Using Generics in Angular

Introduction

In this guide, we’ll be learning about generics in Angular components that are capable of working on the data of today as well as the data of tomorrow. They will give you the most flexible capabilities for building up large software systems.

In many languages, one of the main tools in the toolbox for creating reusable components is generics. That is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their types.

Beginning with Typescript

Let’s now see how we can start using the generics. Let’s consider a myFunction is a function that will return whatever is passed in. You can think of this in a similar way to the echo command.

Let’s now look at an example, without generics, where we would either have to give the myFunction function a specific type or use ‘any’ type. As you can see, we are using a string type.

1

2

3

function myFunction(name: string): string {

return name;

}

typescript

Or, we could describe the identity function using any type.

1

2

3

function myFunction(value: any): any {

return value;

}

typescript

While using any is certainly generic in that it will cause the function to accept any type for the type of value, we are actually the information about what that type was when the function returns. If we passed in a string, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of argument in such a way that we can also use it to denote what is being returned. Here, we will use a special kind of variable that works on types rather than values.

1

2

3

function myFunction<T>(val: T): T {

return val;

}

typescript

We’ve now added a type variable T to the identity function. This T allows us to capture the type the user provides (e.g. string) so that we can use that information later. Here, we use T again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the myFunction function is generic, as it works over a range of types. Unlike using any, it’s also just as precise as the first identity function that used numbers for the argument and return type. Once we’ve written the generic myFunction function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function.

1

let output = myFunction<number>(number);

typescript

Using Generics with Types

In the above section, we have created the generic function myFunction that worked over different generic types. In this section, we’ll explore the type of the functions themselves and how to create generic interfaces. The type of generic function is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations.

1

2

3

4

5

6

7

8

9

10

function myFunction<T>(arg: T): T {

return arg;

}

let myValue: <T>(arg: T) => T = myFunction;

//Using a different name for the generic type parameter in the type

function myFunction <T>(arg: T): T {

return arg;

}

let myValue: <U>(arg: U) => U = myFunction;

typescript

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up. We can also write the generic type as a call signature of an object literal type. Which leads us to write our first generic interface. Let’s take the object literal from the previous example and move it to an interface.

1

2

3

4

5

6

7

8

9

10

11

12

13

function myFunction<T>(arg: T): T {

return arg;

}

let myValue: {<T>(arg: T): T} = myFunction;

// Using generics with an interface

interface GenericMyFunction {

<T>(arg: T): T;

}

function myFunction<T>(arg: T): T {

return arg;

}

let myValue: GenericMyFunction = myFunction;

typescript

We may want to move the generic parameter to be a parameter of the whole interface. This lets us see what types we are generic over (e.g. Dictionary rather than just Dictionary). This makes the type parameter visible to all the other members of the interface.

1

2

3

4

5

6

7

interface GenericMyFunction <T> {

(arg: T): T;

}

function myFunction<T>(arg: T): T {

return arg;

}

let myValue: GenericMyFunction <number> = myFunction;

typescript

The above example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use GenericMyFunction, we will now also need to specify the corresponding type argument, effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

Using Generics with Classes

A class in generic has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (<>) following the name of the class.

1

2

3

4

5

6

7

8

9

class GenericMyFunction <T> {

zeroValue: T;

add: (x: T, y: T) => T;

}

let myGenericMyFunction = new GenericMyFunction <number>();

myGenericMyFunction.zeroValue = 0;

myGenericMyFunction.add = function(x, y) {

return x + y;

};

typescript

This is a pretty literal use of the GenericMyFunction class, but you may have noticed that nothing is restricting it to only use the number type. We could have used string or even more complex objects instead.

1

2

3

4

5

let stringNumeric = new GenericMyFunction<string>(); stringNumeric.zeroValue = "";

stringNumeric.add = function(x, y) {

return x + y;

};

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));

typescript

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

Constraints in Generics

In this section, let’s see how we can use constraints in generics. We may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In this example, we can see loggingValue example, we wanted to be able to access the .length property of arg, but the compiler could not prove that every type had a .length property, so it warns us that we can’t make this assumption.

1

2

3

4

5

6

7

interface Lengthwise {

length: number;

}

function loggingValue<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

typescript

Now we know it has a .length property, so no more error. To do so, in the above example, we have created an interface that describes our constraint. Here, we will create an interface that has a single .length property and then we will use this interface and the extends keyword to denote our constraint. We need to pass in values whose type has all the required properties.

1

loggingValue({length: 12, value: 3});

typescript

Conclusion

In this guide, we have explored how we can use generics to create reusable functions in Angular. We have also seen how we can use generics along with interface and classes in our application.

You can learn more about Using the Async Pipe in Angular in my guide [Using the Async Pipe in Angular](https://app.pluralsight.com/guides/using-async-pipe-angular).