React and TypeScript: `useState` Hook

This lesson is a detailed analysis of typing the `useState` hook in TypeScript.

WE'LL COVER THE FOLLOWING

Overview

React hooks are a recent addition to React that make function components have almost the same capabilities as class components. Most of the time, using React hooks in TypeScript is straightforward.

However, there are some situations when a deeper understanding of a hook's type might prove useful. In this lesson, we're going to focus on the `useState` hook.

I'm going to assume that you have a basic understanding of this hook. If this is not the case, please read this first.

Reading the types

First of all, let's take a look at the type signature of `useState`. You'll see how much information you can extract solely from types, without looking at the docs or the implementation.

Overloads

```
function useState<S = undefined>(): [S | undefined, Dispatch<SetStateActio
n<S | undefined>>];
function useState<S>(initialState: S | (() => S)): [S, Dispatch<SetStateAc
tion<S>>];
```

As you can see, there are two *versions* of the `useState` function. TypeScript lets you define multiple type signatures for a function as it is often the case in JavaScript that a function supports different types of parameters. Multiple type signatures for a single function are called **overloads**.

Both overloads are generic functions. The type parameter `S` represents the type of the piece of state stored by the hook. The type argument in the second overload can be inferred from `initialState`. However, in the first overload, it defaults to `undefined` unless the type argument is explicitly provided. **If you don't pass the initial state to `useState`, you should provide the type argument explicitly.**

`useState` parameters

The first overload doesn't take any parameters; it's used when you call `useState` without providing any initial state.

The second overload accepts `initialState` as a parameter. Its type is a union of `S` and `() => S`. Why would you pass a function that returns the initial state instead of passing the initial state directly? Computing the initial state can be expensive. It's only needed when the component is mounted. However, in a function component it would be calculated on every render. **Therefore, you have an option to pass a function that calculates initial state, meaning expensive computation will only be executed once, not on every render.**

`useState` return type

Let's move to the return type. It's a **tuple** in both cases. A tuple is like an array that has a specific length and contains elements with specific types.

For the second overload, the return type is `[S, Dispatch<SetStateAction<S>>]`. The first element of the tuple has type `S`, the type of the piece of state. It will contain the value retrieved from the component's state.

The second element's type is `Dispatch<SetStateAction<S>>`. `Dispatch<A>` is simply defined as `(value: A) => void`, a function that takes a value and doesn't return anything. `SetStateAction<S>` is defined as `S | ((prevState: S) => S)`. Therefore, the type of `Dispatch<SetStateAction<S>>` is actually `(value: S | ((prevState: S) => S)) => void`. It is a function that takes either an updated version of the piece of state OR a function that produces the updated version based on the previous version. In both cases, we can deduce that the second element of the tuple returned by `setState` is a function that we can call to update the component's state.

The return type of the first overload is the same, but here instead of `S`, `S | undefined` is used anywhere. If we don't provide the initial state it will store `undefined` initially. It means that `undefined` has to be included in the type of the piece of state stored by the hook.

Usage examples

Most of the time you don't need to bother with providing type arguments to `useState`, the compiler will infer the correct type for you. However, in some situations, type inference might not be enough.

Empty initial state

The first type of situation is when you don't want to provide the initial state to `useState`.

As we saw in the type definition, the type argument `S` for the parameterless defaults as `undefined`. Therefore, the type of `pill` should be inferred to `undefined`. However, due to a design limitation in TypeScript, it's actually inferred to `any`.

Similarly, `setPill`'s type is inferred to `React.Dispatch<any>`. This is really bad, as nothing would stop us from calling it with an invalid argument: `setPill({ hello: 5 })`.

```
export const PillSelector: React.FunctionComponent = () => {
    const [pill, setPill] = useState();
    return (
        <div>
            <button onClick={() => setPill('red')}>Red pill</button>
            <button onClick={() => setPill('blue')}>Blue pill</button>
            <span>You chose {pill} pill!</span>
        </div>

    );
}
```

In order to fix this issue, we need to pass a type argument to `setState`. We treat `pill` as text in JSX, so our first bet could be `string`. However, let's be more precise and limit the type to only allow values that we expect.

```
const [pill, setPill] = useState<'red' | 'blue'>();
```

Note that the inferred type of `pill` is now `"red" | "blue" | undefined` because this piece of state is initially empty. With `strictNullChecks` enabled, TypeScript wouldn't let us call anything on `pill`:

```
// 🛑 Object is possibly 'undefined'.ts(2532)
<span>You chose {pill.toUpperCase()} pill!</span>
```

...unless we check the value first:

```
// ✅ No errors!
{pill && <span>You chose {pill.toUpperCase()} pill!</span>}
```

Clearable state

Another situation where you would provide a type argument to `useState` is when the initial state is defined, but you want to be able to **clear** the state later.

```
export const PillSelector: React.FunctionComponent = () => {
    const [pill, setPill] = useState('blue');
    return (<div>
        <button onClick={() => setPill('red')}>Red pill</button>
        <button onClick={() => setPill('blue')}>Blue pill</button>
        // 🛑 Argument of type 'undefined' is not assignable
        // to parameter of type 'SetStateAction<string>'.
        <button onClick={() => setPill(undefined)}>Reset</button>
        {pill && <span>You chose {pill.toUpperCase()} pill!</span>}
    </div>);
}
```

Since the initial state is passed to `useState`, the type of `pill` gets inferred to `string`. Therefore, when you try to pass `undefined` to it, TypeScript will error.

You can fix the problem by providing the type argument.

```
const [pill, setPill] = useState<'blue' | 'red' | undefined>('blue');
```

The next lesson talks about a more complex typing scenario in React.