Complex Components

Key principles:

- **Single responsibility:** Reduce components to the smallest size so that they aren't responsible for multiple things
- **Separation of concern:** Breaking an application into different layers or types of components

1. Create subcomponents
2. Create complex component and pass required parameters to subcomponents.
3. Render complex component to get all subcomponents.

Functional Components (Stateless Components)

```
let container=document.querySelector("#container");
let root=ReactDOM.createRoot(container);

function Author(props){
        return(
                <div>
                        <h4>{props.author}</h4>
                        <p>@{props.authdl}</p>
                </div>
        )

function Content(props){
        return(
                <p>{props.content}</p>
                <p>{props.when}</p>
        )
}

function Reaction(props){
        return(
                <div>
                        <p style={{display:"inline"}}>{props.numcomm} comments</p>
                        <p style={{display:"inline"}}>{props.numlikes} likes</p>
                </div>
        )
}

function Post(props){
        return(
                <div>
                        <Author  {...props} />
                        <Content  {...props} />
                        <Reaction  {...props} />
                </div>
        )
```

```
}

root.render(
        <div>
                <Post author="Elon Musk" authdl="daboss" content="Trump is the best
presidential candidate" when="1 hour ago" numcomm="3214" numlikes="420,069"></Post>
        </div>
);
```

Class Components

```
const container = document.querySelector("#container");
const root = ReactDOM.createRoot(container);

class Author extends React.Component {
  render() {
    return (
      <div>
        <h4>{this.props.author}</h4>
        <p>@{this.props.authdl}</p>
      </div>
    );
  }
}

class Content extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.content}</p>
        <p>{this.props.when}</p>
      </div>
    );
  }
}

class Reaction extends React.Component {
  render() {
    return (
      <div>
        <p style={{ display: "inline", marginRight: "10px" }}>
          {this.props.numcomm} comments
        </p>
        <p style={{ display: "inline" }}>{this.props.numlikes} likes</p>
      </div>
    );
  }
}

class Post extends React.Component {
  render() {
    return (
```

```
      <div>
        <Author author={this.props.author} authdl={this.props.authdl} />
        <Content content={this.props.content} when={this.props.when} />
        <Reaction numcomm={this.props.numcomm} numlikes={this.props.numlikes} />
      </div>
    );
  }
}

root.render(
  <div>
    <Post
      author="Elon Musk"
      authdl="daboss"
      content="Trump is the best presidential candidate"
      when="1 hour ago"
      numcomm="3214"
      numlikes="420,069"
    />
  </div>
);
```

Note: Spread Operator (`...`)

- Used to expand elements of an iterable (like an array or object) into individual elements or key-value pairs.
- Used here to pass all props from one component to another without explicitly listing each one.

Note: Empty Complex Component

If no parameters are passed, each prop value will just be taken as empty and nothing will be shown where those values are accessed.

Dynamically Passing Values

```
let postdata = [{author:"Elon Musk", authdl:"daboss", content:"Hello", when:"1 hour ago", numcomm:"3214", numlikes:"420,069"},

    {author:"Narendra Modi", authdl:"pm", content:"Hello again", when:"3 hours ago", numcomm:"3294", numlikes:"49238"}];

root.render(
        <div>
                <Post  {...postdata[0]}></Post>
                <Post  {...postdata[1]}></Post>
        </div>
);
```

Component State

- States -> used with component classes to make them dynamic, enables them to keep track of changing info
- Also called smart/container components
- Three main operations:
  - Initialization of state
  - Change state based on certain trigger
  - Usage of updated state

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

  // Method to update state
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Counting Down using Lifecycle Methods

```
const root = ReactDOM.createRoot(document.querySelector("#root"))

class Counter extends React.Component{
  constructor(props){
    super(props)
    this.state = { count: 0 }
  }

  increment = () => {
    this.setState( { count: this.state.count + 1 } )
  }

  render(){
    return(
      <div>
        <p>State Value: {this.state.count}</p>
```

```
        <button onClick={this.increment}>Increment</button>
      </div>
    )
  }
}

class CountDown extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      seconds: props.start,
    };
  }

  componentDidMount() {
    // Use this.tatata instead of this.t
    this.tatata = setInterval(this.timer.bind(this), 1000);
  }

  componentDidUpdate(prevprops,prevstate){
    console.log("component updated from" +prevstate.seconds);
    if(this.state.seconds == 0){
        root.unmount();
    }
}
  componentWillUnmount() {
    // Clear the interval using this.tatata
    console.log("Unmounting..")
    clearInterval(this.tatata);
  }

  timer() {
    // if (this.state.seconds === 0) {
    //   clearInterval(this.tatata);          //unnecessary when using the DidUpdate
Condition
    // } else {
      this.setState((prevState) => ({
        seconds: prevState.seconds - 1,
      }));
    }


  render() {
    return <h1>{this.state.seconds}</h1>;
  }
}


root.render(
<div>
<Counter/>
<CountDown start={10}/>
</div>);
```

NOTE: Refer beginning for typical stateless components (functional). Also called presentational/dump components