Function Types

This lesson covers a brief overview of function types to ensure you have a solid foundation before the introduction to generic functions.

WE'LL COVER THE FOLLOWING

- Functions as values
- Callbacks
- Higher-order functions

Functions as values

Every value in TypeScript has a type. Since JavaScript is a functional language, this means that functions are values too. Therefore, they also have types.

The easiest way to check the type of a function is to assign it to a constant and hover over the constant inside the code editor.

```
const add = (x: number, y: number) => x + y;
```

Hover over `add` to see the inferred type.

The type of `add` is inferred to be `(x: number, y: number) => number`. As you can see, it consists of a list of function parameters with their types and function return type, separated by an arrow symbol.

You can also define functions using the classic `function` syntax. From the type perspective, there is no difference between these two versions.

```
function sub(x: number, y: number) {
    return x - y;
}

const operation: (x: number, y: number) => number = sub;
```

Hover over `sub` to see the inferred type.

There is one other way to type functions in TypeScript using the `Function` type. However, using it is highly discouraged. By using it, you declare only that something is a function; you provide no information about its parameter types while the return type is provided.

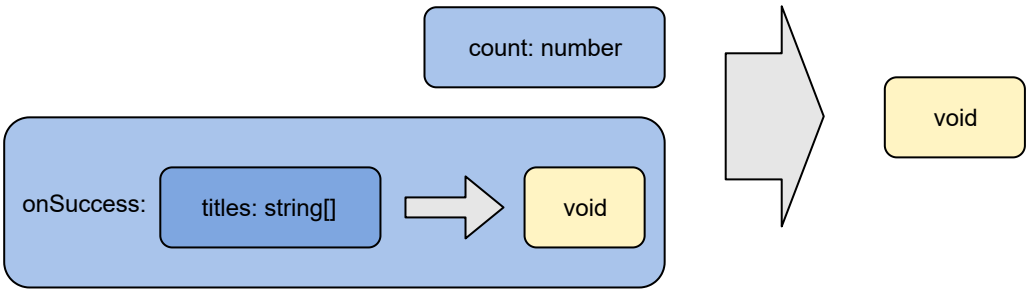
```
// Don't do this!
const operation2: Function = sub;
```

Callbacks

Callbacks are extremely popular in JavaScript and TypeScript. They are a great example of when a function type is necessary. A callback is essentially a function that you pass as an argument to another function. It is invoked when something specific happens, like a backend call returns or when some event occurs. It might be invoked with some arguments with data from the backend or an event object.

To express that a function accepts a callback as a parameter, you need to use a function type.

```
function fetchTitles(
    count: number,
    onSuccess: (titles: string[]) => void
) { /* ... */ }
```



Symbolic representation of the type of `fetchTitles`.

The `fetchTitles` function takes an `onSuccess` callback. The `onSuccess` callback is a function that should accept an array of strings and return nothing as a result. This type annotation carries a lot of information; it says that a list of fetched titles will be provided to this callback and that the return value of the callback will be ignored.

value of the callback will be ignored.

Higher-order functions

A function that accepts callbacks is an example of a *higher-order function*.

A higher-order function is such a function that:

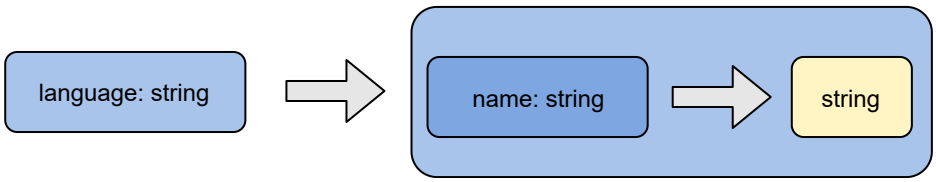
- Has a parameter that has a function type
- OR, it's return type is a function type

Here's another example of a higher-order function that returns a function.

```
const getWelcomeFunction = (language: string) => {
    const hello = language === 'PL' ? 'Witaj' : 'Hello';
    return (name: string) => `${hello}, ${name}`;
}

const welcomeInPl = getWelcomeFunction('PL');
const welcomeInEn = getWelcomeFunction('EN');
```

Hover over `welcomeInPl` and `welcomeInEn` to see the inferred types.



Symbolic representation of the type of `getWelcomeFunction`.

The `getWelcomeFunction` takes a language code string and returns a function that will greet you in a specific language.

Higher-order functions are very common in JavaScript and TypeScript. Many generic functions are also higher-order functions, so it's extremely important to understand this concept.

In the next lesson, we'll look into generic functions.