

2025/01/23

# TS Challenge - 2

Keita Kawabata

## Problem 1

- Length of Tuple -

# Problem 1

Create a generic `Length`, pick the length of the tuple

```
type tesla = ['tesla', 'model 3', 'model X', 'model Y']
type spaceX = ['FALCON 9', 'FALCON HEAVY', 'DRAGON', 'STARSHIP', 'HUMAN SPACEFLIGHT']

type teslaLength = Length<tesla> // expected 4
type spaceXLength = Length<spaceX> // expected 5
```

# Solution

```
// Solution 1
```

```
type Length<T extends any[]> = T["length"];
```

```
// Solution 2
```

```
type Length<T extends Array<any>> = T["length"];
```

```
// Solution 3
```

```
type Length<T extends unknown[]> = T["length"];
```

# Which is the prefer solution ?

```
// Solution 1
```

```
type Length<T extends any[]> = T["length"];
```

```
// Solution 2
```

```
type Length<T extends Array<any>> = T["length"];
```

```
// Solution 3
```

```
type Lentgh<T extends unknown[]> = T["length"];
```

## Best solution

```
type Lentgh<T extends unknown[]> = T["length"];
```

## Prerequisites

1. `any` / `unknown` type
2. Generics
3. index access types

## Prerequisites

1. `any` / `unknown` type

2. Generics

3. index access types



# 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. `any` / `unknown`

2. **Generics**

3. index access types

# 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. `any` / `unknown`

2. Generics

**3. index access types**

# index access types

Can get the type of a specific key in object

```
interface Todo = {  
  title: string;  
  description: string;  
}
```

```
type Title = Todo["title"]; // string
```

**In Addition, it can access properties in TypeScript type definitions.**

```
type Length<T extends unknown[]> = T.length;    // NG
```

```
type Length<T extends unknown[]> = T['length']; // OK
```



# Problem 1

Create a generic `Length`, pick the length of the tuple

```
type tesla = ['tesla', 'model 3', 'model X', 'model Y']
type spaceX = ['FALCON 9', 'FALCON HEAVY', 'DRAGON', 'STARSHIP', 'HUMAN SPACEFLIGHT']

type teslaLength = Length<tesla> // expected 4
type spaceXLength = Length<spaceX> // expected 5
```

## Best solution

```
type Lentgh<T extends unknown[]> = T["length"];
```

## Problem 2

- Tuple to Object -

# Problem 2

transform it into an object type and the key / value must be in the provided array.

```
const tuple = ['tesla', 'model 3', 'model X', 'model Y'] as const

type result = TupleToObject<typeof tuple>

// expected
{
  'tesla': 'tesla',
  'model 3': 'model 3',
  'model X': 'model X',
  'model Y': 'model Y'
}
```

# Solution

it works, but we should avoid using `any` type

```
type TupleToObject<T extends readonly any[]> = {  
  [K in T[number]]: K  
}
```

# Solution

```
type TupleToObject<T extends readonly PropertyKey[]> = {  
  [K in T[number]]: K  
}
```

## Prerequisites

1. `as const` (const assertion)
2. `PropertyKey` type
3. Mapped Types
4. `T[number]` (Tuple)

## Prerequisites

1. `as const` (const assertion)
2. `PropertyKey` type
3. Mapped Types
4. `T[number]` (Tuple)



## as const

Makes the variable readonly deeply with literal types

```
// Arrays
const arr = [1, 2, 3] as const; // readonly [1, 2, 3]

// Objects
const obj = {
  x: 1,
  y: { z: 2 }
} as const;

// type: {
//   readonly x: 1,
//   readonly y: { readonly z: 2 }
// }
```

## Prerequisites

1. `as const` (const assertion)
2. `PropertyKey` type
3. Mapped Types
4. `T[number]` (Tuple)

# PropertyKey

- `PropertyKey` is union type: `string | number | symbol`
  - In TypeScript, object can only have 3 types of values as keys: `string`, `number` and `symbol`

```
// ✓  
type TupleToObject<T extends readonly (string | number | symbol)[]> = {  
  [K in T[number]]: K  
}  
  
// ✓ + Readability  
type TupleToObject<T extends readonly PropertyKey[]> = {  
  [K in T[number]]: K  
}
```

## Prerequisites

1. `as const` (const assertion)

2. `PropertyKey` type

### 3. Mapped Types

4. `T[number]` (Tuple)

# Mapped Types

Can create a new object type  
based on a union of keys

```
type TodoKeys = "title" | "description";

type Todo = {
  [K in TodoKeys]: string
};

// It is same as follows;
type Todo = {
  title: string;
  description: string;
};
```

## Prerequisites

1. `as const` (const assertion)
2. `PropertyKey` type
3. Mapped Types
4. `T[number]` (Tuple)

# What is Tuple

- a special Array type
- Has a fixed length
- Has specific types for each elements

```
type MyTuple = [string, number, boolean];  
  
const example: MyTuple = ["hello", 42, true];
```

## 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
```



## T[number] with const assertion

```
const tuple = ["tesla", "model 3", "model X", "model Y"] as const;

// Type of 'tuple' with readonly & literal types
type T = typeof tuple; // readonly ["tesla", "model 3", "model X", "model Y"];

T[number]; // "tesla" | "model 3" | "model X" | "model Y"
```

## Problem 2

transform it into an object type and the key / value must be in the provided array.

```
const tuple = ['tesla', 'model 3', 'model X', 'model Y'] as const

type result = TupleToObject<typeof tuple>

// expected
{
  'tesla': 'tesla',
  'model 3': 'model 3',
  'model X': 'model X',
  'model Y': 'model Y'
}
```

## Best solution

```
type Lentgh<T extends unknown[]> = T["length"];
```

## Related Problem

- Tuple to Union -

## Related Problem

Implement a generic `TupleToUnion<T>` which covers the values of a tuple to its values union.

```
type Arr = ['1', '2', '3']
```

```
type Test = TupleToUnion<Arr> // expected to be '1' | '2' | '3'
```

# Solution

- `unknown` is the best choice in this case
  - `unknown` permits any element types (e.g., `boolean`)
  - `PropertyKey` permits only `string | number | symbol` types

```
type TupleToUnion<T extends unknown[]> = T[number];
```