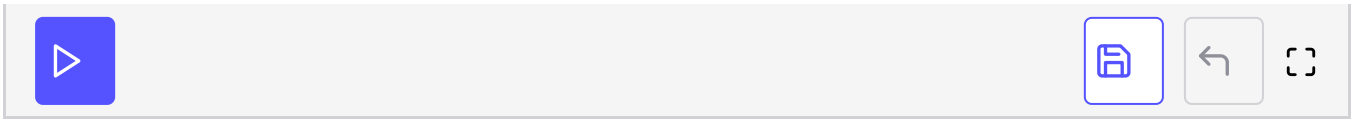
Returning nothing with Void

In this lesson, you will learn how to explicitly mention something that does nothing.

Void means nothing. However, `undefined` can be assigned to `void`. The operation of setting `undefined` to `void` is not useful per se. However, a function that returns nothing should be marked with the `void` reserved keyword.

```
function executeFunctionWithoutReturnType(): void {
    return undefined;
}
let returnType = executeFunctionWithoutReturnType(); // Hover the variable to see "void"
console.log(returnType);
```



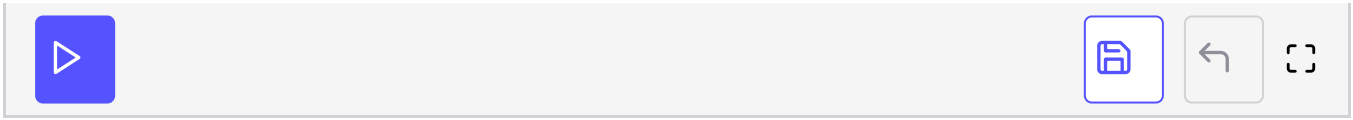
If a function is not explicitly marked with void as the return value, then by default TypeScript will mark the return value as void. This may be problematic, because a programmer may return anything within the function. If it is explicitly set to void, trying to return anything other than undefined will result in a compilation error.

Imagine the following code being the **same function**. The first developer commits code that does not explicitly set the return type. As you will see later, TypeScript infers it to be `number`, not `void`, because of the return statement return `1`. Later, another developer returns a string instead of a number, and now the same function is now returning a string, instead of a number or void. It gets tricky if there are many `return` statements in a big function. Such function will return many types, albeit appearing to return `void` according to the function signature. In reality, it can return anything. The following code shows the evolution of the function returning a number, a string and finally both types.

```
function function1Commit1(){
    return 1;
}


function function1Commit2(){
    return "1";
}

function function1Commit3(){
    if(Math.random()>0.5)
        return "1";
    else
        return 1;
}
```



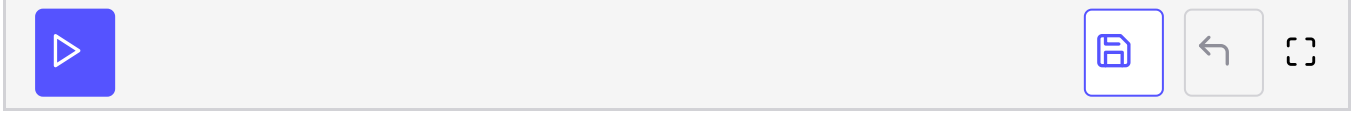
By default, returning `any` allows someone to misuse the result of the function. A misuse arrives when an invocation from the result by using a function that doesn't exist occurs at runtime and raises an error. It is not recommended to use `any` or leave out a type. Instead, `void` should be used to avoid any unnoticed change in behavior.

A pattern to leave a function before its last curly bracket is to use a return statement with no value. In this case, the return is the equivalent of returning `undefined`.

```
function leaveEarly(leaveFast: boolean){
  console.log("Hello");
  if(leaveFast){
    console.log("Quick bye!")
    return;
  }
  console.log("Later good bye");
}

console.log("-- With true --");
let returnValue1 = leaveEarly(true);
console.log("-- With false --");
let returnValue2 = leaveEarly(false);

console.log("-- Types --");
console.log(typeof returnValue1);
console.log(typeof returnValue1);
```

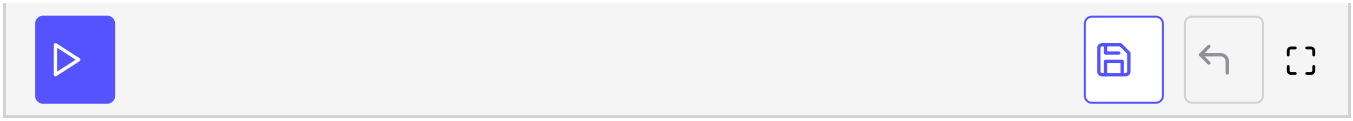


Marking the function as `void` enables the use of an early `return` because it is

similar to return `undefined`; however, for TypeScript, the type of the two variables are narrowed to `void`. You can see by yourself by hovering over the two return variables or in the following code where the variables are set to `void`.

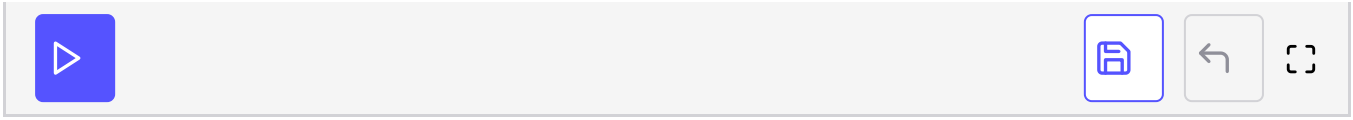
```
function leaveEarly(leaveFast: boolean): void {
  console.log("Hello");
  if(leaveFast){
    console.log("Quick bye!")
    return;
  }
  console.log("Later good bye");
}

console.log("-- With true --");
let voidVar1: void = leaveEarly(true);
console.log("-- With false --");
let voidVar2: void = leaveEarly(false);
```



Finally, as seen in the last example, it is possible to declare a variable as `void` but it is not pragmatic.

```
let variableVoid: void;
variableVoid = undefined;
```



Using `void` is away to ensure that a function does not return a value regardless of its evolution.

Next, we will see that `never` does not mean that a function returns nothing like `void`.