A Closer Look At Generics

Generic Types ("Generics") can be tricky to wrap your head around.

But indeed, we are working with them all the time - one of the most prominent examples is an array.

Consider this example array:

```
let numbers = [1, 2, 3];
```

Here, the type is inferred, but if we would assign it explicitly, we could do it like this:

```
let numbers: number[] = [1, 2, 3];
```

`number[]` is the TypeScript notation for saying "this is an array of numbers".

But actually, `number[]` is just syntactic sugar!

The actual type is `Array`. ALL arrays are of the `Array` type.

BUT: Since an array type really only makes sense if we also describe the type of items in the array, `Array` actually is a generic type.

You could also write the above example liks this:

```
let numbers: Array<number> = [1, 2, 3];
```

Here we have the angle brackets (`<>`) again! But this time NOT to create our own type (as we did it in the previous lecture) but instead to tell TypeScript which actual type should be used for the "generic type placeholder" (`T` in the previous lecture).

Just as shown in the last lecture, TypeScript would be able to infer this as well - we rely on that when we just write:

```
let numbers = [1, 2, 3];
```

But if we want to explicitly set a type, we could do it like this:

```
let numbers: Array<number> = [1, 2, 3];
```

Of course it can be a bit annoying to write this rather long and clunky type, that's why we have this alternative (syntactic sugar) for arrays:

```
let numbers: number[] = [1, 2, 3];
```

If we take the example from the previous lecture, we could've also set the concrete type for our placeholder `T` explicitly:

```
const stringArray = insertAtBeginning<string[]>(['a', 'b', 'c'], 'd');
// or
const stringArray = insertAtBeginning<Array<string>>(['a', 'b', 'c'], 'd');
```

So we can not just use the angle brackets to define a generic type but also to USE a generic type and explicitly set the placeholder type that should be used - sometimes this is required if TypeScript is not able to infer the (correct) type. We'll see this later in this course section!