Provider and Consumer Components

In this lesson, we'll explore provider and consumer components.

WE'LL COVER THE FOLLOWING

- `Provider` and `Consumer` Components
- `Provider` Component
- `Consumer` Component

`Provider` and `Consumer` Components

Well, every context object comes with a `Provider` and `Consumer` component.

The `Provider` component **provides** the value saved in the context object to its children, while the `Consumer` component **consumes** the values from within any child component.

I know that was a mouthful, so let's break it apart slowly.

In the Benny example, we can go ahead and destructure the `BennyPositionContext` to retrieve the `Provider` and `Consumer` components.

```
const BennyPositionContext = createContext({
        x: 50,
        y: 50
})
// get provider and consumer
const { Provider, Consumer } = BennyPositionContext
```

Since `Provider` provides values saved in the context object to its **children**, we could wrap a tree of components with the `Provider` component as shown below:

```
<Provider>
   // the root component for the Benny app.
   <Root />
```

```
  <Root />
</Provider>
```

Now, any child component within the `Root` component will have access to the default values stored in the context object.

Consider the following tree of components for the Benny app.

```
<Provider>
  Root = () => (
    <Scene>
      <Benny />
    </Scene>
  )
</Provider>
```

`Provider` Component

`Scene` and `Benny` are children of the `Root` component and represent the game scene and the Benny character respectively.

In this example, the `Scene`, or even the deeper-nested `Benny` component, will have access to the value provided by the `Provider` component.

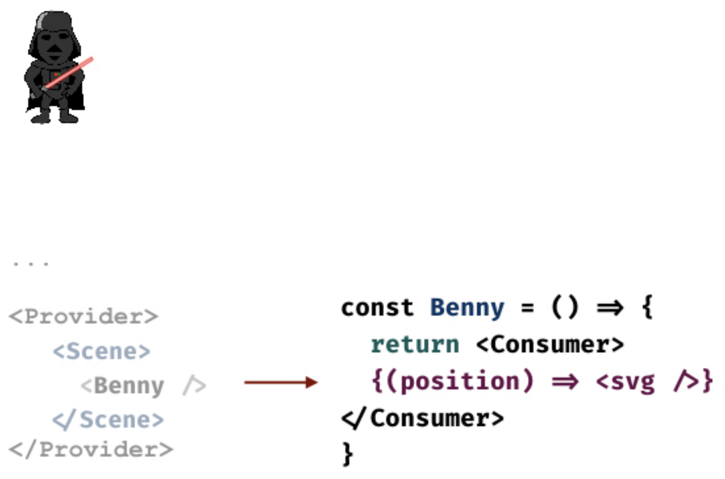
It is worth mentioning that a `Provider` also takes in a `value` prop.

This `value` prop is useful if you want to provide value other than the initial value passed in at the context object creation time via `createContext(initialStateValue).`

Here's an example where a new set of values are passed into the `Provider` component:

```
<Provider value={x: 100, y: 150}>
   <Scene>
     <Benny />
   </Scene>
</Provider>
```

Now that we have values provided by the `Provider` component, how can a nested component such as `Benny` consume this value?



```
...

<Provider>
   <Scene>
     <Benny />
   </Scene>
</Provider>
```

```
const Benny = () => {
  return <Consumer>
  {(position) => <svg />}
</Consumer>
}
```

`Consumer` Component

The simple answer is by using the `Consumer` component.

Consider the `Benny` component being a simple component that renders some SVG.

```
const Benny = () => {
  return <svg />
}
```

Now, within `Benny`, we can go ahead and use the `Consumer` component like this:

```
const Benny = () => {
  return <Consumer>
  {(position) => <svg />}
</Consumer>
}
```

Okay, what's going on here?

Okay, what's going on here?

The `Consumer` component exposes a render prop API, meaning the children are a function. This function is then passed arguments corresponding to the values saved in the context object. In this case, the `position` object with the `x` and `y` coordinate values.

It is worth noting that whenever the value from a `Provider` component changes, the associated `Consumer` component and the children will be re-rendered to keep the value(s) consumed in sync.

Also, a `Consumer` will receive values from the closest `Provider` above it in the tree.

Consider the situation below:

```
// create context object
const BennyPositionContext = createContext({
        x: 50,
        y: 50
})
// get provider and consumer
const { Provider, Consumer } = BennyPositionContext
// wrap Root component in a Provider
<Provider>
        <Root />
</Provider>
// in Benny, within Root.
const Benny = () => (
        <Provider value={x: 100, y: 100}>
                // do whatever
        </Provider>
)
```

With a new provider component introduced in `Benny`, any `Consumer` within `Benny` will receive the value `{x: 100, y: 100}` NOT the initial value of `{x: 50, y: 50}`.

This is a contrived illustrated example, but it helps solidify the foundations of using the Context API.

Quick Quiz!

Q

Why we use `Provider` component?

COMPLETED 0%

1 of 1

Now that we understand the necessary building blocks for using the Context API, let’s build an application utilizing all we’ve learned so far.