Class based components

React components can be either `functional` or `class based`. Nowadays both types of components can be used to achieve the same goals but this was not always the case. Prior to React version 16.8, when hooks were introduced into functional components, there were certain pieces of React functionality that could only be implemented using class based components. If a component needed to hold some values in state, or to perform side effects through React lifecycle methods it needed to be a class as these features were unavailable in functional components.

Modern React apps no longer have this restriction and we can create all of our components as functions. We can still use class based components as part of our project and most legacy projects would use a mix of class based components where necessary and functional components for simpler presentational components. You may come across class based components in documentation or as part of existing React projects and examples of common scenarios can be found below. For more information on class syntax see the React docs

- Note that the docs themselves recommend using functions instead of classes.

For a refresher on classes in JavaScript see the fundamentals notes

Rendering JSX

Consider the following functional component.

```
// components/Greeting.js
const Greeting = (props) => {
  return (
    <section>
      <p>Hello, nice to meet you {props.name}!</p>
    </section>
  );
};

// App.js
const App = () => {
  return (
    <div>
      <Greeting name='Paul' />
    </div>
  );
}
```

React provides us with a `Component` class that we can use to create our own components. To create a class based component we will create a Javascript class that inherits from `Component` using the `extends` keyword.

constructor

When writing the constructor for an inherited class the first thing we must do is to call the parent class constructor using the `super` keyword. This will call the `React.Component` constructor allowing us to inherit all of the functionality of that class. Our constructor will be invoked with the `props` of our component and one of the things `super(props)` will do is attach the props to the instance of our class on `this.props`.

nb Our constructor will be invoked by React and we will not need to instantiate the class ourselves so no need for the `new` keyword as this will be handled by React.

render

In functional components we would return some JSX to be rendered. As we are now in a class we must define a method called `render` to achieve the same purpose. This method is named by React and here we will return the JSX for our components UI. Note that to access the props we need to use the `this` keyword in order to access the instance properties

```
import React from 'react';

class Greeting extends React.Component {
  constructor(props) {
    super(props); // this.props = props
  }

  render() {
    return (
      <section>
        <p>Hello, nice to meet you {this.props.name}!</p>
      </section>
    );
  }
}

// App.js
const App = () => {
  return (
    <div>
      <Greeting name='Paul' />
    </div>
  );
}
```

State

Class based components can be used to hold values in state. We can hold state values on a property of `state` and update those values with an inherited method called `setState`

this.state

In our constructor we can set the initial value of our components state. As we can only have a single value in state this value is an object which can hold multiple properties.

this.setState

When extending from the `React.Component` class we inherit the `setState` method. This can be used to update the components state following the same rules as the functions returned by `useState`. We can pass a new state object or pass a function that is invoked with the current state.

nb If we pass a new state object to `setState` it will be merged with the current state instead of replacing the existing state wholesale. This allows us to update single state values without having to copy all of the existing values every time. For more on this behaviour see the setState docs

```
const CounterWithHooks = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount((currCount) => currCount + 1)}>Add one</button>
    </div>
  )
};

class Counter extends React.Component {
  constructor(props) {
    super(props);
    // set initial state
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        // update the state using this.setState
        <button
          onClick={() =>
            this.setState((currState) => {
              return { count: currState.count + 1 };
            })
          }
        >
          Add one
        </button>
      </div>
    );
  }
}
```

setState helper methods

As we are writing classes we can define methods on that class to extract logic. It is a common pattern to extract `setState` logic to it's own method to keep the `render` clean.

In our counter example we can extract the logic of incrementing the counter to a method which can be passed as the `onClick` handler.

When running `this.setState` we must ensure that the value of `this` still points to the instance of our component. When writing an inline event handler we used an arrow function so the value of `this` is bound to the scope the arrow function was defined in.

If we extract the function however `this` will be bound at the point of invocation and not refer to the component. To deal with this we must bind the function's `this` value in the constructor so that it always refers to the instance of our component.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
    // bind incrementCounts value of this to the correct instance of Counter
    this.incrementCount = this.incrementCount.bind(this);
  }

  // define a custom method
  incrementCount() {
    this.setState((currState) => {
      return {
        count: currState.count + 1,
      };
    });
  }

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        // pass the method as a handler
        <button onClick={this.incrementCount}>Add one</button>
      </div>
    );
  }
}
```

Lifecycle methods and side effects

In order to perform side effects such as api calls in class based components we have a number of lifecycle methods we can use. Notable ones are

- `componentDidMount` - Invoked immediately after the component is first rendered.
- `componentDidUpdate` - Invoked whenever props change, setState or forceUpdate are called
- `componentWillUnmount` - Invoked immediately before a component is removed from the page

Here's a handy visualisation of when lifecycle methods run

componentDidMount

Consider the following `useEffect` that fetches some data from an api the first time a component is mounted.

```
// rest of the component holding planets in state omitted
useEffect(() => {
  fetch('https://space-facts.herokuapp.com/api/planets')
    .then((res) => res.json())
    .then((data) => setPlanet(data.planets));
}, []);
```

To perform similar logic in a class based component we would use an appropriate lifecycle method to fetch the data, in this case `componentDidMount`.

```
import React from 'react';

class Planets extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      planets: [],
    };
  }

  componentDidMount() {
    fetch('https://space-facts.herokuapp.com/api/planets')
      .then((res) => res.json())
      .then((data) => {
        this.setState({ planets: data.planets });
      });
  }

  render() {
    return (
      <section>
        <h2>Planets</h2>
        <ul>
          {this.state.planets.map((planet) => {
            // render each planet...
          })}
        </ul>
      </section>
    );
  }
}
```

componentDidUpdate

When fetching data with `useEffect` we often have dependencies when the effect should be re-ran, by adding a variable to the dependency array React will compare the current value against the value on the previous render to see if they differ.

```
const SinglePlanetWithHooks = ({ planetId }) => {
  const [planet, setPlanet] = useState({});

  useEffect(() => {
    fetch(`https://space-facts.herokuapp.com/api/planets/${planetId}`)
      .then((res) => res.json())
      .then((data) => setPlanet(data.planet));
  // if planetId changes between renders, re-run the effect
  }, [planetId]);

  return (
    <section>
      <h2>{planet.name}</h2>
      {/* render the rest of the planet info */}
    </section>
  );
};
```

To achieve a similar effect in a class based component we can use the `componentDidUpdate` lifecycle method. This method is invoked with some arguments in order for us the make the comparison between renders.

- 0: the previous render's props
- 1: the previous render's state

We can then compare the two to see if the planetId has changed re-fetch the data if it has.

nb Just as with `useEffect` dependencies be careful how this comparison is made to avoid infinite loops. Notable `prevProps` and `prevState` are objects and `prevProps !== this.props` will always be true as they are both objects and have different references.

```
class SinglePlanet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      planet: {},
    };
    this.fetchPlanet = this.fetchPlanet.bind(this);
  }

  componentDidMount() {
    fetchPlanet();
  }

  componentDidUpdate(prevProps, prevState) {
    if (prevProps.planetId !== this.props.planetId) {
      fetchPlanet();
    }
  }

  fetchPlanet() {
    fetch(
      `https://space-facts.herokuapp.com/api/planets/${this.props.planetId}`
    )
      .then((res) => res.json())
      .then((data) => {
        this.setState({ planet: data.planet });
      });
  }

  render() {
    return (
      <section>
        <h2>{planet.name}</h2>
        {/* render the rest of the planet info */}
      </section>
    );
  }
}
```

copy original markdown to clipboard