown, 3, 2, 1]`
- Without `[unknown, ...T]`
  - `[...T][T["length"]]` = `T[3]` is `undefined` (out of bounds.)

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

// or

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