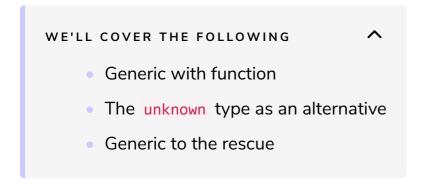
Generic Outside Class

This lesson shows how to use generic outside the concept of classes.



Generic with function

Generic is a concept that is not limited to classes. It can also be used directly on global functions or interfaces. You can have a function that takes generic parameters and also returns a generic type.

```
function countElementInArray<T>(elements: T[]): number {
    return elements.length;
}

function returnFirstElementInArray<T>(elements: T[]): T | undefined {
    if (elements.length > 0) {
        return elements[0];
    }
    return undefined;
}

const arr = [1, 2, 3];
console.log(countElementInArray(arr));
console.log(returnFirstElementInArray(arr));
```

The two functions are examples of what can be accomplished with generic without residing inside a class. Both take an array of type T. The first one uses that type to return a number that is the length of the array. The second returns a particular T element of the array.

The unknown type as an alternative

You may argue that the same code can be accomplished by relying on the unknown type. It is partially true for the former function but is not true for the latter, even if that particular example would transpile and produce the desired output.

```
function countElementInArray(elements: unknown[]): number {
    return elements.length;
}

function returnFirstElementInArray(elements: unknown[]): unknown | undefined {
    if (elements.length > 0) {
        return elements[0];
    }
    return undefined;
}

const arr = [1, 2, 3];
console.log(countElementInArray(arr));
console.log(returnFirstElementInArray(arr));
```

A small modification by using the returned type shows that the unknown type is not suitable. Because it returns unknown instead of the original number type, the result cannot be multiplied by 10 without having TypeScript raise a potential type mismatch. The following code fails for that reason.

 \blacksquare **Note**: The following code is expected to throw an error imes

```
function returnFirstElementInArray(elements: unknown[]): unknown {
    if (elements.length > 0) {
        return elements[0];
    }
    return 0;
}
const arr = [1, 2, 3];
const bigger = returnFirstElementInArray(arr) * 10; // Does not work: unknown cannot be multiconsole.log(bigger);
```

Generic to the rescue

A migration to generic solves the issue and continues to work for a list of number or string or any type passed to the generic function.

```
function returnFirstElementInArray<T>(elements: T[]): T {
    return elements[0];
}
const arr = [1, 2, 3];
const bigger = returnFirstElementInArray(arr) * 10;
console.log(bigger);

const arr2 = ["Test", "is", "good"];
const first = returnFirstElementInArray(arr2);
console.log(arr2.indexOf(first))
```

Now that we have covered the use of generic outside of classes, let's progress to the next lesson, and study generic comparison ahead.