100 XP

Use the methods and properties of a generic type

5 minutes

When using type variables to create generic components, you may only use the properties and methods of objects that are available for **every** type. This prevents errors from occurring when you try to perform an operation on a parameter value that is incompatible with the type that's being passed to it.

If you add the statement `let result: T = value + value` to the `identity` function, TypeScript raises the error **The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type** because it doesn't know what value will be passed to it at runtime. If you were to pass a non-numeric value, the expression would generate an error, so TypeScript makes you aware of the problem at compile time.

TypeScript

```
function identity<T, U> (value: T, message: U) : T {
    let result: T = value + value;    // Error
    console.log(message);
    return result
}
```

Using generic constraints to limit types

The `identity` function can accept any type that you choose to pass to the type variables. But, in this case, you should constrain the types that the `value` parameter can accept to a range of types that you can perform an add operation on, rather than accepting any possible type. This is called a **generic constraint**.

There are several ways to do this depending on the type variable. One way is to declare a custom `type` as a tuple and then `extend` the type variable with the custom type. The following example declares `ValidTypes` as a tuple with a `string` and a `number`. Then, it extends `T` with the new type. Now, you can only pass `number` or `string` types to the type variable.

TypeScript

```
type ValidTypes = string | number;

function identity<T extends ValidTypes, U> (value: T, message: U) : T {
    let result: T = value + value;    // Error
    console.log(message);
    return result
}

let returnNumber = identity<number, string>(100, 'Hello!');      // OK
let returnString = identity<string, string>('100', 'Hola!');     // OK
let returnBoolean = identity<boolean, string>(true, 'Bonjour!'); // Error: Type
'boolean' does not satisfy the constraint 'ValidTypes'.
```

You can also constrain a type to the property of another object. This example uses `extends` with the `keyof` operator, which takes an object type and produces a string or numeric literal union of its keys. Here, `K extends keyof T`, ensuring that the key parameter is of the correct type for type assigned to `pet`.

TypeScript

```
function getPets<T, K extends keyof T>(pet: T, key: K) {
  return pet[key];
}

let pets1 = { cats: 4, dogs: 3, parrots: 1, fish: 6 };
let pets2 = { 1: "cats", 2: "dogs", 3: "parrots", 4: "fish"}

console.log(getPets(pets1, "fish"));  // Returns 6
console.log(getPets(pets2, "3"));     // Error
```

You'll learn more about using generic constraints with classes later in this module.

Using type guards with generics

You'll notice that TypeScript still raises an issue with the `value + value` expression in the `identity` function. But now you know that only `number` and `string` types can be passed to the function.

You can use the `typeof` type guard in an `if` block to check the type of the `value` parameter before performing an operation, as shown in the following example. TypeScript can determine

from the `if` statement if the operation will work with the values provided within the block.

TypeScript

```
type ValidTypes = string | number;
function identity<T extends ValidTypes, U> (value: T, message: U) {   // Return
type is inferred
    let result: ValidTypes = '';
    let typeValue: string = typeof value;

    if (typeof value === 'number') {           // Is it a number?
        result = value + value;                // OK
    } else if (typeof value === 'string') {    // Is it a string?
        result = value + value;                // OK
    }

    console.log(`The message is ${message} and the function returns a ${typeValue}
value of ${result}`);

    return result
}

let numberValue = identity<number, string>(100, 'Hello');
let stringValue = identity<string, string>('100', 'Hello');

console.log(numberValue);       // Returns 200
console.log(stringValue);       // Returns 100100
```

> **Note**
>
> You can only use a `typeof` type guard to check the primitive types `string`, `number`, `bigint`, `function`, `boolean`, `symbol`, `object`, and undefined. To check the type of a class, use an `instanceof` type guard.

Next unit: Exercise - Implement generics with interfaces and classes

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆