Higher-order components

In a previous lesson, you learned about Higher-order components (HOC) as a pattern to abstract shared behavior, as well as a basic example of an implementation.

Let's dive deeper to illustrate some of the best practices and caveats regarding HOCs.

These include never mutating a component inside a HOC, passing unrelated props to your wrapped component, and maximizing composability by leveraging the `Component => Component` signature.

Don't mutate the original component

One of the possible temptations is to modify the component that is provided as an argument, or in other words, mutate it. That's because JavaScript allows you to perform such operations, and in some cases, it seems the most straightforward and quickest path. Remember that React promotes immutability in all scenarios. So instead, use composition and turn the HOC into a pure function that does not alter the argument it receives, always returning a new component.

```
const HOC = (WrappedComponent) => {
  // Don't do this and mutate the original component
  WrappedComponent = () => {

  };
 …
}
```

Pass unrelated props through to the Wrapped Component

HOC adds features to a component. In other words, it enhances it. That's why they shouldn't drastically alter their original contract. Instead, the component returned from a HOC is expected to have a similar interface to the wrapped component.

HOCs should spread and pass through all the props that are unrelated to their specific concern, helping ensure that HOCs are as flexible and reusable as possible, as demonstrated in the example below:

```
const withMousePosition = (WrappedComponent) => {
  const injectedProp = {mousePosition: {x: 10, y: 10}};

  return (originalProps) => {
    return <WrappedComponent injectedProp={injectedProp} {...originalProps} />;
  };
};
```

Maximize composability

So far, you have learned that the primary signature of a HOC is a function that accepts a React component and returns a new component.

Sometimes, HOCs can accept additional arguments that act as extra configuration determining the type of enhancement the component receives.

```
const EnhancedComponent = HOC(WrappedComponent, config)
```

The most common signature for HOCs uses a functional programming pattern called "currying" to maximize function composition. This signature is used extensively in React libraries, such as React Redux, which is a popular library for managing state in React applications.

```
const EnhancedComponent = connect(selector, actions)(WrappedComponent);
```

This syntax may seem strange initially, but if you break down what's happening separately, it would be easier to understand.

```
const HOC = connect(selector, actions);
const EnhancedComponent = HOC(WrappedComponent);
```

`connect` is a function that returns a higher-order component, presenting a valuable property for composing several HOCs together.

Single-argument HOCs like the ones you have explored so far, or the one returned by the connect function has the signature `Component => Component`. It turns out that functions whose output type is the same as its input type are really easy to compose together.

```
const enhance = compose(
  // These are both single-argument HOCs
  withMousePosition,
  withURLLocation,
  connect(selector)
);

// Enhance is a HOC
const EnhancedComponent = enhance(WrappedComponent);
```

Many third-party libraries already provide an implementation of the compose utility function, like lodash, Redux, and Ramda. Its signature is as follows:

`compose(f, g, h)` is the same as `(...args) => f(g(h(...args)))`

Caveats

Higher-order components come with a few caveats that aren't immediately obvious.

1. Don't use HOCs inside other components: always create your enhanced components outside any component scope. Otherwise, if you do so inside the body of other components and a re-render occurs, the enhanced component will be different. That forces React to remount it instead of just updating it. As a result, the component and its children would lose their previous state.

```
const Component = (props) => {
  // This is wrong. Never do this
  const EnhancedComponent = HOC(WrappedComponent);
  return <EnhancedComponent />;
};

// This is the correct way
const EnhancedComponent = HOC(WrappedComponent);
const Component = (props) => {
  return <EnhancedComponent />;
};
```

2. Refs aren't passed through: since React `refs` are not `props`, they are handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component. To solve this, you can use the React.forwardRef API. You can learn more about this API and its use cases in the additional resources section from this lesson.

Conclusion

And in summary, you have examined higher-order components in more detail. The main takeaways are never mutating a component inside a HOC and passing unrelated props to your wrapped component.

You also learned how to maximize composability by leveraging the `Component => Component` signature and addressed some caveats about HOC.