TypeScript Generics

Summary: in this tutorial, you'll learn about TypeScript generics that allow you to use types as formal parameters.

Introduction to TypeScript Generics

TypeScript generics allow you to write reusable and generalized forms of functions, classes, and interfaces. In this tutorial, you're focusing on developing generic functions.

It'll be easier to explain TypeScript generics through a simple example.

Suppose you need to develop a function that returns a random number in an array of numbers.

The following `getRandomNumberElement()` function takes an array of numbers as its parameter and returns a random element from the array:

```
function getRandomNumberElement(items: number[]): number {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

To get a random element of an array, you need to:

- Find the random index first.
- Get the random element based on the random index.

To find the random index of an array, we use the `Math.random()` that returns a random number between 0 and 1, multiplies by the length of the array, and applies the `Math.floor()` on the

result.

The following shows how to use the `getRandomNumberElement()` function:

```
let numbers = [1, 5, 7, 4, 2, 9];
console.log(getRandomNumberElement(numbers));
```

Assuming that you need to get a random element from an array of strings. This time, you may come up with a new function:

```
function getRandomStringElement(items: string[]): string {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

The logic of the `getRandomStringElement()` function is the same as in the `getRandomNumberElement()` function.

This example shows how to use the `getRandomStringElement()` function:

```
let colors = ['red', 'green', 'blue'];
console.log(getRandomStringElement(colors));
```

Later you may need to get a random element in an array of objects. Creating a new function every time you want to get a random element from a new array type is not scalable.

Using any type

One option to address this issue is to set the type of the array argument as `any[]`. By doing this, you need to write one function that works with an array of `any` type.

```
function getRandomAnyElement(items: any[]): any {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

The `getRandomAnyElement()` works with an array of the `any` type including number, string, object, and so on:

```
let numbers = [1, 5, 7, 4, 2, 9];
let colors = ['red', 'green', 'blue'];

console.log(getRandomAnyElement(numbers));
console.log(getRandomAnyElement(colors));
```

This option works fine but has a drawback: it doesn't allow you to enforce the type of the returned element. In other words, it isn't type-safe.

A better solution to avoid code duplication while preserving the type is to use generics.

TypeScript generics come to the rescue

The following shows a generic function that returns the random element from an array of type `T`:

```
function getRandomElement<T>(items: T[]): T {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

This function uses type variable `T`. The `T` allows you to capture the type provided when calling the function. Additionally, the function uses the `T` type variable as its return type.

This `getRandomElement()` function is generic because it can work with any data type including string, number, object,...

By convention, we use the letter `T` as the type variable. However, you can freely use other letters such as `A`, `B` `C`, ...

Calling a generic function

The following shows how to use the `getRandomElement()` with an array of numbers:

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement<number>(numbers);
console.log(randomEle);
```

This example explicitly passes `number` as the `T` type into the `getRandomElement()` function.

In practice, you'll use type inference for the argument. It means that you let the TypeScript compiler set the value of `T` automatically based on the type of argument that you pass into, like this:

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement(numbers);
console.log(randomEle);
```

In this example, we didn't pass the `number` type to the `getRandomElement()` explicitly. The compiler looks at the argument and sets `T` to its type.

Now, the `getRandomElement()` function is also type-safe. For example, if you assign the returned value to a string variable, you'll get an error:

```
let numbers = [1, 5, 7, 4, 2, 9];
let returnElem: string;
returnElem = getRandomElement(numbers);  // compiler error
```

Generic functions with multiple types

The following illustrates how to develop a generic function with two type variables `U` and `V`:

```
function merge<U, V>(obj1: U, obj2: V) {
    return {
        ...obj1,
        ...obj2
    };
}
```

The `merge()` function merges two objects with the type `U` and `V`. It combines the properties of the two objects into a single object.

Type inference infers the returned value of the `merge()` function as an intersection type of `U` and `V`, which is `U & V`

The following illustrates how to use the `merge()` function that merges two objects:

```
let result = merge(
    { name: 'John' },
    { jobTitle: 'Frontend Developer' }
);

console.log(result);
```

Output:

```
{ name: 'John', jobTitle: 'Frontend Developer' }
```

Benefits of TypeScript generics

The following are the benefits of TypeScript generics:

- Leverage type checks at the compile time.
- Eliminate type castings.
- Allow you to implement generic algorithms.

Summary

- Use TypeScript generics to develop reusable, generalized, and type-safe functions, interfaces, and classes.