2025/01/16

TS Challenge - 1

Keita Kawabata

Problem 1

Let's define a `MyReadonly` type!

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

const todo: MyReadonly<Todo> = {
  title: "Hey",
  description: "foobar"
};

todo.title = "Hello"; // Error: cannot reassign a readonly property
todo.description = "barFoo"; // Error: cannot reassign a readonly property
```

```ts
const todo: MyReadonly<Todo> = {
  title: "Hey",
  description: "foobar"
};

type MyReadonly<Todo> = {
  readonly title: "Hey",
  readonly description: "foobar"
};
```

Solution

```
type MyReadonly<T> = {
  readonly [K in keyof T]: T[K]
};
```

Prerequisites

1. Generics
2. keyof operator
3. Mapped Types
4. index access types

Prerequisites

1. Generics
2. keyof operator
3. Mapped Types
4. 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. Generics
2. keyof operator
3. Mapped Types
4. index access types

keyof

A method to get object keys as a union type

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

type TodoKeys = keyof Todo; // "title" | "description"
```

Prerequisites

1. Generics
2. keyof operator
3. **Mapped Types**
4. index access types

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;
};
```

Mapped Types

with keyof operator

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

type Todo1 = {
  [K in keyof Todo]: string;
};
```

Prerequisites

1. Generics
2. keyof operator
3. Mapped Types
4. **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
```

Let's define a `MyReadonly` type!

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

const todo: MyReadonly<Todo> = {
  title: "Hey",
  description: "foobar"
};

todo.title = "Hello"; // Error: cannot reassign a readonly property
todo.description = "barFoo"; // Error: cannot reassign a readonly property
```

Solution

```
type MyReadonly<T> = {
  readonly [K in keyof T]: T[K]
};
```

Related Probelms - ①

Implement a generic `MyReadonly2<T, K>` which takes two type argument T and K.

```ts
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

const todo: MyReadonly2<Todo, 'title' | 'description'> = {
  title: "Hey",
  description: "foobar",
  completed: false,
};

todo.title = "Hello"; // Error: cannot reassign a readonly property
todo.description = "barFoo"; // Error: cannot reassign a readonly property
todo.completed = true; // OK
```

Solution

- `Omit<Type, Keys>`
  - Creates a new type by excluding properties K from type T.
- `Pick<Type, Keys>`
  - Creates a new type by selecting only the properties K from type T.

```
type MyReadonly2<T, K extends keyof T = keyof T> = Omit<T, K> &
Readonly<Pick<T, K>>;
```

Related Probelms - ②

Implement a generic `DeepReadonly<T>` which make every parameter of an object and its sub-objects recursively readonly.

```ts
type X = {
  x: {
    a: 1;
    b: 'hi';
  };
  y: 'hey';
};

type Expected = {
  readonly x: {
    readonly a: 1;
    readonly b: 'hi';
  };
  readonly y: 'hey';
};

type Todo = DeepReadonly<X>; // should be same as Expected
```

Solution

- **If T is a primitive type**
  - it directly returns T with readonly
  - because primitive types don't have properties
- **If T is an object type, the type applies recursion**
  - It iterates over all properties of T using keyof T.

```
type DeepReadonly<T> = T extends any ? {
  readonly [P in keyof T]: keyof T[P] extends never ? T[P] : DeepReadonly<T[P]>;
} : never;
```