Generic Functions Example - Typing `reduce`

This lesson walks through the implementation of an example generic function. Typing array utility functions is a great exercise that helps you fully understand the complexities of generics.

WE'LL COVER THE FOLLOWING

What is `reduce`?

`Array.reduce` is an extremely versatile function that lets you calculate results from all elements of an array. When calling `reduce` you specify a function that describes how to calculate the result for the next array element, given you already know the result for the previous array elements. It's like a functional version of a `foreach` loop with an additional variable that you change in the loop and return at the end.

```
const sum = [1, 2, 3, 4, 5].reduce((sum, el) => sum + el, 0);
console.log(sum);

interface Person {
    name: string;
    age: number;
}

const persons: Person[] = [
    { name: 'John', age: 30 },
    { name: 'Alice', age: 45 },
];

const ageByPerson = persons.reduce((result, person) => ({
    ...result,
    [person.name]: person.age
}), {});
console.log(ageByPerson);
```

Run the code to see how `reduce` works.

Our `reduce` will be a function that accepts three arguments:

- `array` - the array to be reduced
- `reducer` - a function that accepts an element of the array and the partial result (calculated based on elements processed so far)
- `initialResult` - a value that will be passed to the `reducer` before any array elements have been processed

Typing `reduce`

As you might have guessed, `reduce` is a generic, higher-order function. There will be two type arguments:

- `TElement` - the type of elements of the array
- `TResult` - the type of the calculated result

Let's look at the types of particular function arguments:

- `array` - since `TElement` is the type of array element, the type of the whole array is `TElement[]`
- `initialResult` - the type should be the same as the type of the final result; so it's `TResult`
- `reducer` - it's a function that takes a result (`TResult`) and an array element (`TElement`) and returns an updated result (`TResult`); therefore, it's type is `(result: TResult, el: TElement) => TResult`

Finally, the return type of the whole function is `TResult` (the type of the result we'd like to calculate).

Below you can find a simple implementation of `reduce` along with the usage examples with type arguments provided explicitly.

```
function reduce<TElement, TResult>(
    array: TElement[],
    reducer: (result: TResult, el: TElement) => TResult,
    initialResult: TResult
): TResult {
    let result = initialResult;
    for (const element of array) {
```

```
        result = reducer(result, element);
    }
    return result;
}

const total = reduce<number, number>([1, 2, 3, 4, 5], (sum, el) => sum + el, 0);

const ageByName = reduce<Person, Record<string, number>>(
    persons,
    (result, person) => ({
        ...result,
        [person.name]: person.age
    }),
    {}
);
```

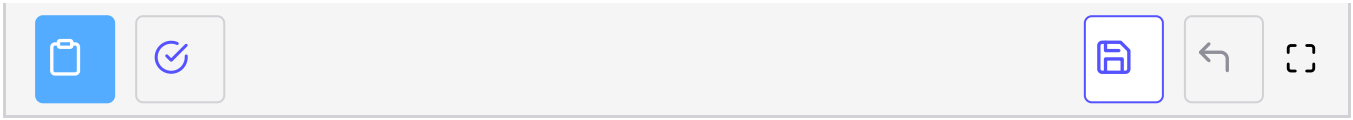
Hover over `total` and `ageByName` to see whether their types are inferred correctly.

Exercise

Add types to the following `zip` function that takes two arrays and returns an array of pairs with elements from both arrays.

*Note that the system can only verify that your code compiles without errors. To check your solution, click on **Show solution** and compare the code.*

```
function zip(array1, array2) {
    const length = Math.min(array1.length, array2.length);
    const result = [];
    for (let i = 0; i < length; i++) {
        result.push([array1[i], array2[i]]);
    }
    return result;
}
```



The next lesson starts discussing generic interfaces.