WORKSHOP #3

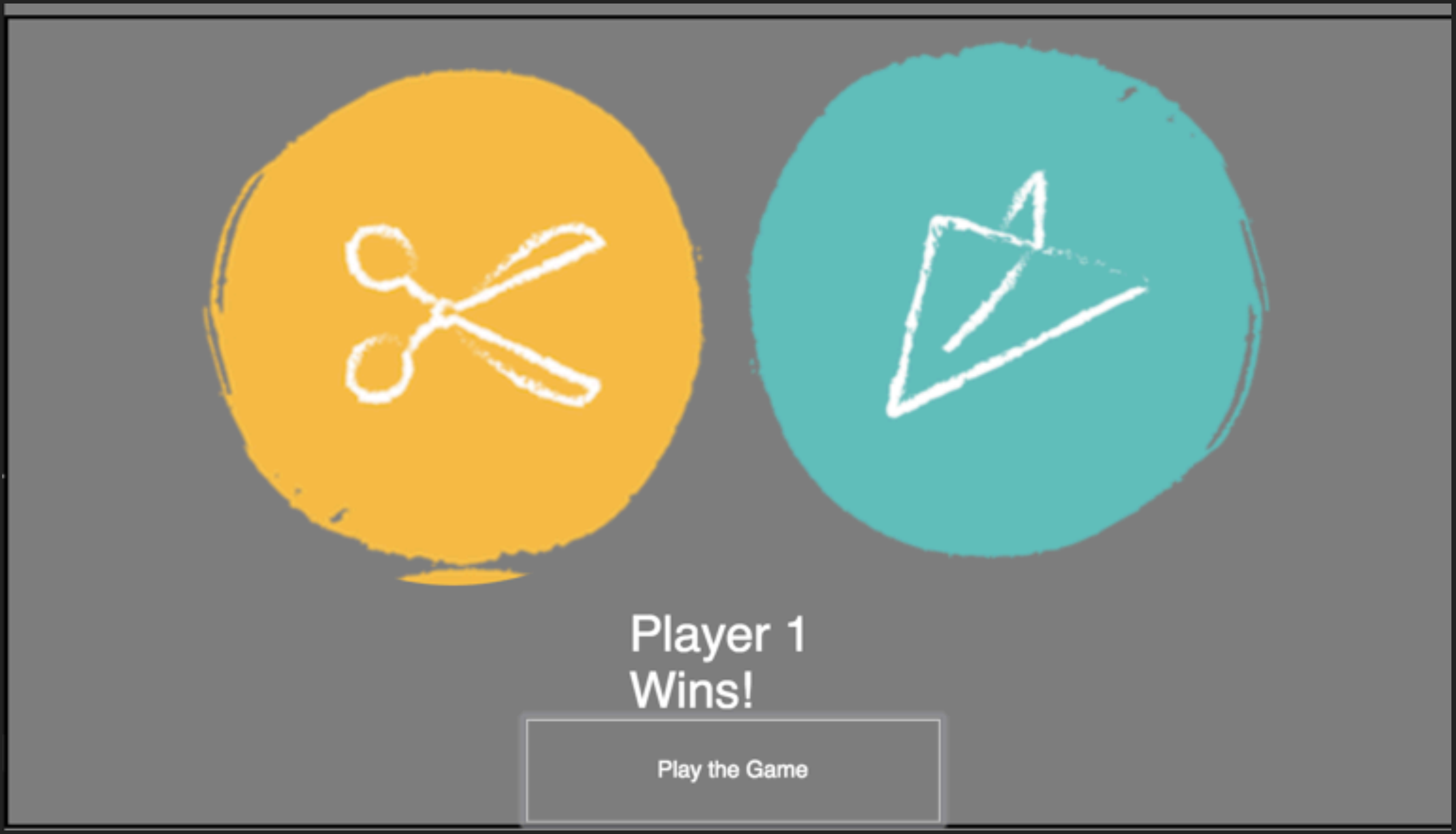
ROCK SCISSORS PAPER

OPEN YOUR PROJECT IN AND IDE

- Create a directory for your project by using the create-react-app command.
- If you use Atom you want to navigate to your project directory in the command prompt and type in **atom .**
- If you use Sublime, navigate to your project directory and in the command prompt type in **subl .**
- If you use VS code, navigate to your project directory and in the command prompt type in **code .**

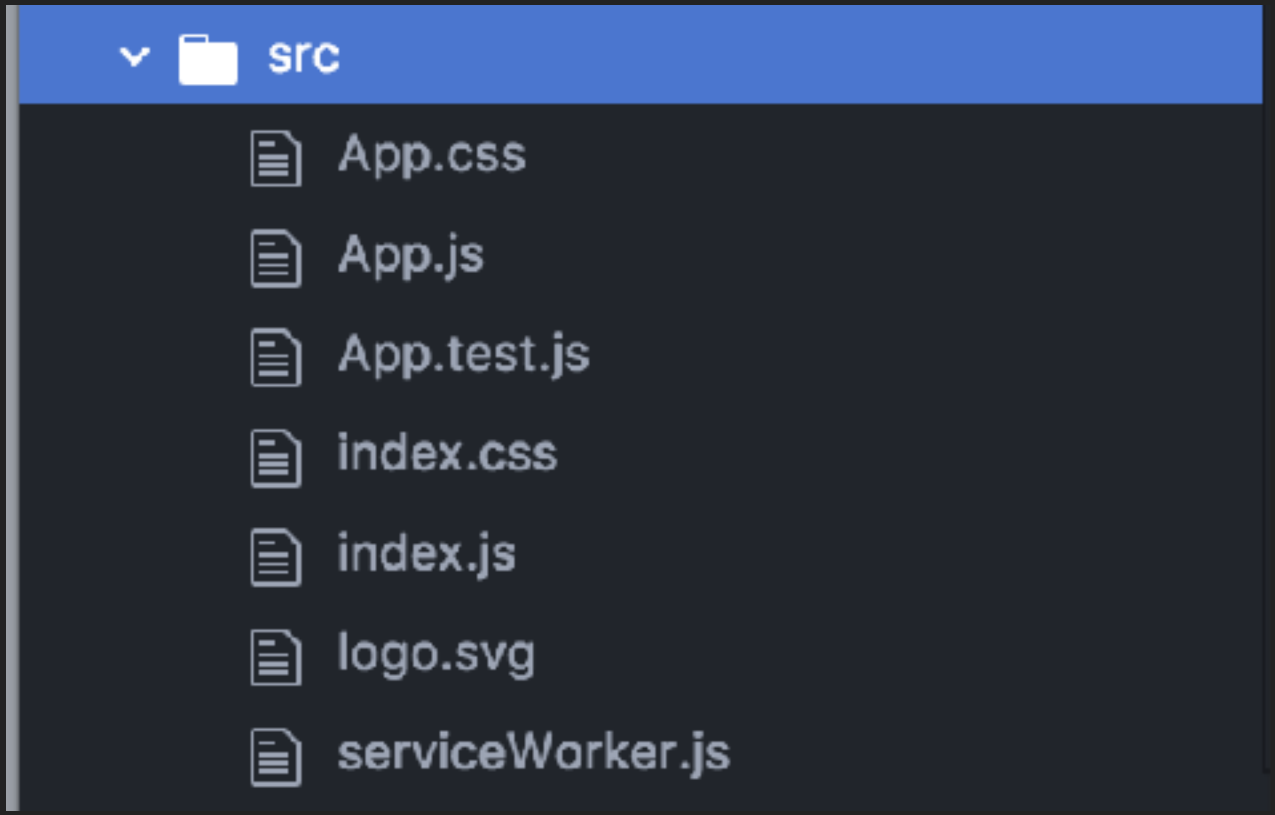
OUR GOAL

Our goal is to build an rock, scissors, paper game that will randomly generate both players picks and decide the winner.



ROCK, SCISSORS, PAPER

- Delete all the code in your src/App.js file and lets start from scratch again.



This is how your Files tree should look.

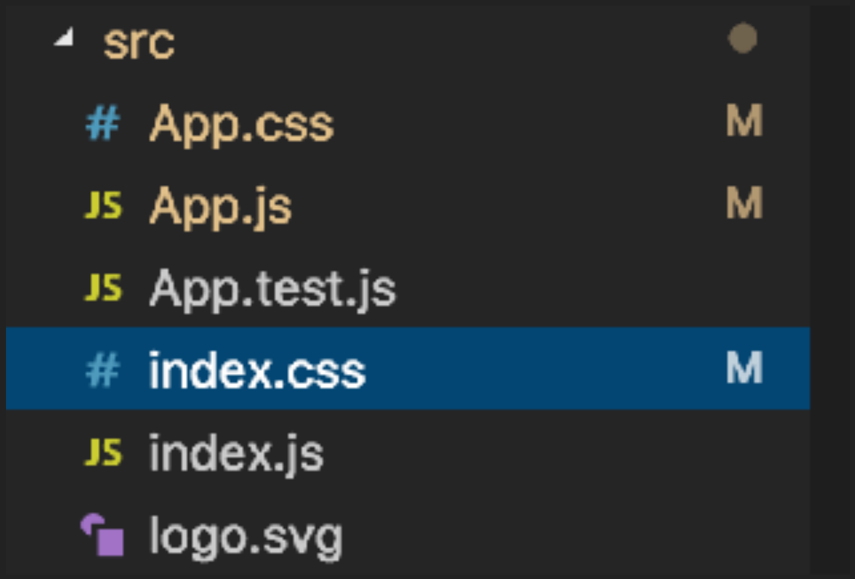
- So now that we have an idea on what we want to create. Take a moment and think about where should we start. See if you can answer these questions.
- What is the major functionality of the application?
- What type of components should we implement based on the functionality of the application? I.E. Stateful class component, or a pure functional component?
- How will you structure the JSX and styles to accomplish this user interface?

ROCK, SCISSORS, PAPER

- We know that this application will randomly generate the players picks.
- Our application will also displays images of the players picks, have a button that will play the game, and decide the winner.
- Usually, and almost always, when you have values that are changing in your views you are going to need Stateful component.
- I'll provide the CSS and guide you along the way on how to add in styling.

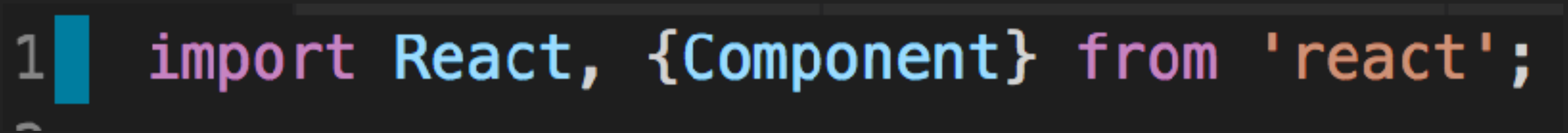
- Lets start by navigating to the github and copying the CSS that is prebuilt for this workshop.

    - Click for CSS and select workshop # 3 and then the CSS folder, then the click the index.css file and copy everything from the index.css file into your index.css file in your src directory.



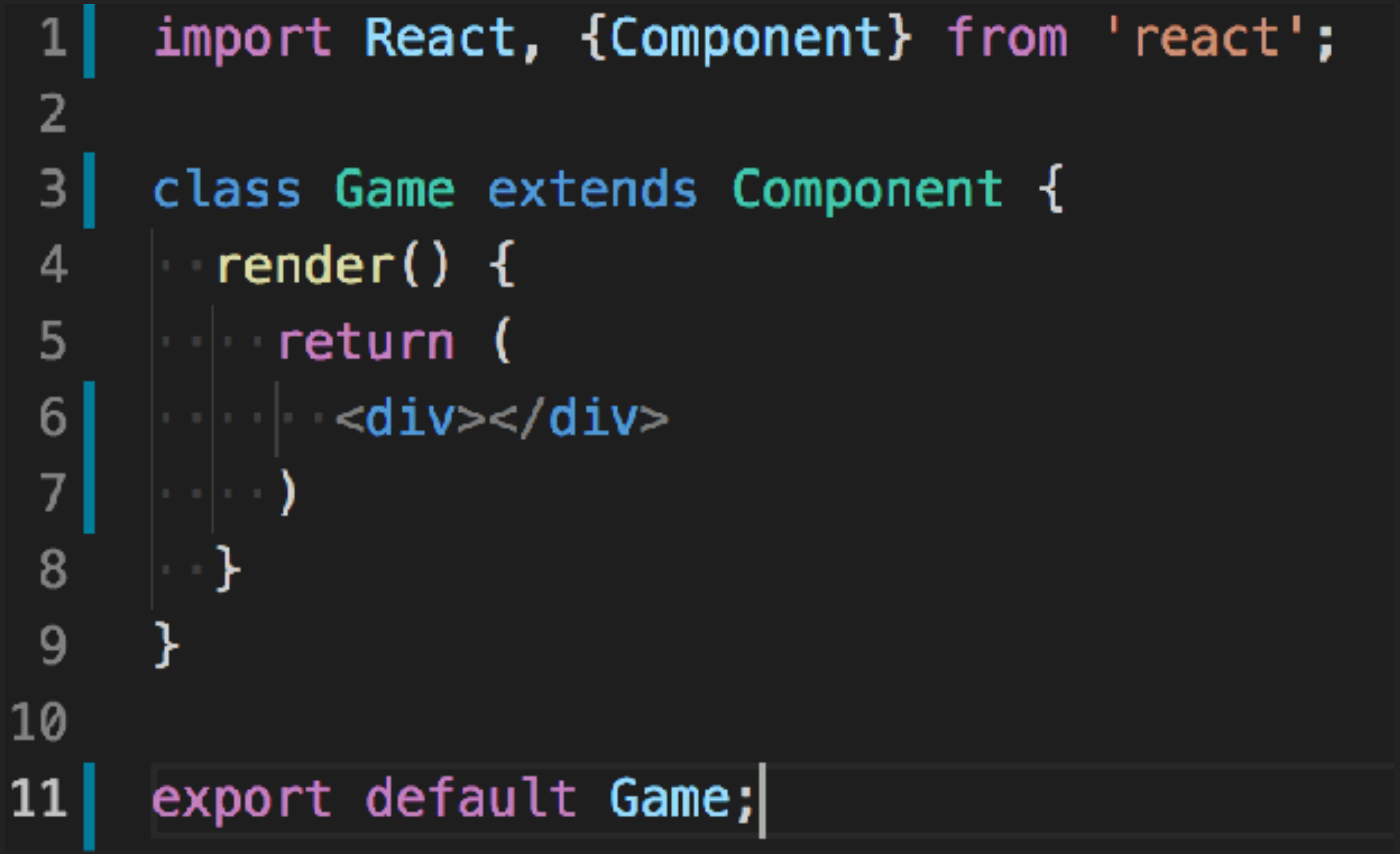
Delete all the CSS in your **index.css** and replace it with the css copied from the index.css file from github.

- Lets start by importing the proper libraries into your **App.js** file.
- Import React and the Component so we can create our stateful class component and use JSX.



```
import React, {Component} from 'react';
```

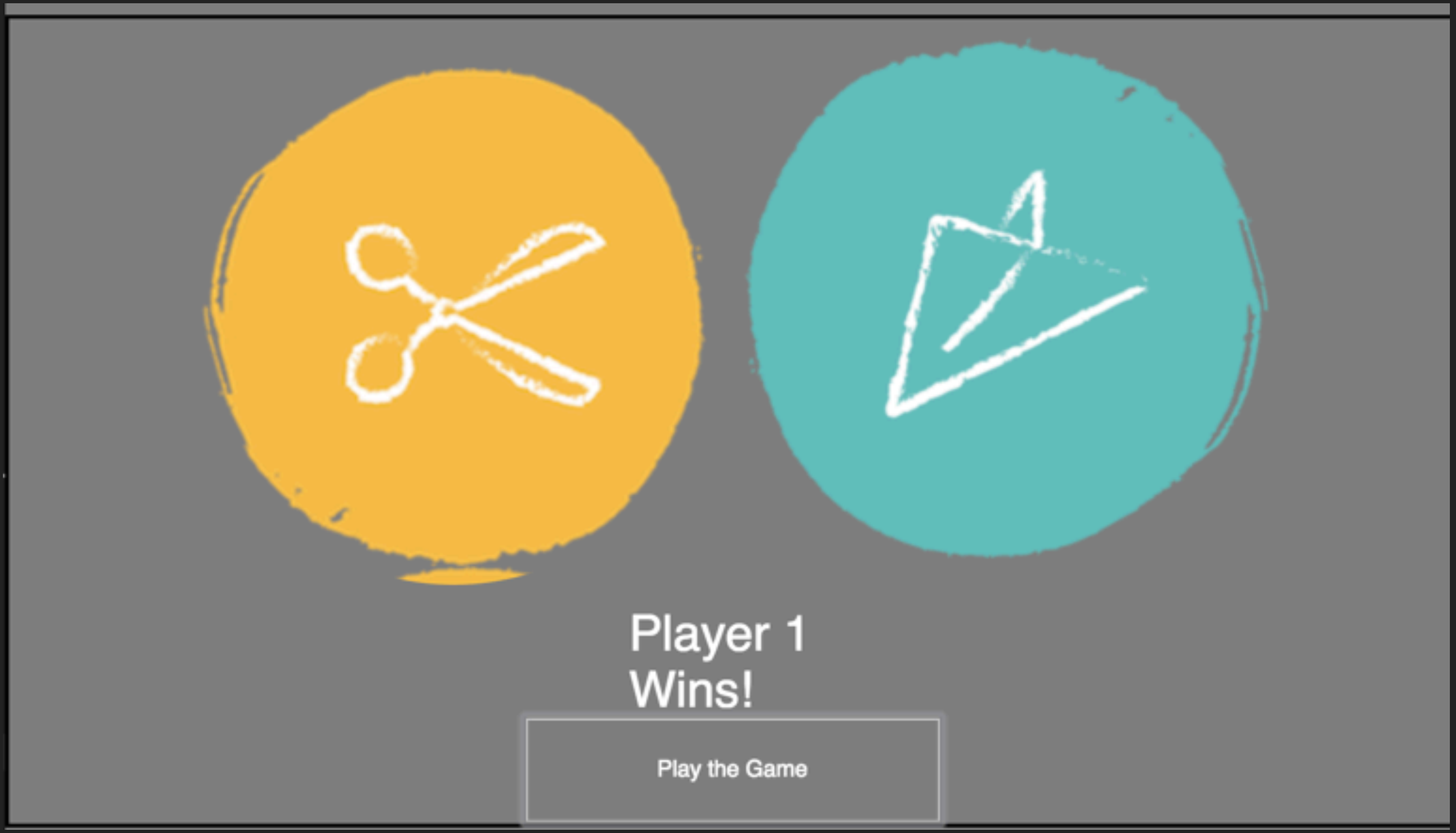
- Now we need to create our Game class component.



```
import React, {Component} from 'react';

class Game extends Component {
  render() {
    return (
      <div></div>
    )
  }
}

export default Game;
```

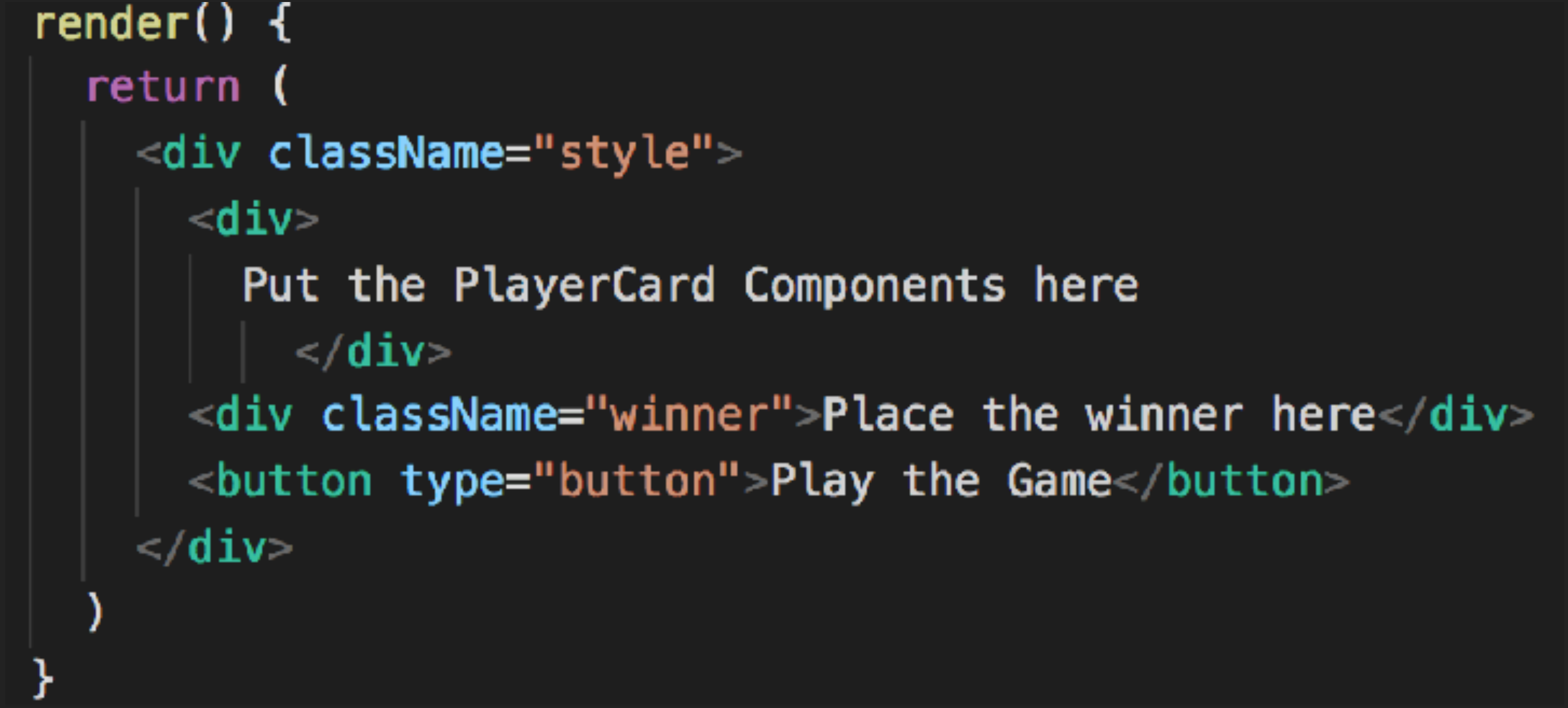
We export our game component so we are able to use this component throughout our application.

- Now that we have our component built, we should think about what we want our app to look like so we can start building the JSX. Lets take a look at the end result again.



ROCK, SCISSORS, PAPER

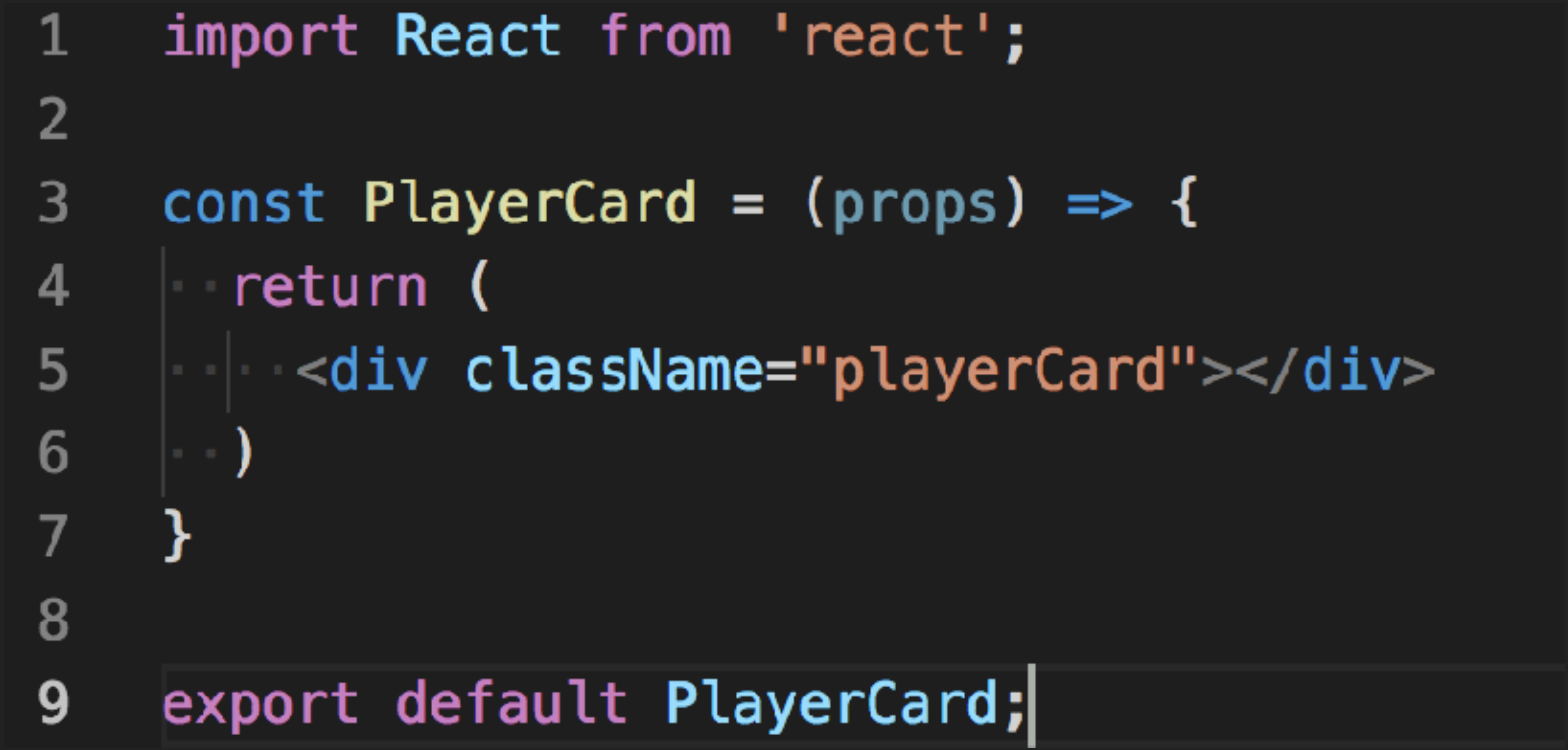
- To help get you started lets go ahead and build out what we know so far.



```
render() {
  return (
    <div className="style">
      <div>
        Put the PlayerCard Components here
          </div>
      <div className="winner">Place the winner here</div>
      <button type="button">Play the Game</button>
    </div>
  )
}
```

In your return statements you can only have one parent JSX element. A common pattern is to wrap everything in a div.

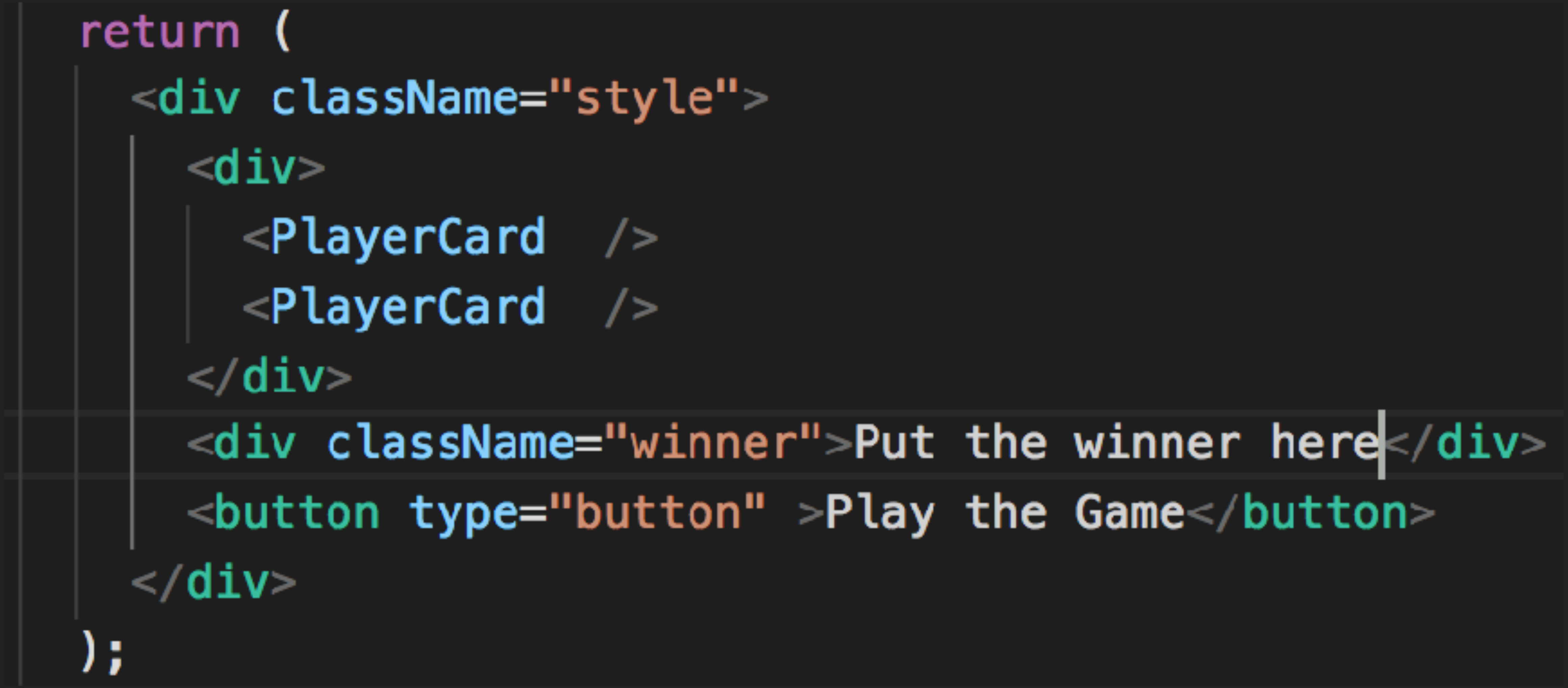
- In this workshop we will be using a functional component we defined as PlayerCard. Lets create a file named **PlayerCard.js in our src directory,** and start coding our functional component.



```
import React from 'react';

const PlayerCard = (props) => {
  return (
    <div className="playerCard"></div>
  )
}

export default PlayerCard;
```

ROCK, SCISSORS, PAPER

- Navigate back to your App.js file.
- Make sure to Import our PlayerCard component from "./PlayerCard.js"
- We only want two of our PlayerCard components to be rendered in our Game component between a div tag.



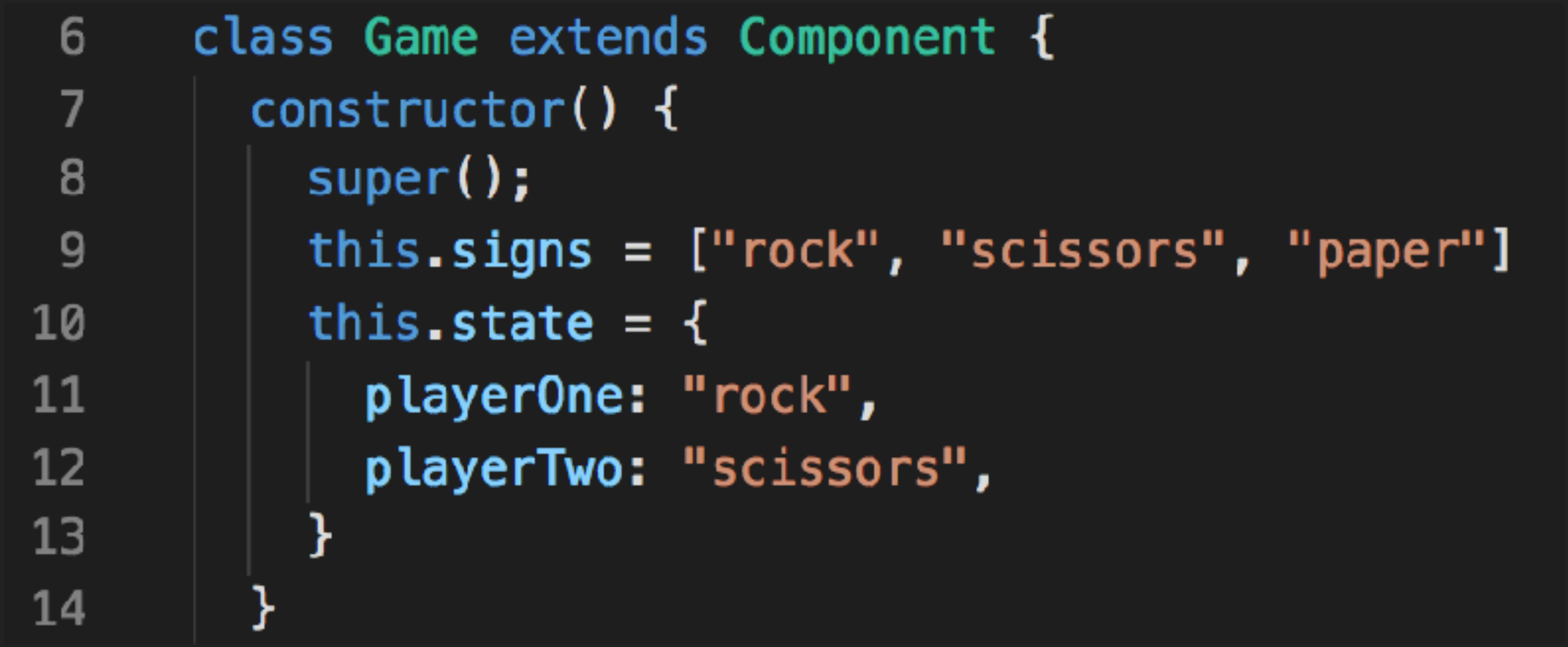
```
return (
  <div className="style">
    <div>
      <PlayerCard  />
      <PlayerCard  />
    </div>
    <div className="winner">Put the winner here</div>
    <button type="button" >Play the Game</button>
  </div>
);
```

- Remember, that we need a stateful component. Think for a moment: what is changing in our RSP app as user interact with it, and how could we represent that data?

- The data that changes in our UI is the players generated pick that is being represented by a string at first and then an image. Let's add state to our component, and some behavior that will cause that state to change.

- To do this we need to add a constructor to your Game class component and define state. Note that state is always defined as a Javascript object that will have key value pairs.

- In this workshop we are going to be adding the signs as well. We will use this array to randomly generate picks from, but now we can set them in our constructor so we can use them through our application.
- Navigate to your App.js file and add your constructor here.

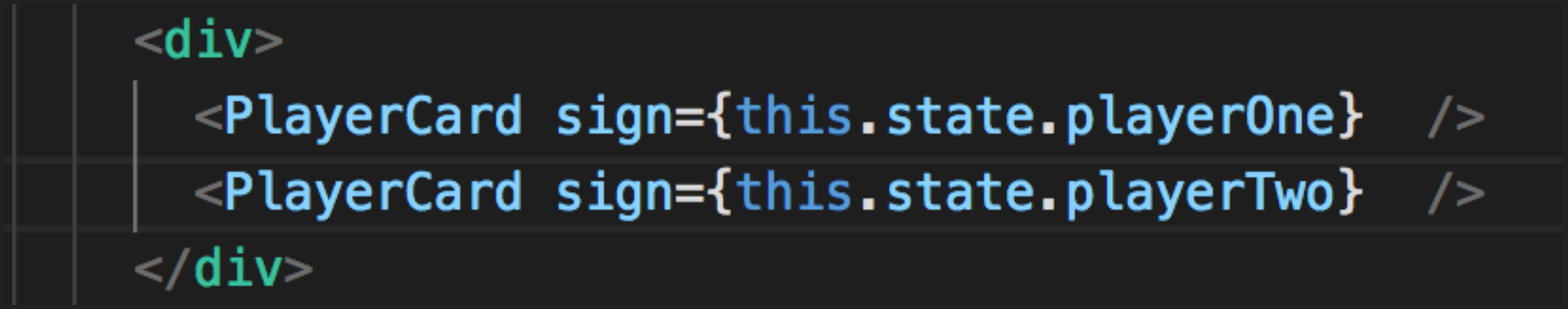


```
class Game extends Component {
  constructor() {
    super();
    this.signs = ["rock", "scissors", "paper"]
    this.state = {
      playerOne: "rock",
      playerTwo: "scissors",
    }
  }
```

We initialize our state by setting this.state to a Javascript object and the key and value we want our component to be initialized to.

ROCK, SCISSORS, PAPER

- Now that we have state that represents the playerOne and playerTwo picks. We want to pass those values into our PlayerCard component that will allow us to represent each players generated choice.
- In our Game component. Add these properties to our PlayerCard Components.

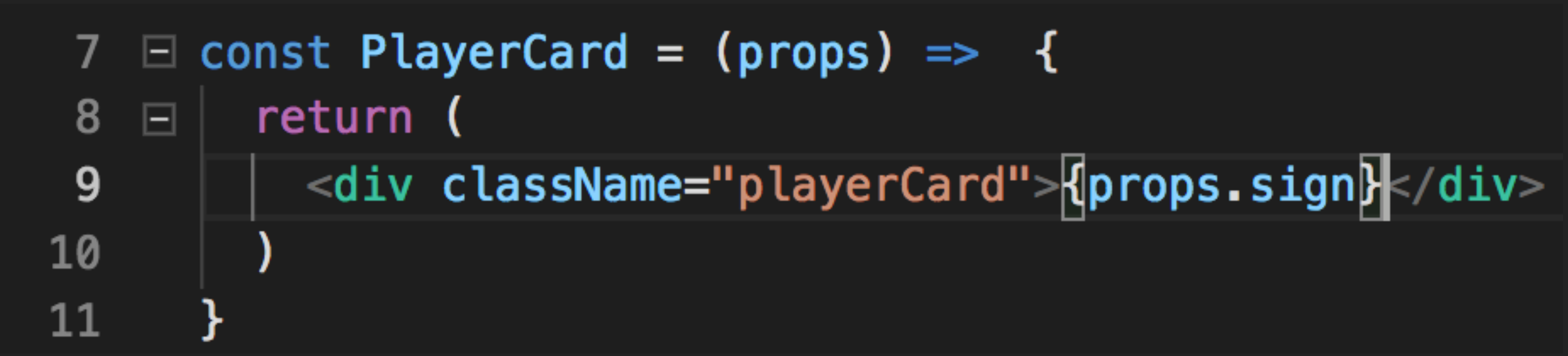


```
<div>
  <PlayerCard sign={this.state.playerOne}  />
  <PlayerCard sign={this.state.playerTwo}  />
</div>
```

The **sign** property we are passing to our component is going to be passed in as a **key** on the props object and the **value** will be whatever this.state.playerOne || this.state.playerTwo equals.

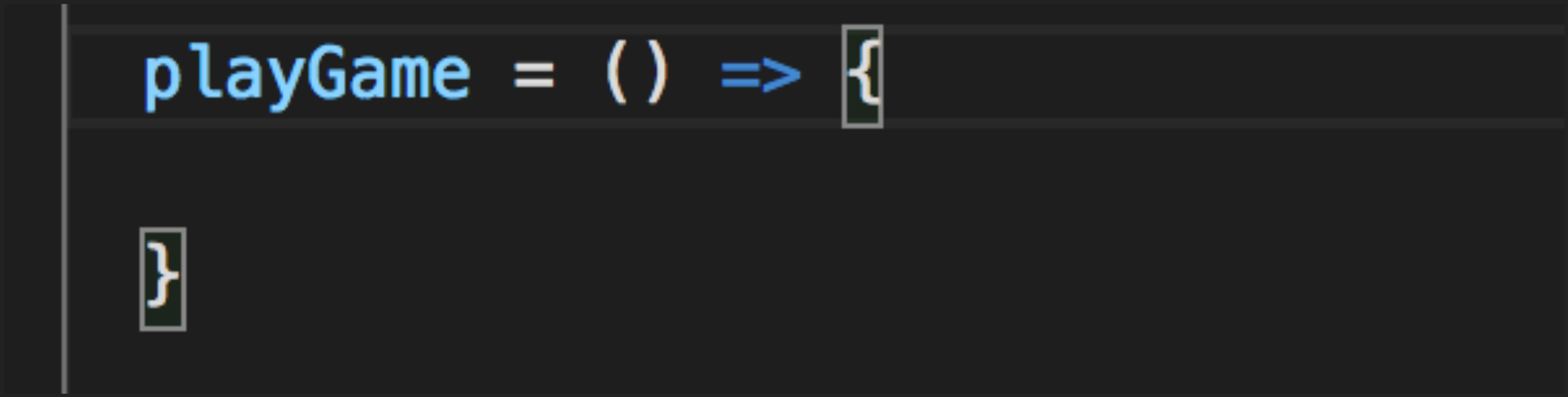
ROCK, SCISSORS, PAPER

- Now that we're passing the sign property into our PlayerCard component.   Navigate to our PlayerCard.js file and lets grab that sign key and represent its value between our div tags.
- Remember that props is an object with key value pairs so we can use the dot notation to access the values within the object.



```
const PlayerCard = (props) =>  {
  return (
    <div className="playerCard">{props.sign}</div>
  )
}
```

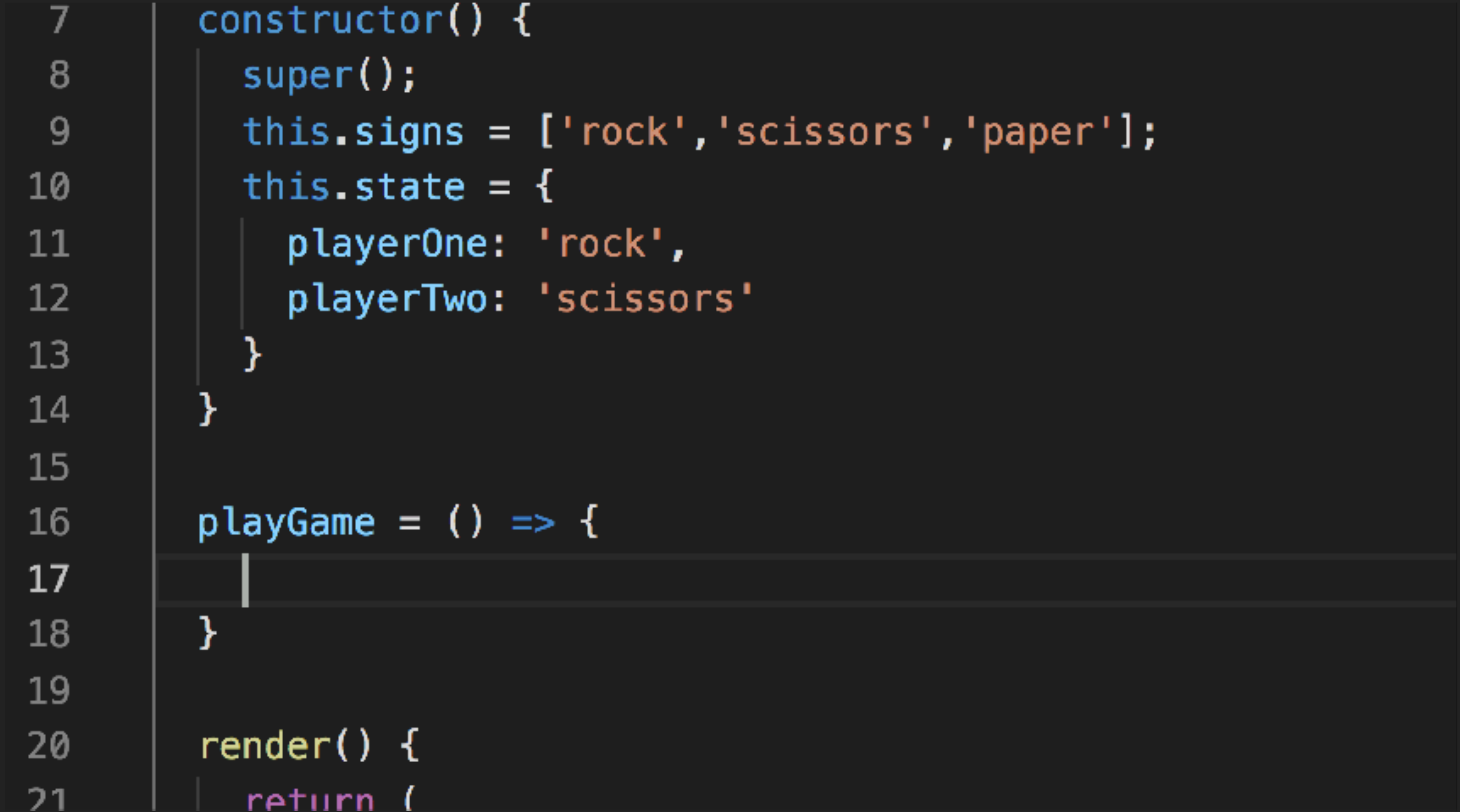
- Now that we have our blueprint all set up. Lets make this game randomly generate the signs.
- Back in our App.js file, we want to create a method in our Game component that will allow us to set the state to a randomly generated sign from our signs array. Think for a moment how you would do this in Javascript.



```
playGame = () => {

}
```

Remember!! That methods come after the constructor code block, and before the render () method.

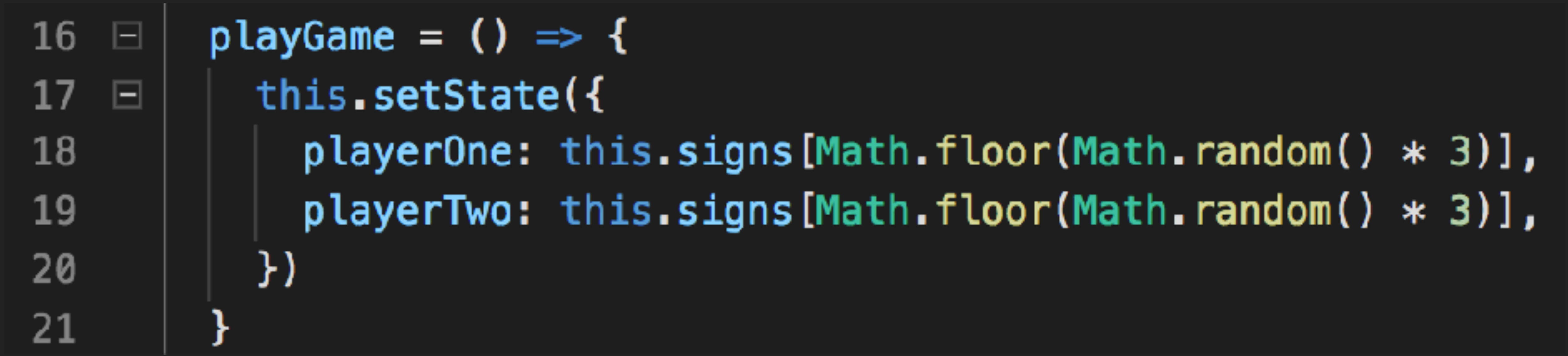
This is where our playGame method should be.



```
  constructor() {
    super();
    this.signs = ['rock','scissors','paper'];
    this.state = {
      playerOne: 'rock',
      playerTwo: 'scissors'
    }
  }

  playGame = () => {

  }

  render() {
    return (
```

ROCK, SCISSORS, PAPER

- When we need to set the state to a new value. We always call the setState method. setState is actually copying a new value into the key playerOne and playerTwo.

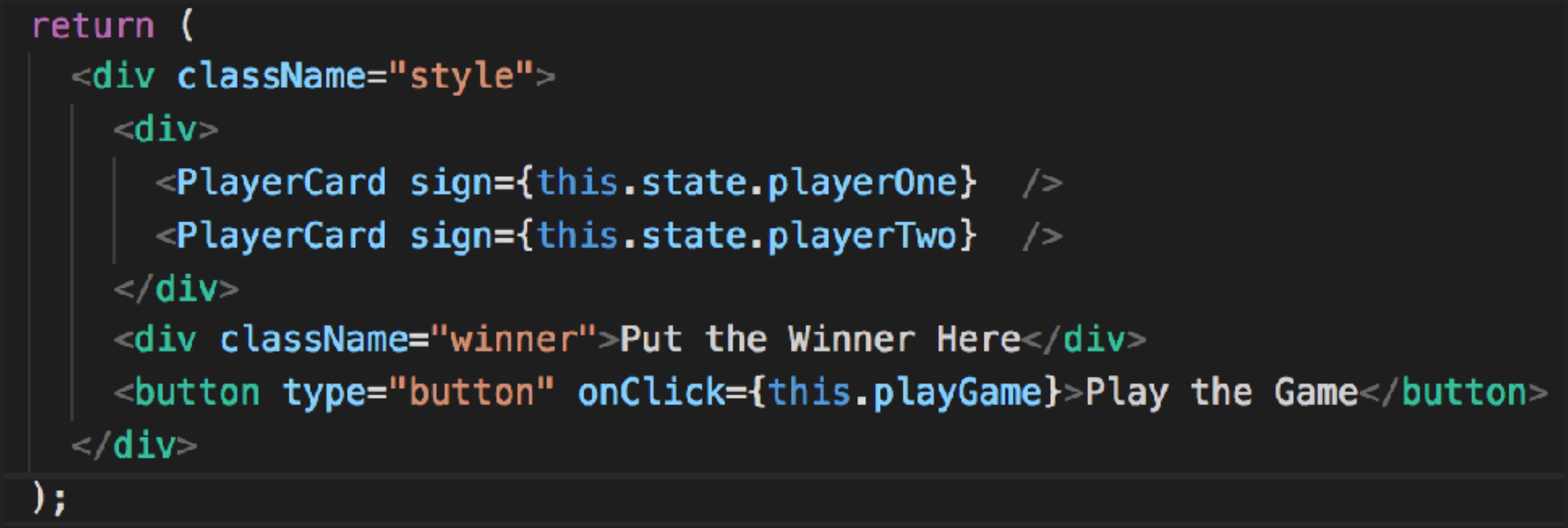


```
playGame = () => {
  this.setState({
    playerOne: this.signs[Math.floor(Math.random() * 3)],
    playerTwo: this.signs[Math.floor(Math.random() * 3)],
  })
}
```

We use the signs array and then call Math.random() *3 because Math.random generates values between 0-1. Then we use Math.floor() to remove any decimal places and round down.

ROCK, SCISSORS, PAPER

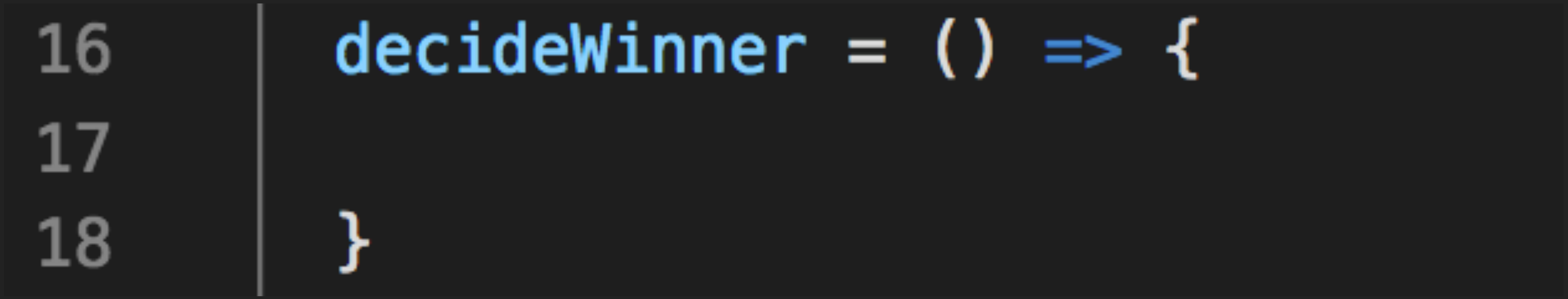
- Now that we have our playGame method completed. Lets attach a onClick event listener to our button. This will invoke our playGame method every time the button is clicked.



```
return (
  <div className="style">
    <div>
      <PlayerCard sign={this.state.playerOne}  />
      <PlayerCard sign={this.state.playerTwo}  />
    </div>
    <div className="winner">Put the Winner Here</div>
    <button type="button" onClick={this.playGame}>Play the Game</button>
  </div>
);
```

ROCK, SCISSORS, PAPER

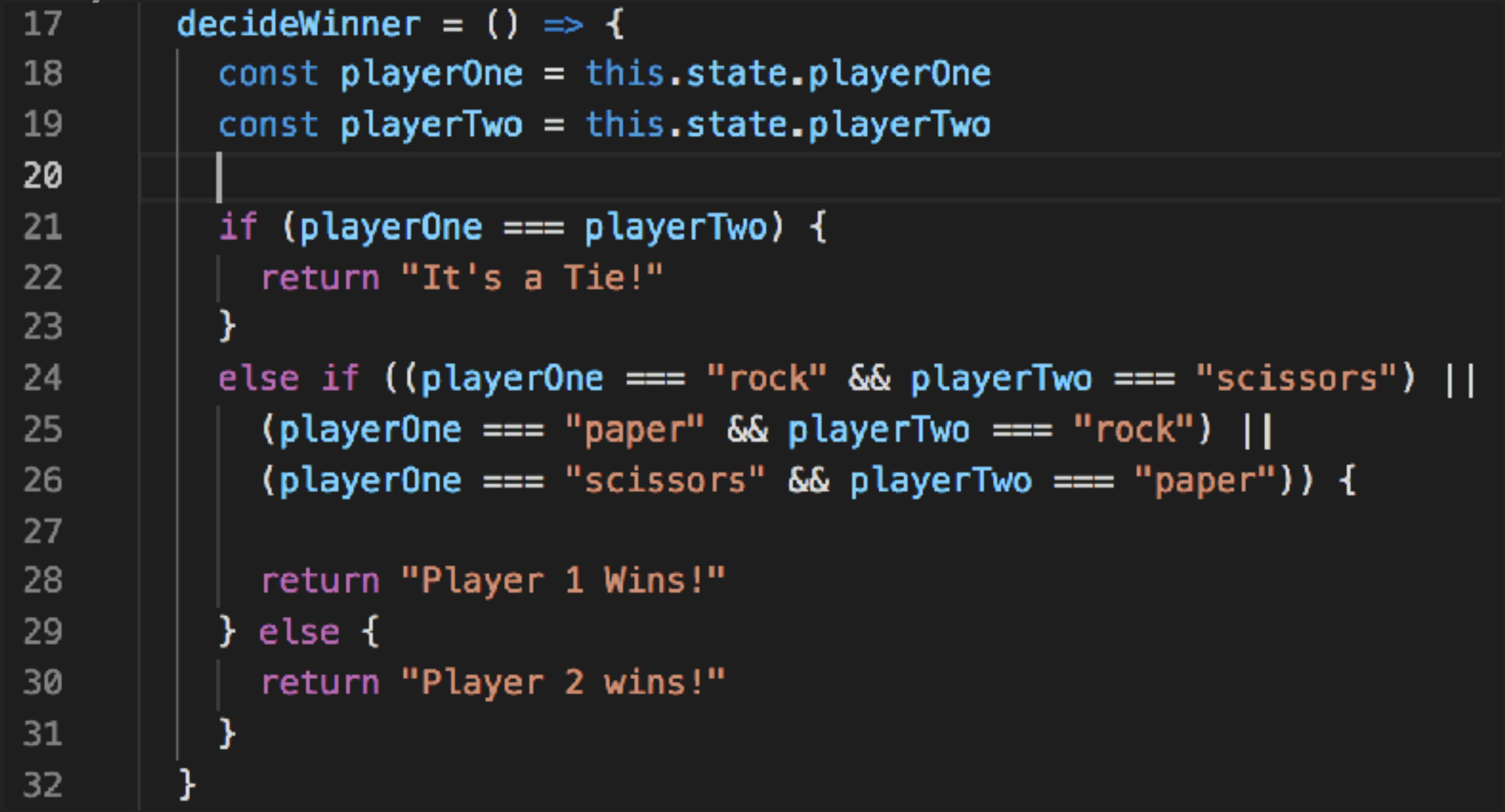
- Our game should now randomly generate signs for each player.
- Now we need to make sure our Game decides who has won the round. To do this we need to create another method. Lets call this method decideWinner.



```
decideWinner = () => {

}
```

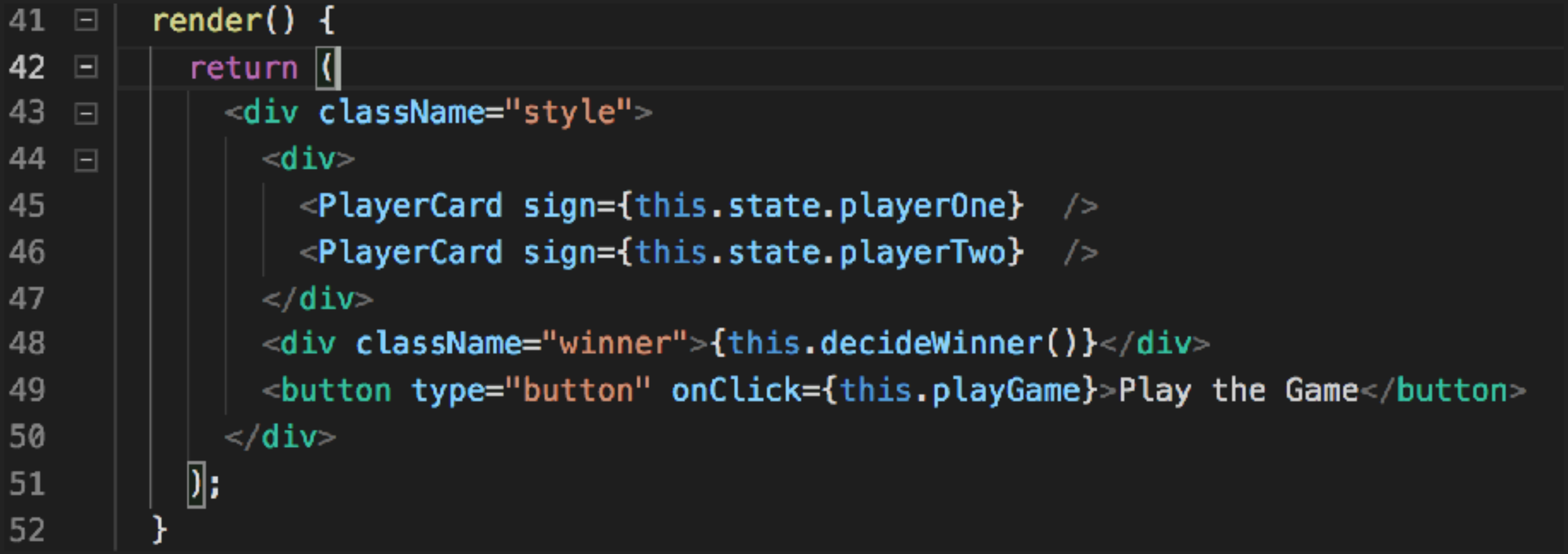
Think for a moment on how this is actually going to decide the winner.

- We know that rock beats scissors, paper beats rock, and scissors beats paper. Lets represent these conditionals in our method to decide who has won the round. If we coded this properly we should only have to do it for one player.



```
decideWinner = () => {
  const playerOne = this.state.playerOne
  const playerTwo = this.state.playerTwo

  if (playerOne === playerTwo) {
    return "It's a Tie!"
  }
  else if ((playerOne === "rock" && playerTwo === "scissors") ||
    (playerOne === "paper" && playerTwo === "rock") ||
    (playerOne === "scissors" && playerTwo === "paper")) {

    return "Player 1 Wins!"
  } else {
    return "Player 2 wins!"
  }
}
```

- Now we have the ability to decide who wins the round. We know we wanted to represent who won above the "Play the Game button" so lets place in the correct spot.

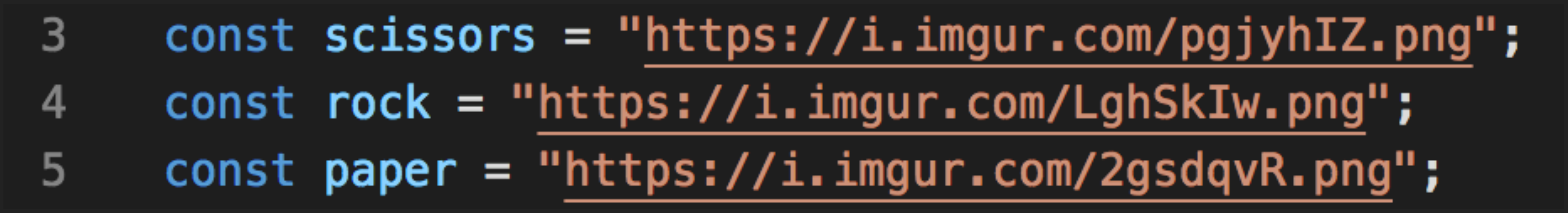


```
render() {
  return (
    <div className="style">
      <div>
        <PlayerCard sign={this.state.playerOne}  />
        <PlayerCard sign={this.state.playerTwo}  />
      </div>
      <div className="winner">{this.decideWinner()}</div>
      <button type="button" onClick={this.playGame}>Play the Game</button>
    </div>
  );
}
```

We know that every time the button is clicked that state changes, causing our game to re-render. By invoking our decideWinner method in this manner, this will run every time the button is clicked. Giving us the correct winner to be represented.

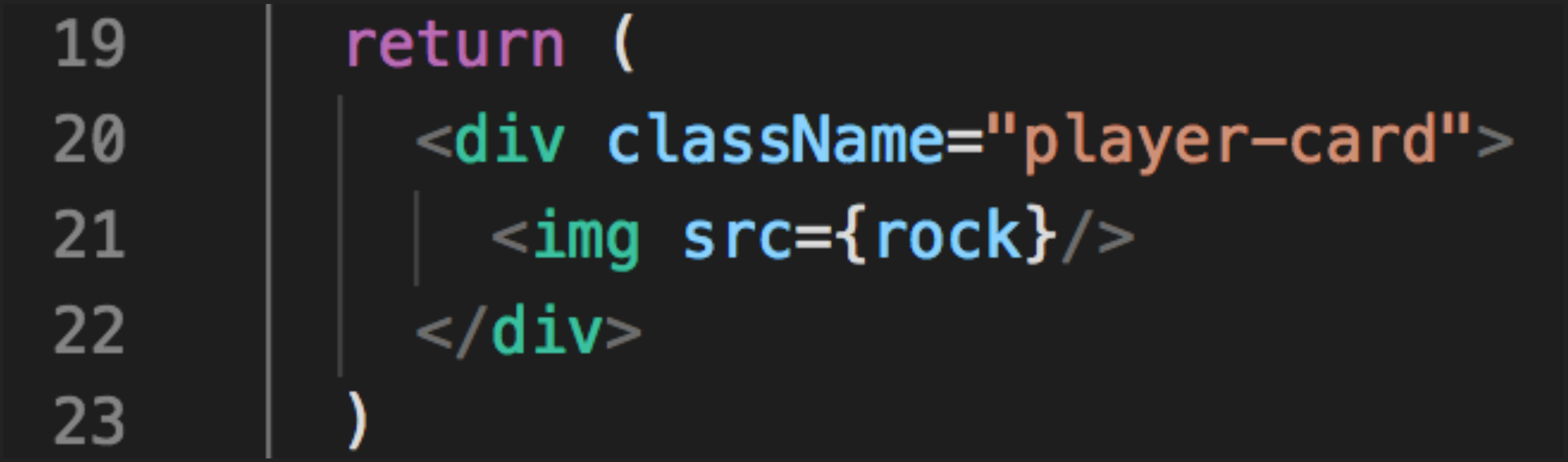
ROCK, SCISSORS, PAPER

- Now our end goal was to have images instead of the plain text we see rendering to our application. Lets set up where our PlayerCard component displays an image.
- In our PlayerCard.js file lets add these lines of code **below** our import statements and **above** our PlayerCard component.



```
const scissors = "https://i.imgur.com/pgjyhIZ.png";
const rock = "https://i.imgur.com/LghSkIw.png";
const paper = "https://i.imgur.com/2gsdqvR.png";
```

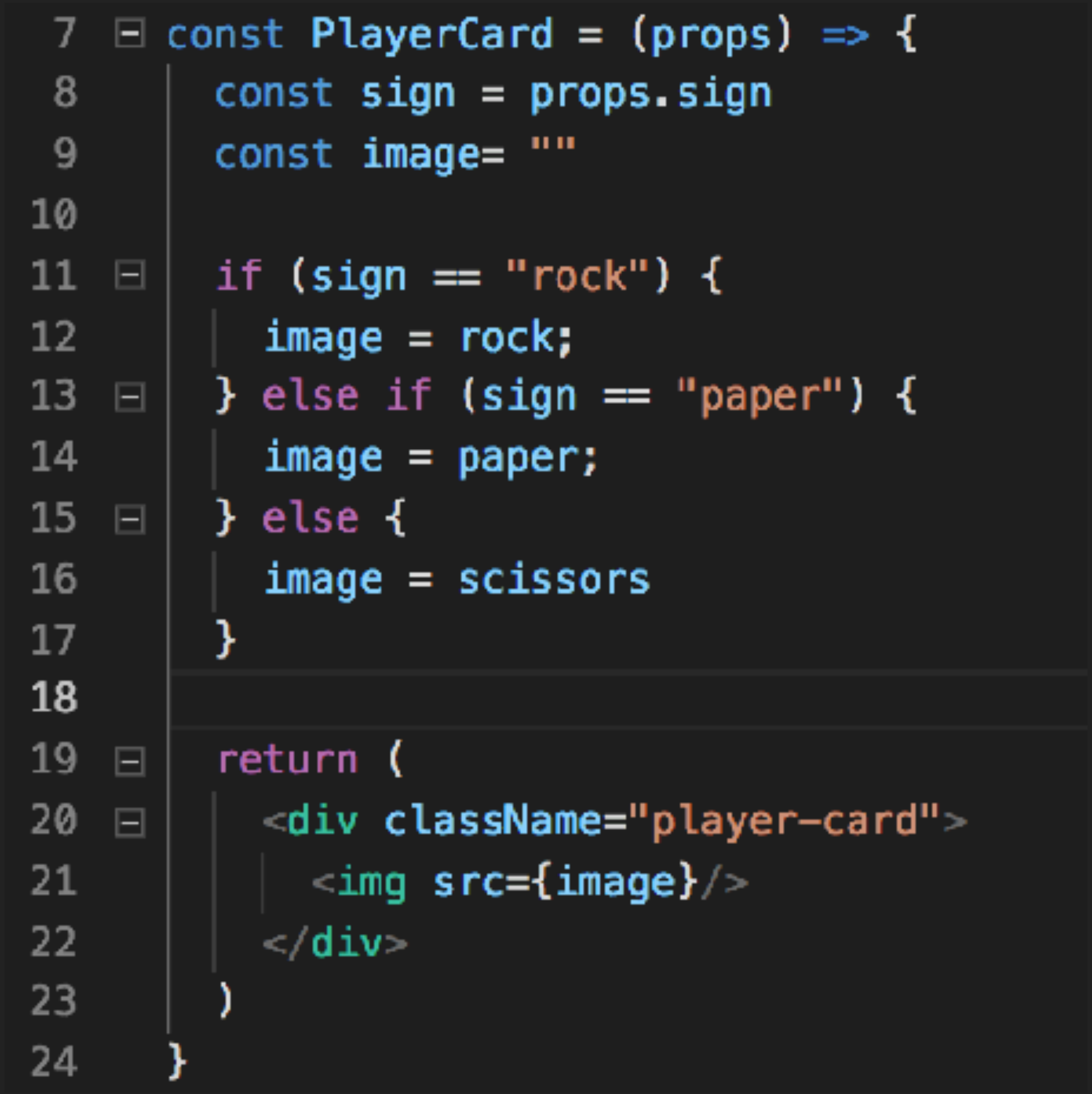
- Now that we have the url of our images. We need to set these images in our **PlayerCard.js component**. We can accomplish this by adding a <img /> tag between the main <div> in our JSX.



```
return (
  <div className="player-card">
    <img src={rock}/>
  </div>
)
```

Right now we have it hard coded to our rock image. Lets think about how we're going to represent this data if the props.sign being passed in is something other than rock.

- To accomplish the task of rendering the correct image corresponding to our props.sign value. We need to first create a variable that will hold our image string that we will set to depending on what value is being passed in.



```
 7  const PlayerCard = (props) => {
 8    const sign = props.sign
 9    const image= ""
10
11    if (sign == "rock") {
12      image = rock;
13    } else if (sign == "paper") {
14      image = paper;
15    } else {
16      image = scissors
17    }
18
19    return (
20      <div className="player-card">
21        <img src={image}/>
22      </div>
23    )
24  }
```

We create our image variable on line 9

Then, depending on what value the sign being pass in is. We then set the image to the correct url to be displayed.

Then we set the src of our <img> tag to represent this url

ROCK, SCISSORS, PAPER EDGE CASES

- Now your game should be correctly displaying the images, generating random signs, and picking the correct winner. YESSSSS!!!!

- EDGE CASE TIME!
- Set some names for the players.
- Make one player controlled and able to manually select what sign to play.