Creating generic functions

In this lesson, we will learn how to create strongly-typed functions with flexible parameters and return types.

WE'LL COVER THE FOLLOWING

A non-generic function

Below is a function that takes in an array and returns its first element. If the array is empty, `null` is returned.

```
function firstOrNull(array: string[]): string | null {
  return array.length === 0 ? null : array[0];
}
```

This function is strongly-typed, which is excellent; but what if we need to do the same thing for an array of numbers? We can't use the above function because it is restricted for arrays of strings. Wouldn't it be nice if we could pass the array item type into this function? Well, this is what generic functions allow us to.

Generic function syntax

We define the type parameters for a generic function in angle-brackets before the function's parentheses:

```
function someFunc<T1, T2, ...>(...) {
```

```
}
```

Generic parameters can be passed into arrow functions as well:

```
const someVar = <T1, T2, ...>(...) => {
  ...
}
```

The type parameters are placeholders for the real types. We can choose any names we want for these. These type parameter names can then be referenced within the function implementation.

Creating a generic function

Let's turn our attention back to the function, `firstOrNull`, from the start of the lesson. We want to call this function and pass in the array item type as follows:

```
firstOrNull<string>(["Rod", "Jane", "Fred"]);
firstOrNull<number>([1, 2, 3]);
```

So, `firstOrNull` needs a single generic parameter. Let's call this `ItemType`:

```
function firstOrNull<ItemType>( ... )
```

We can reference this for the array type that is passed into it:

```
function firstOrNull<ItemType>(array: ItemType[]) { ... }
```

We can also reference this in the function's return type:

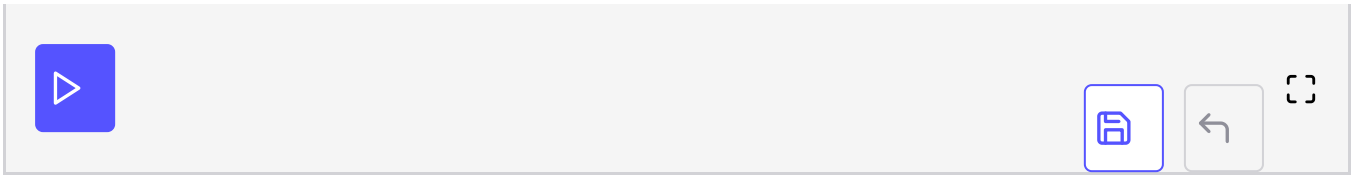
```
function firstOrNull<ItemType>(array: ItemType[]): ItemType | null { ... }
```

So, the final implementation of the generic function is below:

</> TypeScript

```
function firstOrNull<ItemType>(array: ItemType[]): ItemType | null {
  return array.length === 0 ? null : array[0];
}

console.log(firstOrNull<string>(["Rod", "Jane", "Fred"]));
```

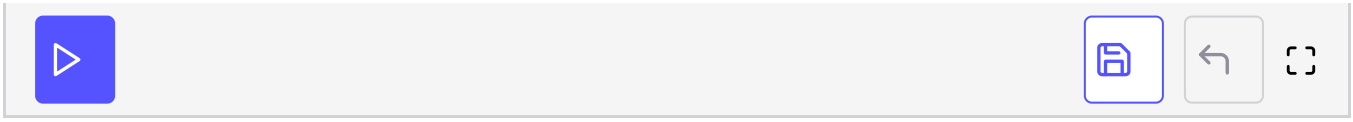


Below is another non-generic function:

</> TypeScript

```
const findFirst = (array, match) =>
  array.indexOf(match) === -1 ? null : array.filter(item => item === match)[0]

console.log(findFirst(["Rod", "Jane", "Fred"], "Jane"));
```



It checks to see if an item is in an array and returns it if so. If the item isn't found, `null` is returned.

Modify this function and turn this into a strongly-typed function with the use of generics.

Show Answer

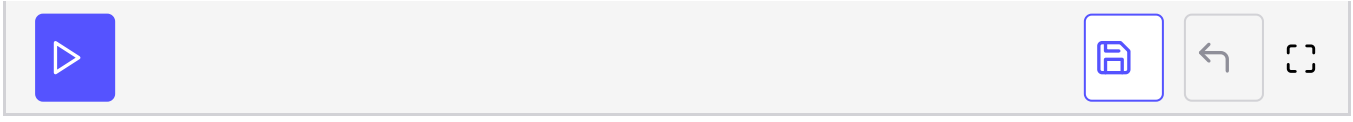
Type inference on generic functions

Let's call our `firstOrNull` function without passing the array item types in the generic parameter:

</> TypeScript

```
function firstOrNull<ItemType>(array: ItemType[]): ItemType | null {
  return array.length === 0 ? null : array[0];
}

const first = firstOrNull([1, 2, 3]);
```



What has TypeScript inferred the type of `first` to be?

Show Answer

So, TypeScript can cleverly infer the return type of a generic function if its parameters are based on the generic type.

Wrap up

Using generic types in a function allows them to be strongly-typed but still be used with different types. The generic types can be referenced within the function's parameters, implementation, and the return type.

More information can be found on generic functions in the TypeScript handbook.

Next up, we'll learn how we can use generics to create reusable interfaces.