Mutable and Immutable Arrays

In this lesson, you will learn about two different syntaxes to define an array.

We'll cover the following

- Mutable arrays
- Immutable arrays
- Control flow analysis for array construction

Mutable arrays#

Arrays in TypeScript are exactly like the ones in JavaScript in terms of features. The difference is that TypeScript assigns a type to the list.

The syntax, as shown below, utilizes square brackets `[]` with the actual type before the brackets and after the colon `:` like so:

```
1  let a: number[];
```

It is also possible to initialize values during the declaration. In the following example, the two arrays are typed. TypeScript infers that the first one is a `number` and the second is a `string`. You can move your cursor over the two variables to see the types.

```
1  let arrayOfNumber = [1, 2, 3];
```

```
2
3  let arrayOfString = ["string", "array", "only"];
```

Using multiple types will require you to evaluate what the type of each value is before using an individual item of the array. This is because the variable's operations are type-dependent. There is an equivalent syntax that uses the generic `Array<T>`. Both are the same.

```
let usingArraySyntax: Array<number> = [1, 2, 3];
```

You can initialize an empty list by setting the variable equal to empty square brackets. An array can use one or many unions to allow multiple types. Unions will be discussed later. At that stage, remember that with multiple types, you must wrap the union with parentheses.

The following code demonstrates the answer to the previous quiz.

```
let multipleTypeArray = [1, true, 3];
// Same as:
let multipleTypeArrayExplicit: (number | boolean)[] = [1, true, 3];
```

The preference to explicitly declare type or not depends on your style of coding. For maintainability, it is good practice to be explicit. If you take a peek at the code above, you may notice that the array takes two types without having to read further. It also forces future additions to be constrained by the expected type. Without explicitly typing the variable,

someone could add a string and suddenly, the variable allows a number, boolean, and string.

Before moving on, it is important to note that you can also instantiate a strongly-typed object array. This is equivalent to creating a new array without assigning any values.

```
let myArray = new Array<number>();
printArray(myArray);

// Is the same as:
let myArray2: Array<number> = [];
printArray(myArray2);

// Is the same as:
let myArray3: number[] = [];
printArray(myArray3);

function printArray(a: number[]): void {
    console.log(`Before: ${a}`);
    a.push(1);
    console.log(`After: ${a}`);
}
```

Immutable arrays#

While the two syntaxes presented above refer to mutable collections, there is also the possibility of creating a list that is immutable. The `ReadonlyArray` is a generic array that only allows you to read from the array once it's constructed. As with the mutable array, there are two ways to write a read-only collection.

The first approach is to use `ReadonlyArray<T>` instead of `Array<T>`.

```
let list: ReadonlyArray<number> = [1, 2];
list.push(3);
console.log(list);
```

The code above does not compile because you cannot mutate the array with `push`. This error is interesting and introduces the second way to write a read-only collection.

> The property ‘push’ does not exist on type 'readonly `number[]`'

The second way is to use the keyword `readonly` in front of the type and square brackets.

```
let list: readonly number[] = [1, 2];
list.push(3);
console.log(list);
```

You may ask yourself, “What is the difference between a constant array and a read-only array?” The answer is that a constant array won’t let you assign values to a list while a read-only array blocks you from changing values.

```
const list1: number[] = [1, 2];
list1.push(3); // Legit because list1 is not re-assigned.
// list1 = [4, 5]; // ERROR: We cannot reassign a constant
```

Compared to read-only:

```
let list1: readonly number[] = [1, 2];
// list1.push(3); // Error, cannot mutate the content
list1 = [4, 5]; // Legit, content is not mutated, we create a new list
```

Control flow analysis for array construction#

Control flow analysis is how TypeScript dynamically figures out what type it should infer. Arrays can be tricky since they accept many values which can be of many types. For example, the following code accepts a list of numbers or strings:

```
let myArray: (number | string)[] = [];
```

If we let TypeScript figure it out, we will have:

```
let myArray  = [];
```

The last question may surprise you. The empty array is an evolving type that will be analyzed during the “flow” of the code, meaning depending on what happens with the following operations. Functions like `push`, `shift`, `unshift`, or setting directly to an index a value `myArray[index] = value` will transform the type. The type is finally attributed once it stops changing, hence question #4 gets to its real type at the end of the code, not before, which could be anything.

In this lesson, we saw two different syntaxes used to write an array in TypeScript. We also illustrated that it is possible to have the content of an

TypeScript. We also illustrated that it is possible to have the content of an

array to be immutable with `readonly`. We described the difference between a constant and a read-only list as well.

Back

Understanding and using the void type

Next

Undefined Versus Null

Report an Issue