✓ 100 XP

Introduction to generics

5 minutes

In earlier modules in the Build JavaScript applications by using TypeScript learning path, you learned how to apply type annotations to your interfaces, functions, and classes to create strongly typed components. But what if you want to create a component that can work over a variety of types rather than just one? You could use the `any` type, but then you lose the power behind the TypeScript type checking system.

Generics are code templates that you can define and reuse throughout your codebase. They provide a way to tell functions, classes, or interfaces what type you want to use when you call it. You can think about this in the same way that arguments are passed to a function, except a generic enables you to tell the component what type it should expect at the time it's called.

Create generic functions when your code is a function or class that:

- Works with a variety of data types.
- Uses that data type in several places.

Generics can:

- Provide more flexibility when working with types.
- Enable code reuse.
- Reduce the need to use the `any` type.

Why use generics?

To understand why you might use generics, it's helpful to see an example.

The `getArray` function generates an array of items of `any` type.

TypeScript

```typescript
function getArray(items : any[]) : any[] {
    return new Array().concat(items);
}
```

Then, the `numberArray` variable is declared by calling the `getArray` function, passing to it an array of numbers, and the `stringArray` variable is declared with an array of strings. However, because the `any` type is used, there's nothing preventing the code from pushing a `string` to the `numberArray` or a `number` to the `stringArray`.

TypeScript

```typescript
let numberArray = getArray([5, 10, 15, 20]);
let stringArray = getArray(['Cats', 'Dogs', 'Birds']);
numberArray.push(25);                       // OK
stringArray.push('Rabbits');                // OK
numberArray.push('This is not a number');   // OK
stringArray.push(30);                       // OK
console.log(numberArray);                   // [5, 10, 15, 20, 25, "This is not a
number"]
console.log(stringArray);                   // ["Cats", "Dogs", "Birds",
"Rabbits", 30]
```

What if you want to determine the type of the values that the array will contain when you call the function and then have TypeScript do the work of type checking the values that you pass to it to ensure they are of that type? This is where generics come into play.

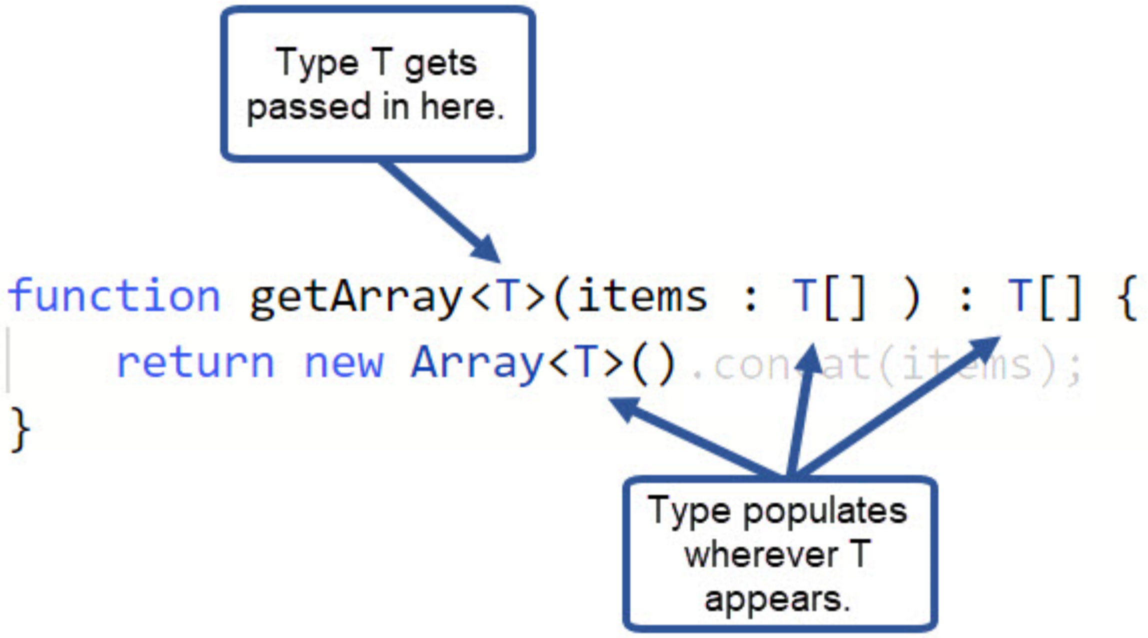
This example rewrites the `getArray` function using generics. It can now accept any type that you specify when calling the function.

TypeScript

```typescript
function getArray<T>(items : T[]) : T[] {
    return new Array<T>().concat(items);
}
```

Generics define one or more **type variables** to identify the type or types that you will pass to the component, enclosed in angle brackets (`< >`). (You'll also see type variables referred to as type parameters or generic parameters.) In the example above, the type variable in the function is called `<T>`. `T` is a commonly used name for a generic, but you can name it however you wish.

After you specify the type variable, you can use it in place of the type in parameters, the return type, or anywhere else in the function that you would add a type annotation.


Type T gets passed in here.
Type populates wherever T appears.

```
function getArray<T>(items : T[] ) : T[] {
    return new Array<T>().concat(items);
}
```

The type variable T can be used wherever the type annotation is needed. In the getArray function, it is used to specify the type for the items parameter, the function return type, and to return a new Array of items.

To call the function and pass a type to it, append `<type>` to the function name. For example, `getArray<number>` instructs the function to only accept an array of `number` values and return an array of `number` values. Because the type has been specified as a `number`, TypeScript will expect that `number` values will be passed to the function and will raise an error if it's something else.

Note

If you omit the type variable when calling the function, TypeScript will infer the type. However, this only works with simple data. Passing in arrays or objects infers the type of any and eliminates type checks.

In this example, with the variable declarations for `numberArray` and `stringArray` updated to call the function with the desired type, TypeScript prevents the invalid items from being added to the array.

TypeScript

```
let numberArray = getArray<number>([5, 10, 15, 20]);
numberArray.push(25);                      // OK
numberArray.push('This is not a number');  // Generates a compile time type check
```

```
error

let stringArray = getArray<string>(['Cats', 'Dogs', 'Birds']);
stringArray.push('Rabbits');                // OK
stringArray.push(30);                       // Generates a compile time type check
error
```

Using multiple type variables

You are not limited to using a single type variable in your generic components.

For example, the `identity` function accepts two parameters, `value` and `message`, and returns the `value` parameter. You can use two generics, `T` and `U`, to assign different types to each parameter and to the return type. The variable `returnNumber` is initialized by calling the `identity` function with `<number, string>` as the types for the `value` and `message` arguments, `returnString` is initialized by calling it with `<string, string>`, and `returnBoolean` is initialized by calling it with `<boolean, string>`. When using these variables, TypeScript can type check the values and return a compile-time error if there is a conflict.

TypeScript

```typescript
function identity<T, U> (value: T, message: U) : T {
    console.log(message);
    return value
}

let returnNumber = identity<number, string>(100, 'Hello!');
let returnString = identity<string, string>('100', 'Hola!');
let returnBoolean = identity<boolean, string>(true, 'Bonjour!');

returnNumber = returnNumber * 100;   // OK
returnString = returnString * 100;   // Error: Type 'number' not assignable to
type 'string'
returnBoolean = returnBoolean * 100; // Error: Type 'number' not assignable to
type 'boolean'
```

Next unit: Use the methods and properties of a generic type

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆