Debugging Errors: Narrowing Types

This lesson walks through the debugging of a compile error that can be addressed by appropriate type narrowing.

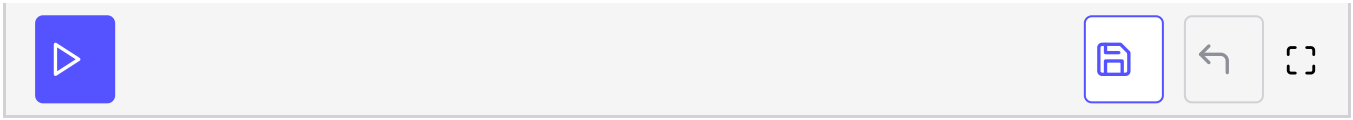
WE'LL COVER THE FOLLOWING

Overview

Another error that we'll look at is an example of a situation when TypeScript is not able to narrow the type of a variable, even though it might be obvious from the perspective of the programmer.

```
interface Person { name: string; employer?: string; };

function getEmployers(persons: Person[]): string[] {
    return persons
        .filter(person => person.employer !== undefined)
        .map(person => person.employer); // Error!
}
```



Run the code to see the error.

The above code will result in the following error message.

```
Type '(string | undefined)[]' is not assignable to type 'string[]'.
  Type 'string | undefined' is not assignable to type 'string'.
    Type 'undefined' is not assignable to type 'string'.ts(2322)
```

Digging deeper

Let's break down the error message.

```
Type '(string | undefined)[]' is not assignable to type 'string[]'.
```

The error message points to the line containing the `return` statement. It says that that the inferred type of the expression passed to the `return` statement `(string | undefined)[]` does not match the expected return type `string[]`. The subsequent lines aim to provide some explanation on why these two types don't match. Both types are array types, so TypeScript compares the type of the array element in both. `string | undefined` is indeed not assignable to `string` because `undefined` is not assignable to `string`. To summarize, we're trying to return an array of possibly undefined string values, where a value of always defined string values is expected.

"But we filter out undefined values," one might say! However, TypeScript has no way of knowing how the predicate provided to `filter` method affects the type of the array. Even though it's obvious that `person.employer` will be defined inside `map`, its type is still `string | undefined`. We can't expect TypeScript to understand the semantics of `Array.filter`, so we need to help it a bit.

Type guard to the rescue

Fortunately, there is a way to fix this in an elegant way. Interestingly, there are two overloads for `Array.filter`:

```
filter(callbackfn: (value: T, index: number, array: T[]) => unknown, thisA
rg?: any): T[];
filter<S extends T>(callbackfn: (value: T, index: number, array: T[]) => v
alue is S, thisArg?: any): S[];
```

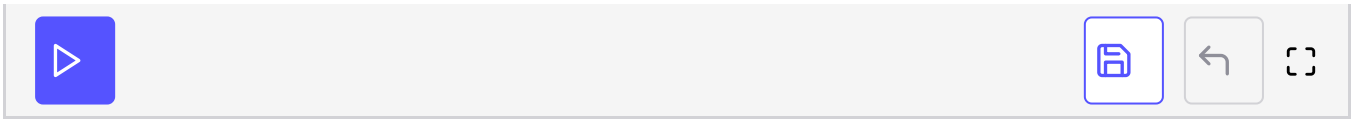
The first one is the more obvious one. You've probably been using it all the time without even knowing about it. The second one is more interesting. It's a generic function with type argument `S`, where `S` is a subtype of `T`. The type of the `callbackFn` parameter is `(value: T, index: number, array: T[]) => value is S`; it's a type guard! All array elements for which `callbackFn` returns `true` can be typed as `S` instead of `T`. Therefore, the return type of the whole method is `S[]`.

All we need to do is provide a type guard version of the predicate, where we

All we need to do is provide a type guard version of the predicate, where we tell TypeScript that if `person.employer !== undefined` then the type of `person` can be narrowed to `Required<Person>`.

```
interface Person { name: string; employer?: string; };

function getEmployers(persons: Person[]): string[] {
    return persons
        .filter((person): person is Required<Person> => person.employer !== undefined)
        .map(person => person.employer);
}
```



Run the code to verify that it compiles successfully.

This example shows how it's sometimes required to provide the type checker a little help to successfully resolve an error.

The next lesson discusses coping with unintuitive error messages.