Named and Anonymous Functions

In this lesson, you will see how to name a function, create an anonymous function as well as how to categorize them.

We'll cover the following

- Expressions and declarations
- Function's arguments
- Function's return type
- Inference of void or never
- Short version

Expressions and declarations

JavaScript, and therefore TypeScript, have many ways to define functions. At a high level, functions can be divided into *function declarations* and *function expressions*. A *function declaration* is when a function is defined by not assigning the function to a variable. e.g.

```
function myFunction() ...
```

A function expression is a function assigned to a variable. e.g.

```
const f = function() ...
```

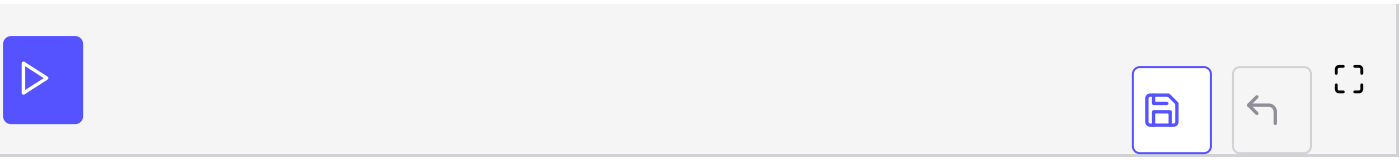
The *function expression* example leads to the creation of *anonymous functions* where a pointer to a function exists but the function is nameless.

```
// Named Function
function functionName1(){}
functionName1(); // Invocation

// Anonymous Function 1
const pointerToFunction1 = function(){}
pointerToFunction1(); // Invocation

// Anonymous Function 2 + Invocation A.K.A. IIFE
```

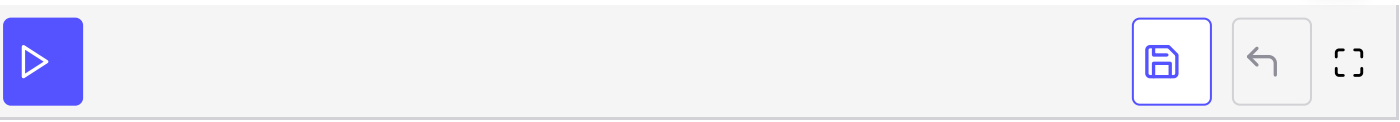
```
(function(){})();
```



Function’s arguments

TypeScript types are used for function arguments. Types at the level of arguments are valid for named functions and anonymous functions. Adding a type to a function’s arguments is similar to how we specify the type of a variable, which is done by using a colon after the variable name.

```
function functionWithParameters(arg1: string, arg2: number){}
```



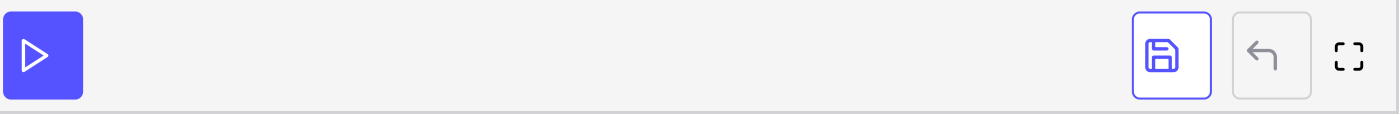
Function’s return type

TypeScript supports the return type. The function’s return type uses the colon after the ending parenthesis of the function but only before the curly braces. The return type works the same here as it does for a variable and can be of a primitive type, an interface, or a union of many types.

The following example has two functions. The first one returns a `boolean` on **line 1**. The second function, on **line 5**, returns a single value which can be a `boolean` or a `number`.

```
function functionWithAReturnType(): boolean {
    return true;
}

function functionWithTwoReturnType(): boolean | number {
    return 1;
}
```



No output, Two functions with different return types.

Inference of void or never

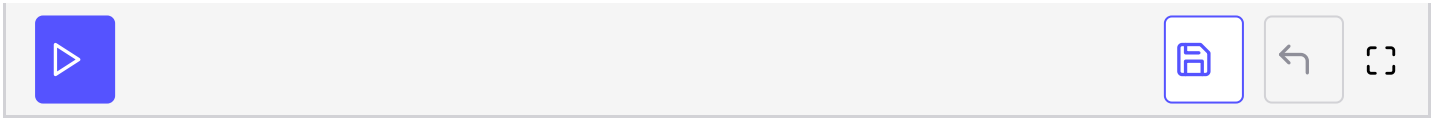
In a previous lesson on `never`, you briefly saw that TypeScript infers the return

type of functions differently in a particular situation. In the following code, the *function declaration* named `c` returns `void` instead of `never`.

```
let a = () => {
    throw new Error("A");
}

let b = function() {
    throw new Error("B");
}

function c() {
    throw new Error("C");
}
```

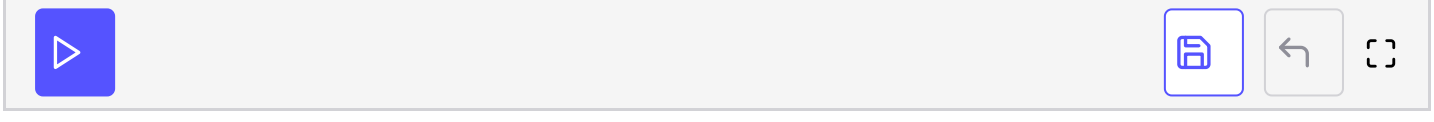


The reason that the *function declaration* is different is because of the pattern that a base class defines a function that is expected to be re-defined by a subclass. The pattern dictates that the body of the base class returns an exception that refers to the body of the function later. If TypeScript infers that the type is `never`, the definition of the body by the base class will not be able to specify a return type, since `never` is not a subtype of any type.

```
class BaseClass {
    defineMeLater(): never { // Infer void, uncomment never to see the consequence
        throw new Error("Define me in a subclass");
    }
}

class SubClass extends BaseClass {
    defineMeLater() {
        console.log("SubClass code");
    }
}

let c = new SubClass();
c.defineMeLater();
```



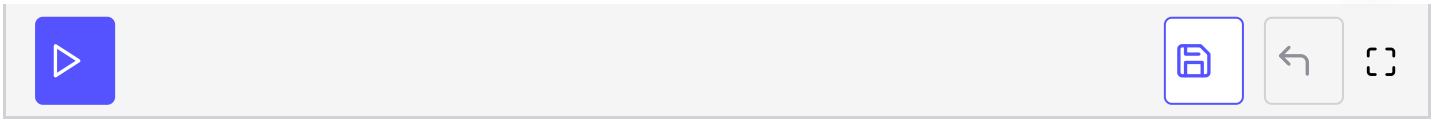
In the code above, changing the `BaseClass` return type of `defineMeLater` causes the `SubClass` to not transpile properly. You can see that by changing (uncommenting) the type of `defineMeLater` to `never`, the code will not compile since throwing an error returns `never` but the `SubClass` returns an implicit `void`.

Short version

When typing an anonymous function, you can opt to use the short version which uses the fat arrow symbol to specify the return type followed by the equal sign to define the function.

The arrow function was introduced with ECMAScript 6 and is another way to define a function. It reduces the number of keystrokes by removing the keyword `function` at the expense of using the equal sign followed by the greater than sign. This syntax is to open the parenthesis, define all parameters, close the parameter, use the fat arrow `=>`, open curly braces, and write the code of the function before finally closing the curly braces. The position after the closing parenthesis (between the colon and the fat arrow with the colon) contains the return type.

```
const fatarrow = (p: number): number => { return p * 100; }
```



Other than brevity, the arrow syntax also carries the reference for the `this` keyword from the parent context where the function is created. This is a huge improvement because before, you had to create a temporary variable that was assigned to `this` before creating the function to use the variable. Without doing any gymnastics, `this` could be different depending on how the function was invoked, which may change easily, causing fickleness.