Generic Functions

This lesson introduces generic functions.

WE'LL COVER THE FOLLOWING

- Motivation
- Let's use `any`!
- Generic functions
- Calling generic functions
- When should you use generic functions?

Motivation

Let's say you are adding types to some JavaScript codebase and you encounter this function:

```
function getNames(persons) {
  const results = [];
  for (let person of persons) {
    results.push(person.name);
  }
  return results;
}

console.log(getNames([
  { name: 'John' },
  { name: 'Alice' },
]));
```

Run the code to see what it does.

Typing this function is straightforward. It accepts an array of person objects as a parameter and returns an array of names (strings). For the person object, you can either create a `Person` interface or use one that you've already created.

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

function getNames(persons: Person[]): string[] {
  /* ... */
}
```

Next, you notice that you don't actually need this function. Instead, you can use the built-in `Array.map` method.

```
interface Person { name: string; }

const persons: Person[] = [
  /* ... */
];

const names = persons.map(person => person.name);
```

Hover over `names` to see the inferred type.

Hmm, but what about types? You check the type of `names` and realize that it has been correctly inferred to `string[]`! How does TypeScript achieve this?

To properly understand this, let's try to type the following implementation of `map` function.

```
function map(items, mappingFunction) {
  const results = [];
  for (let item of items) {
    results.push(mappingFunction(item));
  }
  return results;
}

const persons = [
  { name: 'John' },
  { name: 'Alice' },
];
console.log(map(persons, person => person.name));
```

Run the code to see that it's working correctly.

The main issue with typing `map` is that you don't know anything about the type of the elements of the array it will be called with. What makes `map` so cool is that it works with *any* kind of array!

```
// Works with array of Persons
const names = map(persons, person => person.name);
// Works with array of names too
const uppercaseNames = map(names, name => name.toUpperCase());
// Works even with an array of numbers!
const evenNumbers = map([1, 2, 3, 4, 5], n => n * 2);
```

Let's use `any`!

As a first step, let's try using the `any` type to `map` this function.

```
function map(items: any[], mappingFunction: (item: any) => any): any[] {
  /* ... */
}
```

Let's break this down. `map` has two parameters. The type of the first one (`items`) is `any[]`. We tell the type system that we want `items` to be an array, but we don't care about the type of those items. The type of the second parameter (`mappingFunction`) is a function that takes `any` and returns `any`. Finally, the return type is again `any[]`; an array of *anything*.

Did we gain anything by doing this? We sure did! TypeScript now won't allow us to call `map` with nonsensical arguments:

```
interface Person { name: string; }
const persons: Person[] = [];

function map(items: any[], mappingFunction: (item: any) => any): any[] { return [];}

// Error: 'hello' is not an array
map("hello", (person: Person) => person.name);
// Error: 1000 is not a function
map(persons, 1000);
```

Run the code to see the errors.

Unfortunately, the types we provided are not precise enough. The purpose of TypeScript is to catch possible runtime errors as early as possible, like, at compile-time. However, the following calls won't give any compile errors.

```
interface Person { name: string; age: number;  }
const persons: Person[] = [{ name: 'John', age: 35 }];

function map(items: any[], mappingFunction: (item: any) => any): any[] { return items.map(map

// The second argument is a function that only works on numbers, not on `Person` objects.
// The result doesn't make sense.
console.log(map(persons, n => n + 5));
// We tell TypeScript that `ages` is an array of strings while in fact it will be an array of
// The second line results in a runtime error.
const ages: string[] = map(persons, person => person.age);
ages[0].toLowerCase();
```

Run the code to see the issues that were not captured by our definition of `map`.

How can we improve the typing of `map` so that the above examples would result in a compile-time error? Enter generics.

Generic functions

Making a function generic is (in this case) a way of saying "this function works with any kind of array" and while maintaining type safety at the same time.

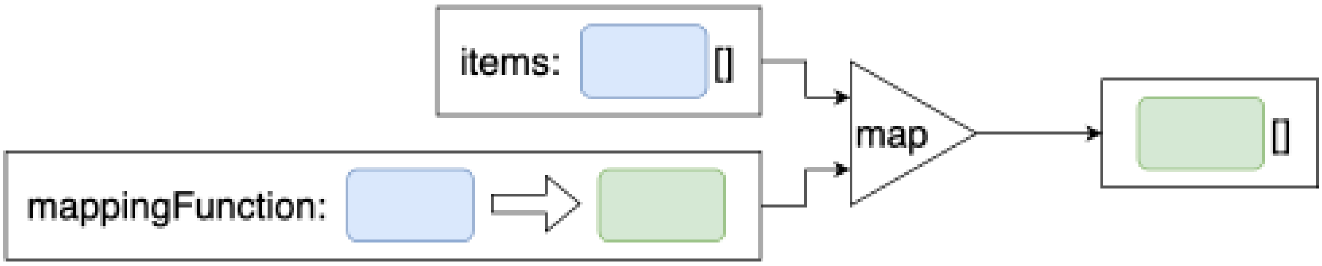
```
function map<TElement, TResult>(
  items: TElement[],
  mappingFunction: (item: TElement) => TResult
): TResult[] {
  /* ... */
}
```

We replaced `any` with `TElement` and `TResult` type parameters. Type parameters are *named* *`any`*. Typing `items` as `TElement[]` still means that it is an array of anything. However, because it's *named*, it lets us establish relationships between types of function parameters and the return type.

Here, we've just expressed the following relationships:

- `mappingFunction` takes anything as a parameter, but it must be *the same type of "anything"* as the type of elements in the `items` array
- `mappingFunction` can return anything, but whatever type it returns, it will be the same as the type of elements of the array returned by the `map` function

The picture below demonstrates these relationships. Shapes of the same color have to be of the same type.



You might have noticed the `<TElement, TResult>` that we added next to `map`. Type parameters have to be declared explicitly using this notation. Otherwise, TypeScript wouldn't know if `TElement` is a type argument or an actual type.

For some reason, it is a common convention to use single-character names for type parameters (with a strong preference for `T`). I'd recommend using full names, especially when you are not very experienced with generics. On the other hand, it's a good idea to prefix type arguments with `T` so that they're easily distinguishable from regular types.

Calling generic functions

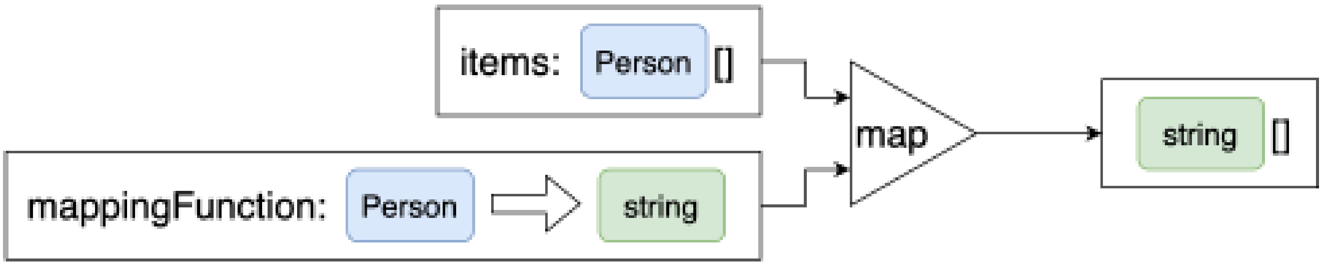
How to call a generic function? As we saw, generic functions have type parameters. These parameters are replaced with actual types when the function is called although technically, it's all happening at compile-time. You can provide the actual types using angle brackets notation.

```
interface Person { name: string; age: number;  }
const persons: Person[] = [{ name: 'John', age: 35 }];

function map<TElement, TResult>(
  items: TElement[],
  mappingFunction: (item: TElement) => TResult
): TResult[] {
```

```
  /* ... */
}

const names = map<Person, string>(persons, person => person.name);
```

Hover over `names` to see that type is inferred correctly.

Imagine that when we provide type arguments, `TElement` and `TResult` are replaced with `Person` and `string`.




```
function map<TElement, TResult>(
  items: TElement[],
  mappingFunction: (item: TElement) => TResult
): TResult[] {
  /* ... */
}

// ...becomes...

function map(
  items: Person[],
  mappingFunction: (item: Person) => string
): string[] {
  /* ... */
}
```

Having to provide type arguments when calling generic functions would be cumbersome. Fortunately, TypeScript can infer them by looking at the types of the arguments passed to the function. Therefore, we end up with the following code.

```
interface Person { name: string; age: number;  }
const persons: Person[] = [{ name: 'John', age: 35 }];

function map<TElement, TResult>(
  items: TElement[],
  mappingFunction: (item: TElement) => TResult
): TResult[] {
```

```
  /* ... */
}


const names = map(persons, person => person.name);
```

Hover over `map` on line 11 to see how its type arguments have been inferred correctly.

Woohoo! It looks exactly like the JavaScript version, except this one is type-safe! Contrary to the first version of `map`, the type of `names` is `string[]` instead of `any[]`. What's more, TypeScript is now capable of throwing a compile error for the following call.

```
interface Person { name: string; age: number;  }
const persons: Person[] = [{ name: 'John', age: 35 }];

function map<TElement, TResult>(
  items: TElement[],
  mappingFunction: (item: TElement) => TResult
): TResult[] {
  return [];
}

// Error! Operator '+' cannot be applied to Person and 5.
map(persons, n => n + 5);
```

Run the code to see the error.

Here is a simplified sequence of steps that leads the compiler to throw an error.

1. The compiler looks at the type of `persons`. It sees `Person[]`.
2. According to the definition of `map`, the type of the first parameter is `TElement[]`. The compiler deduces that `TElement` is `Person`.
3. The compiler looks at the second parameter. It should be a function from `Person` to `TResult`. It doesn't know what `TResult` is yet.
4. It checks the body of the function provided as the second argument. It infers that the type of `n` is `Person`.
5. It notices that you're trying to add `5` to `n`, which is of type `Person`. This doesn't make sense, so it throws an error.

When should you use generic functions?

The good news is that, most likely, you won't be creating generic functions very often. It's much more common to call generic functions than to define them. However, it's still very useful to know how generic functions work, as it can help you understand compiler errors better.

As exemplified by `map`, functions that take arrays as parameters are often generic functions. If you look at the typings for the `lodash` library, you will see that nearly all of them are typed as generic functions. Such functions are only interested in the fact that the argument is an array, they don't care about the type of its elements.

In React framework, Higher-Order Components are generic functions, as they only care about the argument being a component. The type of the component's properties is not important.

In RxJs, most operators are generic functions. They care about the input being `Observable`, but they're not interested in the type of values being emitted by the observable.

The next lesson will walk you through an example of typing a non-trivial generic function.