React Reducers

Sometimes, when there are many state variables and their associated logic, the code for a component can get complex and messy. When you find it more difficult to follow the state logic due to many event handlers in the component function, you may find it useful to use **reducers**.

Reducer functions allow you to separate the state logic from the component code. This results in simpler component code by decoupling the component code from the state logic. All of your state logic can then be moved into its own code file. (Note that, from the React documentation, the name "reducer" comes from the array reduce() operation.)

Reducer function structure:

```
export default function myReducer(state, action) {
  switch (action.type) {
    case "actionName1":
      return { <some_result> };
    default:
      return Error("Unknown action");
  }
}
```

A **reducer** is a function you define to determine what should happen to the state whenever an **action** is **dispatched**. In other words, the reducer function receives an action and updates the state depending on what action occurred. When an action is triggered, it is being **dispatched**.

In the code above, the reducer checks an action object's **type** property to determine which logic to execute.

An **action** is an object with a **type** field or property. This "type" is a string token (i.e. a string with no spaces) which describes what happened. The types allowed are defined by you, but just make sure they're descriptive. **The type field is mandatory.**

An action can also have a **payload** field (optional). The payload is the data that is passed. Depending on the what should happen when an action is dispatched, you may not always need a payload.

To dispatch an action, you run `dispatch({<action_object>})`.

In the component file, you can use the useReducer() hook function to use the reducer.

```
import {useReducer} from "react";
…
//inside component function
const [state, dispatch] = useReducer(<reducer_name>, <initial_state>);
```

We can create a very simple counter app to demonstrate this. The app will have two buttons: a click button (to increment each time it's clicked) and a reset button (reset the count).

/components/ClickCounter/clickCounterReducer.jsx

```
export default function clickCounterReducer(counter, action) {
  switch (action.type) {
    case "increment":
      return { count: counter.count + 1 };
    case "reset":
      return { count: 0 };
    default:
      throw Error("Unknown action");
  }
}
```

- The state variable is named counter. It has a count property which stores how many times the counter button has been clicked (see code below).

/components/ClickCounter/index.jsx (the ClickCounter component code)

```
import {useReducer} from "react";
import clickCounterReducer from "./clickCounterReducer";

export default function ClickCounter() {
  const [counter, dispatch] = useReducer(clickCounterReducer, { count: 0 });
  return (
    <div id="click-counter">
      <div>Times clicked: {counter.count}</div>
      <div id="counter-buttons">
        <button
          onClick={() => { dispatch({ type: "increment"})}}
        >Click me</button>
        <button
          onClick={() => { dispatch({ type: "reset" })}}
        >Reset counter</button>
      </div>
    </div>
  );
}
```

- **clickCounterReducer** is the used by the useReducer() hook function. The second parameter for useReducer() is an initial value (in this case, the click count was set to 0).
- When one of the buttons is clicked, an action object is dispatched using the dispatch() function. In this case, all of the actions have no payload (no data to send).
- In the reducer function, an "increment" type of action will add 1 to the counter state variable's **count** property.

Some important points

- You can still use useState(). You don't have to use useReducer() if you find useState() simpler. If you *do* find that your component code is getting complicated with a lot of state logic cluttering the component function, this is a good sign that a reducer is probably a good idea because it simplifies your component code by taking a lot of the state logic outside the component code. If you find your state variable refresh to be buggy due to its complexity, it's a good idea to start using reducers.
- You can use Context with reducers for global state.
- Reducers must be **pure**. This means that the "same inputs always result in the same output" (from the documentation). This means that if you dispatch a particular action, it always does the same thing. It should not make requests or do other things which affect things outside the component.
- Reducers should only make immutable updates.
- Each action should only describe a single interaction. This simplifies your code and makes it easier to follow and debug. For example, instead of an updateCount action (vague and too broad), increment and reset actions were defined.

Extra resources and self-study

- https://react.dev/reference/react/useReducer#adding-a-reducer-to-a-component
- https://react.dev/learn/extracting-state-logic-into-a-reducer
- https://dmitripavlutin.com/react-usereducer/#:~:text=D.-,Reducer%20function,and%20return%20the%20new%20state.

NOTE: This next resource is not reducer-related but if you were curious about implementing a login using React, take a look at this resource: https://www.digitalocean.com/community/tutorials/how-to-add-login-authentication-to-react-applications