Conditional Types Example - React Component Props

This lesson shows a real-life usage of conditional types in React.

WE'LL COVER THE FOLLOWING

- Detecting a React component
- Extracting component props type with `infer`

Detecting a React component

How can we take advantage of conditional types and use them to extract the types of React component properties? Let's create a conditional type that checks whether a given type is a React component.

```
type IsReactComponent<T> =
  T extends React.ComponentType<any> ? "yes" : "no";
```

`IsReactComponent` takes a type argument `T` and checks whether it extends `React.ComponentType`. If yes, it returns a `"yes"` string literal type. Otherwise, it returns a `"no"` string literal type.

Since `React.ComponentType` is a generic type, we had to specify the type parameter for `React.Component`, so we provided `any`. However, TypeScript lets us do something even cooler; we can use the `infer` keyword instead.

```
type IsReactComponent<T> =
  T extends React.ComponentType<infer P> ? "yes" : "no";
```

Extracting component props type with `infer`

`infer` creates a new type variable, `P`, that will store the type parameter of `T` if it indeed extends `React.Component`. In our case, it will be exactly what we're looking for, the type of `props`! So, instead of returning a `"yes"` literal type, it simply returns `P`.

```
type IsReactComponent<T> =
  T extends React.ComponentType<infer P> ? P : "no";
```

Now, we can assume that this type will only be used with actual React components. We'd like the compilation to fail otherwise. Instead of returning `"no"`, we want it to return the `never` type. It's a special type that is intended exactly for these situations. We return `never` when we don't want something to happen. If a variable has a `never` type, then nothing can be assigned to it.

```
type PropsType<C> =
  C extends React.ComponentType<infer P> ? P : never;

class Article extends React.Component<{ content: string }> {
  render = () => this.props.content; // This would normally be some JSX.
}

type ArticleProps = PropsType<typeof Article>; // { content: string; }
```

Hover over `ArticleProps` to see the inferred type.

And that's it! `PropsType` takes a type argument, `C`, and matches it against `React.ComponentType<infer P>`. If there is a match, the type argument of `React.ComponentType` is stored inside `P` and `P` is returned. Otherwise, `never` is returned.

Let's break down the above example:

1. `Article` extends `React.Component<{ content: string }>`, which extends `React.ComponentType<{ content: string }>`.
2. `React.ComponentType<{ content: string }>` matches `React.ComponentType<infer P>`.
3. `{ content: string }` is stored inside `P` and returned.
4. `ArticleProps` is inferred to `{ content: string }`.

Note that we have to pass the `typeof Article` instead of `Article` to `PropsType`. `Article` type represents the shape of instances of the `Article` class. On the other hand, `typeof Article` represents the constructor function of the `Article` class. We have to use the latter because of how `React.ComponentType` is defined.

The next lesson introduces another advanced type concept, mapped types.