100 XP

Exercise - Implement generics with interfaces and classes

5 minutes

Generics are just a way to pass types to a component, so you can not only apply native types to generic type variables, but also interfaces, functions, and classes. In this unit, you'll see some different ways to use generics with these complex types.

Try using generics with interfaces, functions, and classes. All the code samples perform essentially the same tasks using different approaches.

Declare a generic interface

You can use generics in an interface declaration by replacing the type annotations with type variables.

1. Open the Playground and remove any existing code.
2. Declare a simple interface called `Identity` that has two properties, `value` and `message`, and two generic type variables, `T` and `U`, for the property types.

TypeScript

```
interface Identity<T, U> {
    value: T;
    message: U;
}
```

3. Declare two variables, using the `Identity` interface as an object type.

TypeScript

```
let returnNumber: Identity<number, string> = {
    value: 25,
    message: 'Hello!'
}
```

```
let returnString: Identity<string, number> = {
    value: 'Hello!',
    message: 25
}
```

Declare a generic interface as a function type

You can also declare a generic interface as a function type.

1. Continue working in the Playground.
2. Declare a generic interface called `ProcessIdentity` that includes the generic signature of a method, `(value: T, message: U): T`. Notice that the method doesn't have a name. By doing this, you can apply it to any function with a matching type signature.

TypeScript

```
interface ProcessIdentity<T, U> {
    (value: T, message: U): T;
}
```

3. Declare a function called `processIdentity` that has the same type signature as the `ProcessIdentity` interface.

TypeScript

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

4. Declare a function type variable called `processor` with the `ProcessIdentity` interface as the variable type, passing in `number` for the `T` type and `string` for the `U` type. Then, assign the `processIdentity` function to it. You can now use this variable as a function in your code and TypeScript will verify the types.

TypeScript

```
let processor: ProcessIdentity<number, string> = processIdentity;
let returnNumber1 = processor(100, 'Hello!');   // OK
let returnString1 = processor('Hello!', 100);   // Type check error
```

Declare a generic interface as a class type

You can also declare a generic interface and implement it in a class.

1. Continue working in the Playground.

2. Declare an interface called `ProcessIdentity` that has two properties, `value` and `message`, and two generic type variables, `T` and `U`, for the property types. Then, add a generic signature of a method called `process` that returns a value of type `T`.

   TypeScript

   ```
   interface ProcessIdentity<T, U> {
       value: T;
       message: U;
       process(): T;
   }
   ```

3. Define a generic class called `processIdentity` that implements the `ProcessIdentity` interface. In this case, name the variable types in the `processIdentity` class `X` and `Y`. You can use different variable names in the interface and the class because the type value propagates up the chain and the variable name doesn't matter.

   TypeScript

   ```
   class processIdentity<X, Y> implements ProcessIdentity<X, Y> {
       value: X;
       message: Y;
       constructor(val: X, msg: Y) {
           this.value = val;
           this.message = msg;
       }
       process() : X {
           console.log(this.message);
           return this.value
       }
   }
   ```

4. Declare a new variable and assign a new `processIdentity` object to it, passing in `number` and `string` for the `X` and `Y` variable types, and a `number` and `string` as the argument values.

```typescript
let processor = new processIdentity<number, string>(100, 'Hello');
processor.process();           // Displays 'Hello'
processor.value = '100';       // Type check error
```

Define a generic class

You can also declare a generic class without an interface. This example declares `processIdentity` as a generic class without implementing the `ProcessIdentity` interface.

```typescript
class processIdentity<T, U> {
    private _value: T;
    private _message: U;
    constructor(value: T, message: U) {
        this._value = value;
        this._message = message;
    }
    getIdentity() : T {
        console.log(this._message);
        return this._value
    }
}
let processor = new processIdentity<number, string>(100, 'Hello');
processor.getIdentity();      // Displays 'Hello'
```

Next unit: Implement generics with custom types and classes

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆