Debugging Errors: Type Assignability

This lesson advises you on how to approach one of the most common TypeScript compile errors.

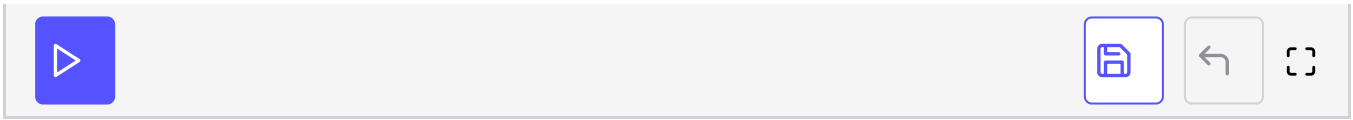
WE'LL COVER THE FOLLOWING

Overview

The most common TypeScript error is related to type assignability. We get them whenever we want to use a value of some type when a different type is required. These are the bread and butter of TypeScript. The main goal of the type checker is to ensure that you provide values with correct types wherever required.

Most of the time, type assignability errors are straightforward. The error message says that the type of the right-hand side of the assignment is not assignable to the type of the left-hand side.

```
// ● Type '5' is not assignable to type 'string'.
const foo: string = 5;
```



Run the code to see the error.

Deeply nested properties

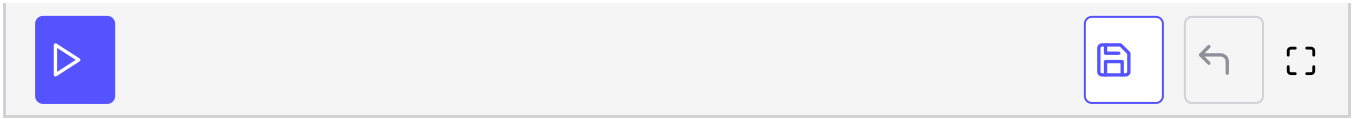
However, these errors can get tricky, especially with type incompatibility of deeply nested properties.

```ts
export interface Foo {
    a: {
        b: {
            c: {
                d: number;
            }
        }
    }
}

function woooot(foo: Foo) {}

const foo = {
    a: {
        b: {
            c: {
                d: 'abc',
            }
        }
    }
};

 // 🔴 Error!
woooot(foo);
```



Run the code to see the error.

Let's look at the error produced by TypeScript for the above piece of code.

```
Argument of type '{ a: { b: { c: { d: string; }; }; }; }' is not assignabl
e to parameter of type 'Foo'.
  Types of property 'a' are incompatible.
    Type '{ b: { c: { d: string; }; }; }' is not assignable to type
'{ b: { c: { d: number; }; }; }'.
      Types of property 'b' are incompatible.
        Type '{ c: { d: string; }; }' is not assignable to type
'{ c: { d: number; }; }'.
          Types of property 'c' are incompatible.
            Type '{ d: string; }' is not assignable to type '{ d: numbe
r; }'.
              Types of property 'd' are incompatible.
                Type 'string' is not assignable to type 'number'.
```

As you can see, the error messages can get quite complex even for such a simple mistake. Let's try to break it down.

```
Argument of type '{ a: { b: { c: { d: string; }; }; }; }' is not assignabl
e to parameter of type 'Foo'
```

The top sentence tells us that the inferred side of the provided function argument `foo` is not assignable to the expected type `Foo`. This is good to know, but why are they not assignable?

```
Types of property 'a' are incompatible.
```

The next sentence explains the reason: types of a property called `a` in both types are not compatible. Good, let's see why.

```
Type '{ b: { c: { d: string; }; }; }' is not assignable to type
'{ b: { c: { d: number; }; }; }'.
```

So, the inferred type of property `a` in `foo` is `{ b: { c: { d: string; }; }; }` while the type of `a` in `Foo` is `{ b: { c: { d: number; }; }; }`. I bet you can already spot the mistake, but imagine these types having more properties. It could be tough to spot the difference with your own eyes. Fortunately, TypeScript helpfully says that `Types of property 'b' are incompatible.`.

The subsequent error messages go deeper and deeper and finally we learn that `Types of property 'd' are incompatible` and `Type 'string' is not assignable to type 'number'`.

Finally, we've found the problem! If you think about it, the last two lines were critical to figuring out what happened. However, sometimes the type might be so complex that you have no idea where to look for the incompatible property. Therefore, sometimes it might be necessary to go through the whole trace of incompatible types.

Simplified error messages

Some good news: starting from TypeScript 3.7, the checker can sometimes simplify and flatten the error message. In particular, the above message would show as below.

```
Argument of type '{ a: { b: { c: { d: string; }; }; }; }' is not assignabl
```

```
e to parameter of type 'Foo'.

  The types of 'a.b.c.d' are incompatible between these types.
    Type 'string' is not assignable to type 'number'.(2345)
```

However, you'll only get the simplified messages in some cases, so it's still important to be able to read deeply nested error messages.

Exercise

Fix the compile error in the below code piece.

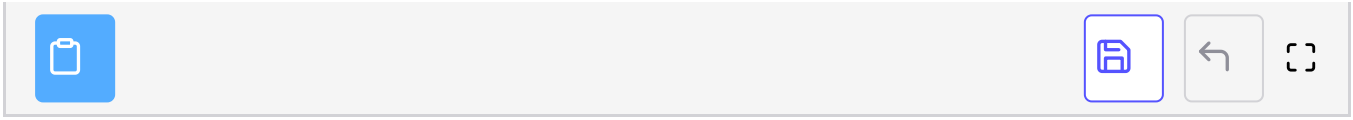
```
interface MyEvent {
    target: {
        id?: string;
    }
}

interface Props {
    width: number;
    height: number;
    callbacks: {
        onSuccess: (event: MyEvent) => void;
    }
}

const component: (props: Props) => string =
    (props) => `<div width="${props.width}"></div>`;

const props = {
    width: 100,
    height: 200,
    callbacks: {
        onSuccess: (event: { target: { id: string } }) => { console.log(event.target.id); }
    }
};

component(props);
```



The next lessons goes through an example of fixing another class of errors.