Non-Null Assertion Operator

This lesson introduces the non-null assertion operator and discuss when it is ok to use it.

WE'LL COVER THE FOLLOWING

Overview

So far, we haven't talked about how to fix the kind of errors that are triggered by strict null checks. The reason I haven't mentioned a non-null assertion operator is that you shouldn't be using it unless you have a very good reason to do so.

Non-null assertion operators are simply a way of telling the compiler, *trust me, I'm absolutely sure that this optional value is in fact never empty*. Syntactically, you can express this by postfixing the optional value with a `!` character.

```
interface Person {
    hello(): void;
}

function sayHello(person: Person | undefined) {
    person!.hello(); // no errors!
}
```

Run the code to verify that there are no compile errors despite the code not being safe. `strictNullChecks` flag enabled.

While you might think that a value is always non-empty, you could be wrong.

Or worse, you could be correct at the moment, but it will not be the case in the future, because some other developer will call your function with a `null` or `undefined`.

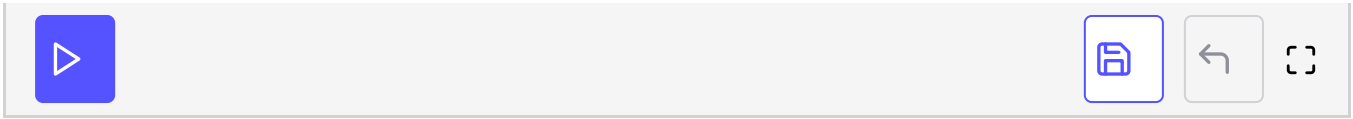
Using with class properties

One possible valid use-case for non-null assertion operators is when you are working with classes and you have the `strictPropertyInitialization` flag enabled.

```
class CounterService {
    counter: number | undefined;

    increase() {
        if (this.counter !== undefined) {
            this.counter += 1;
            this.printCounter();
        }
    }

    private printCounter() {
        console.log(this.counter!.toLocaleString());
    }
}
```



Run the code to see that there are no errors. `strictNullChecks` enabled.

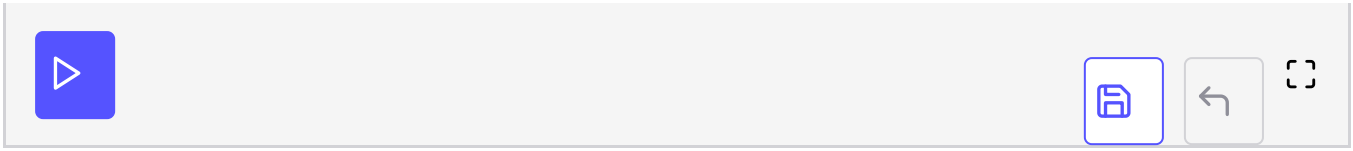
Here, we know that the `printCounter` is only called inside `increase`, where we know that `this.counter` is defined. If we wanted to get rid of this error properly, we would need to introduce an `if` statement.

```
class CounterService {
    counter: number | undefined;

    increase() {
        if (this.counter !== undefined) {
            this.counter += 1;
            this.printCounter();
        }
    }

    private printCounter() {
        if (this.counter !== undefined) {
            console.log(this.counter.toLocaleString());
        }
    }
```

```
}
```



Run the code to see that that there are no errors.

However, this check would always yield true, so one could argue that it adds unnecessary runtime overhead just to satisfy the type system. Therefore, the solution with a non-null assertion operator might be preferred. On the other hand, it could happen that later, another developer calls `printCounter` method in a different context and causes a runtime error.

In summary, you need to be extra careful when using the operator.

Using when enabling `strictNullChecks` on large codebases

Another situation in which it might make sense to use this operator is when you want to enable strict null checks on a large, existing codebase. Enabling the flag in such a case might result in hundreds or thousands of errors. It's a huge effort to go through them all at once. On the other hand, you cannot merge your changes to the master if the whole project doesn't compile successfully. In this case, the non-null assertion operator might come in handy, as it will allow you to quickly suppress the errors that you don't have time to fix properly.

The final lesson in this chapter is an exercise in strict typing.