Types of Function Headers

In this lesson, we will see three function signatures

WE'LL COVER THE FOLLOWING

Function with parameters

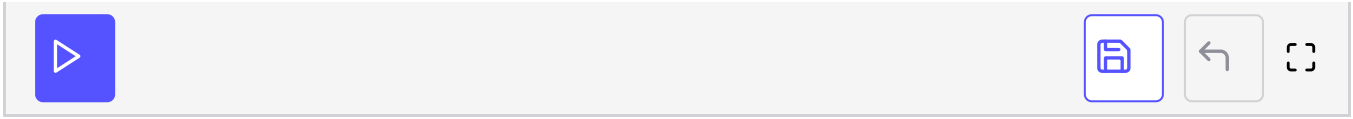
A function's signature can have a parameter that is a function too. These function's parameters can have parameters at their turn but don't require the invoker to use the parameters. A standard scenario is for an optional callback function.

For example, say you have a function that lets you specify an optional error callback named `error` of type `Error`. Not using the argument occurs when notification by the callback is important, but not the details from the argument. The reason is that behind it, JavaScript provides the value. The inspection of the `arguments` object, when not using the argument, corroborates this assertion. However, since you are not using the value, there is no need to clutter the code. TypeScript works in this dynamic fashion because of JavaScript. Since it doesn't generate any potential error, TypeScript doesn't enforce an unnecessary check.

At **line 1** a function with two parameters is defined. The second parameter, named `err`, is a function callback. At **line 2**, the call to the function uses a function without the parameter: it is fine. **Line 3** and **line 4** has the parameter. The two invocation shows that the name of the parameter is not required to be similar to the function.

```
function functParams(p1: string, err: (e: Error) => void): void {}
functParams("test", () => {}); // Parameter e:Error not required


functParams("test", (whatEver: Error) => {}); // Name can be changed
functParams("test", (e: Error) => {});
```



Function with infinite arguments

Functions can take an infinite number of parameters with the rest operator. The rest operator is three dots before the name of the parameter. An array of any type can use the rest operator. Only the last parameter of a function can use the rest operator.

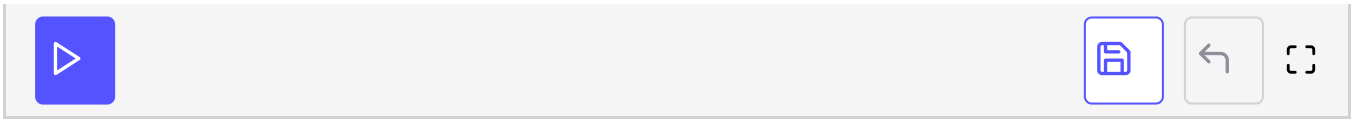
Every parameter beyond the ones provided before the rest operator will fall into this variable. You could accomplish something similar by specifying an optional array which would force the consumer of the function to create an array and set the parameters.

However, with a rest operator parameter, you can, write parameters just like any parameter with a comma between the value passed.

The following example shows in **line 1** a function that takes an infinite number of parameters with the rest operator. **Line 2** shows that adding string with comma-separated is enough. Next, at line 4 the last parameter is an array. The invocation can still take an infinite amount of parameter but must be in an array.

```
function functRest(p1: string, ...remaining: string[]): void {}
functRest("p1", "p2", "p3");

function functBefore(p1: string, remaining: string[]): void {}
functBefore("p1", ["p2", "p3"]);
```



Function with a callback function

A function that has a parameter that is a callback that returns nothing, to be void, can have a function returning a value. The compilation is successful because it provides more than expected. TypeScript does not do anything with

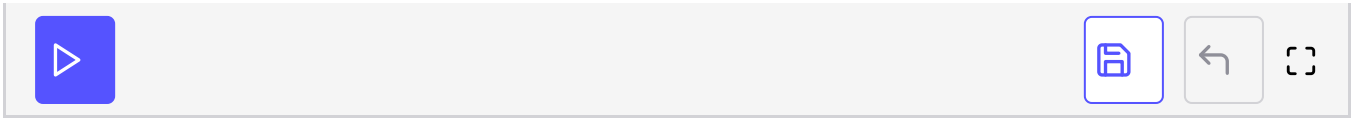
because it provides more than expected. Typescript does not do anything with

the returned value and believes it's void while underneath it will have the value.

For example, if you have a function that accepts a callback function and that you pass a function that returns an object, a number or an array or a string, it will compile. The following example shows that the callback expects a void return. However, the function calls with a callback that returns a string.

As you can see below, at **line 1**, specifying the type `void` to a function doesn't protect the consumer of the callback from reading the return value. **Line 5** calls the function and the callback function returns a `string`.

```
function functReturnVoid(f: () => void): void {
    const c: void = f(); // c is "void" but will store a string
    console.log(c); // Print the string from the return of the function in parameter
}
functReturnVoid(() => {
    return "I am a string, not void!"; // Call back return a string but defined to return voi
});
```



This concept doesn't extend beyond the return type void. For example, if the callback expects to return a `string`, you won't be able to use a function that returns a `number`.