**Generics in TypeScript**

Generics are a fundamental feature of statically-typed languages, allowing developers to pass [types](https://www.digitalocean.com/community/tutorials/how-to-use-basic-types-in-typescript) as parameters to another type, [function](https://www.digitalocean.com/community/tutorials/how-to-use-functions-in-typescript), or other structure. When a developer makes their component a generic component, they give that component the ability to accept and enforce typing that is passed in when the component is used, which improves code flexibility, makes components reusable, and removes duplication.

[TypeScript](https://www.typescriptlang.org/) fully supports generics as a way to introduce type-safety into components that accept arguments and return values whose type will be indeterminate until they are consumed later in your code. In this tutorial, you will try out real-world examples of TypeScript generics and explore how they are used in functions, types, [classes](https://www.digitalocean.com/community/tutorials/how-to-use-classes-in-typescript), and [interfaces](https://www.digitalocean.com/community/tutorials/how-to-use-interfaces-in-typescript). You will also use generics to create mapped types and conditional types, which will help you create TypeScript components that have the flexibility to apply to all necessary situations in your code.

**Use Case for Generics**

Let's start with a simple example, where you want to print the value of an argument passed:

function printData(data: number) {

console.log("data: ", data);

}

printData(2);

**printData.ts**

Now, let's suppose you want to make printData a more generic function, where you can pass any type of argument to it like: **number**/ **string**/**boolean**. So, you might think to follow an approach like below:

function printData(data: number | string | boolean) {

console.log("data: ", data);

}

printData(2);

printData("hello");

printData(true);

**printData-new.ts**

But in the future, you might want to print an array of numbers using the same function. In that case the types will increase and it will become cumbersome to maintain all those different types.

This is when **Generics** come into the picture.

## How Generics Work in TS

Generics are like variables – to be precise, type variables – that store the type (for example number, string, boolean) as a value.

So, you can solve the problem we discussed above with generics as shown below:

function printData<T>(data: T) {

console.log("data: ", data);

}

printData(2);

printData("hello");

printData(true);

**printData-generics.ts**

In the above example printData-generics.ts, there is a slight difference in syntax:

1. You use a type variable inside angular brackets after the function name <T>
2. You then assign the type variable to the parameter data: T

Let's explore these differences a bit more.

To use generics, you need to use angular brackets and then specify a type variable inside them. Developers generally use T, X and Y. But it can be anything depending upon your preference.

You can then assign the same variable name as the type to the parameter of the function.

Now, whatever argument you pass to the function, it gets inferred and there's no need to hardcode the type anywhere.

Even if you pass an array of numbers or an object to the printData function, everything will be displayed properly without TS complaining:

function printData<T>(data: T) {

console.log("data: ", data);

}

printData(2);

printData("hello");

printData(true);

printData([1, 2, 3, 4, 5, 6]);

printData([1, 2, 3, "hi"]);

printData({ name: "Ram", rollNo: 1 });

**printData-new.ts**

Let's see another example:

function printData<X,Y>(data1: X, data2: Y) {

console.log("Output is: ", data1, data2);

}

printData("Hello", "World");

printData(123, ["Hi", 123]);

**generics-example2.ts**

In above example, we passed 2 arguments to printData and used X and Y to denote the types for both the parameters. X refers to 1st value of the argument and Y refers to 2nd value of the argument.

Here as well, the types of data1 and data2 are not specified explicitly because TypeScript handles the type inference with the help of generics.

### How to Use Generics with Interfaces

You can even use generics with interfaces. Let's see how that works with the help of a code snippet:

interface UserData<X,Y> {

name: X;

rollNo: Y;

}

const user: UserData<string, number> = {

name: "Ram",

rollNo: 1

}

**interface-generic.ts**

In above snippet, <string, number> are passed to the interface UserData. In this way, UserData becomes a reusable interface in which any data type can be assigned depending upon the use case.

Here in this example, name and rollNo will always be string and number, respectively. But this example was to showcase how you can use generics with interfaces in TS.

**References**

[www.freecodecamp.org](http://www.freecodecamp.org)

## [www.digitalocean.com](http://www.digitalocean.com)