React Information Flow

Start Assignment

Due No Due Date **Points** 1 **Submitting** a website url

FORK (https://github.com/learn-co-curriculum/react-hooks-information-flow-code-along/fork)
(https://github.com/learn-co-curriculum/react-hooks-information-flow-code-along)
(https://github.com/learn-co-curriculum/react-hooks-information-flow-code-along/issues/new)

Learning Goals

- Understand the flow of information between components with props
- Use callback functions as props to update state in a parent component

Introduction

In this lesson, we'll explore how to pass callback functions as props in order to change state in a parent component.

How Does Information Flow Between Components?

We already know how to use props to pass information *down* from parent to child. But how would we do the reverse? How might we have a **child** component send data *up* to its **parent** component?

In order to propagate information in the opposite direction, we can send a **callback function as props** from the parent component to its child.

This allows the callback to be *owned* by a different component than the one invoking it. Once invoked, the callback can send data to or change state in the parent component that *owns* it, instead of the child component that *invoked* it.

Getting Started

Assuming you've pulled down the starter code and ran `npm install` and `npm start` , you should see a few rectangles in your browser. The large outer rectangle will be a random color every time you refresh the page, but the two smaller rectangles inside will always have a white background.

Take a moment to familiarize yourself with the code base. We have a simple application that renders a single `Parent` component and two `Child` components. The component hierarchy is as follows:

```
App
└───Parent
    ├───Child
    └───Child
```

Help

Deliverables Part 1

- When either `Child` component is clicked, the `Parent` component should change color.

`src/randomColorGenerator.js` has a helper function `getRandomColor()` implemented for you that generates a random color.

Changing the color of Parent

The `Parent` component has a state variable called `color` that is initially set to a random color. To update state, we'll create a simple `handleChangeColor` function:

```
function Parent() {
  const randomColor = getRandomColor();
  const [color, setColor] = useState(randomColor); // initial value for color stat

  function handleChangeColor() {
    const newRandomColor = getRandomColor();
    setColor(newRandomColor); // update color state to a new value
  }

  return (
    <div className="parent" style={{ backgroundColor: color }}>
      <Child />
      <Child />
    </div>
  );
}
```

But we are going to want to run this `handleChangeColor()` function when either `Child` component is clicked. So we are going to pass this state changing function *as a prop* to both `Child` components.

```
return (
  <div className="parent" style={{ backgroundColor: color }}>
    <Child onChangeColor={handleChangeColor} />
    <Child onChangeColor={handleChangeColor} />
  </div>
);
```

Now, `Child` will have a prop called `onChangeColor` that is a *function*. Specifically, it is the same function object as our `Parent` 's `handleChangeColor` function. Want to see for yourself? Put a `console.log` inside the `Child` component.

```
function Child({ onChangeColor }) {
  console.log(onChangeColor);
```

Help

```
    return <div className="child" style={{ backgroundColor: "#FFF" }} />;
  }
```

We can now use this `onChangeColor` prop as an event handler:

```
console.log(onChangeColor);
return (
  <div
    onClick={onChangeColor}
    className="child"
    style={{ backgroundColor: "#FFF" }}
  />
);
```

And ta-da! Now, if you go to the app, clicking on *either* of the white rectangle `Child` components will cause the `Parent` component to change color.

Let's walk though those steps:

- When the `div` in the `Child` component is clicked, it will use the `onChangeColor` variable to determine what function to run
- `onChangeColor` is a prop that is passed down from the `Parent` component, which references the `handleChangeColor` function
- The `handleChangeColor` function is the function that will actually run when the `div` is clicked, and will update state in the `Parent` component

Now, let's add one more feature!

Deliverables Part 2

- When either `Child` component is clicked, it should change its own background color to a random color, and the other `Child` component should change to *that same* color.

Now, we could put some state in our `Child` component to keep track of its color. However:

- **Sibling components cannot pass data to each other directly**
- **Data can only flow up and down between parent and child**

So if we update the color of one `Child` component, we have no way to pass that data to the *other* `Child` component.

The solution is to store the color of the `Child` in the state of the `Parent` component. Then, we let the `Parent` component handle the passing of that data to each of its children components. We'll start by creating a variable to keep track of the color of the `Child` components using state:

```
function Parent() {
  const randomColor = getRandomColor();
  const [color, setColor] = useState(randomColor);
  const [childrenColor, setChildrenColor] = useState("#FFF");
```

Help

```
  // ...
}
```

Since the data that represents the color of the two `Child` components lives in `Parent`, we should pass that data down as props:

```
return (
  <div className="parent" style={{ backgroundColor: color }}>
    <Child color={childrenColor} onChangeColor={handleChangeColor} />
    <Child color={childrenColor} onChangeColor={handleChangeColor} />
  </div>
);
```

Now let's actually use that props data in the `Child` component:

```
function Child({ onChangeColor, color }) {
  return (
    <div
      onClick={onChangeColor}
      className="child"
      style={{ backgroundColor: color }}
    />
  );
}
```

Lastly, we have to update the `handleChangeColor()` function in `Parent` to change not just the `color` state, but also the `childrenColor`. To practice sending data *back* to the parent, let's change our `handleChangeColor` to take in an argument of `newChildColor` and then use that variable to update the state of the `Child` component:

```
function handleChangeColor(newChildColor) {
  const newRandomColor = getRandomColor();
  setColor(newRandomColor);
  setChildrenColor(newChildColor);
}
```

Now that the function takes in an argument, we can create a new function in our `Child` component that invokes `onChangeColor` and passes in a random color as the argument; we also need to update the component's `onClick` callback to be that new function:

```
function Child({ onChangeColor, color }) {
  function handleClick() {
    const newColor = getRandomColor();
    onChangeColor(newColor);
  }
```

Help

```
  return (
    <div
      onClick={handleClick}
      className="child"
      style={{ backgroundColor: color }}
    />
  );
}
```

Wow! Check out the finished product in the browser! When either `Child` component is clicked, the `Parent` changes to a random color, and both `Child` components change to a different random color.

Conclusion

For information to propagate **down** the component tree, parents pass `props` to their children.

For information to propagate **up** the component tree, we must invoke **callbacks** that were passed from parents to children as `props` .

Components of the same level (sibling components) cannot communicate directly! We can only communicate up and down the component tree. So if multiple components need to share the same information, that state should live in the parent component (or a more general ancestor).

Resources

- Lifting State Up (https://reactjs.org/docs/lifting-state-up.html)

Help