# Getting Started With React

André Furlanetti

## Repository

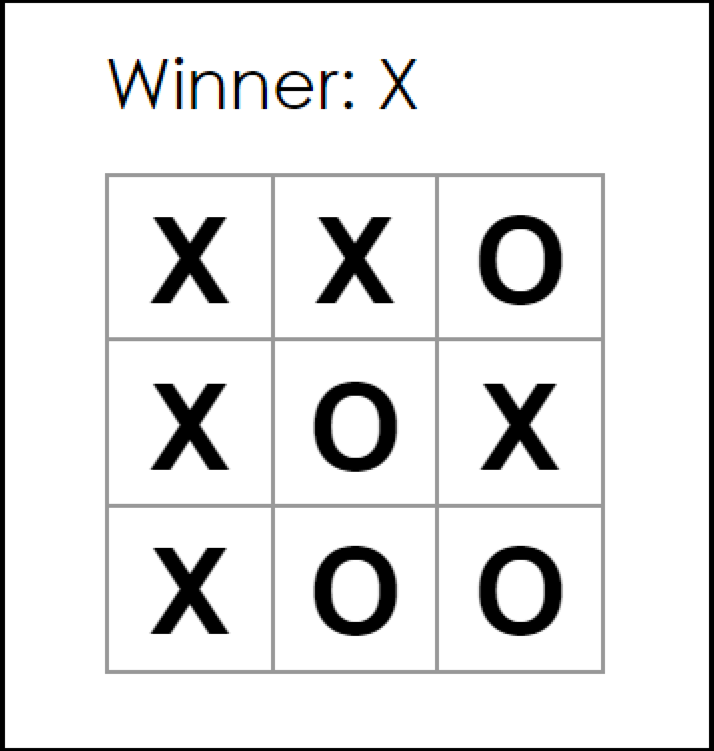
- You can find the presentation code in the repository:
  - https://github.com/andrebpradof/react-course

## What is React?

- React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

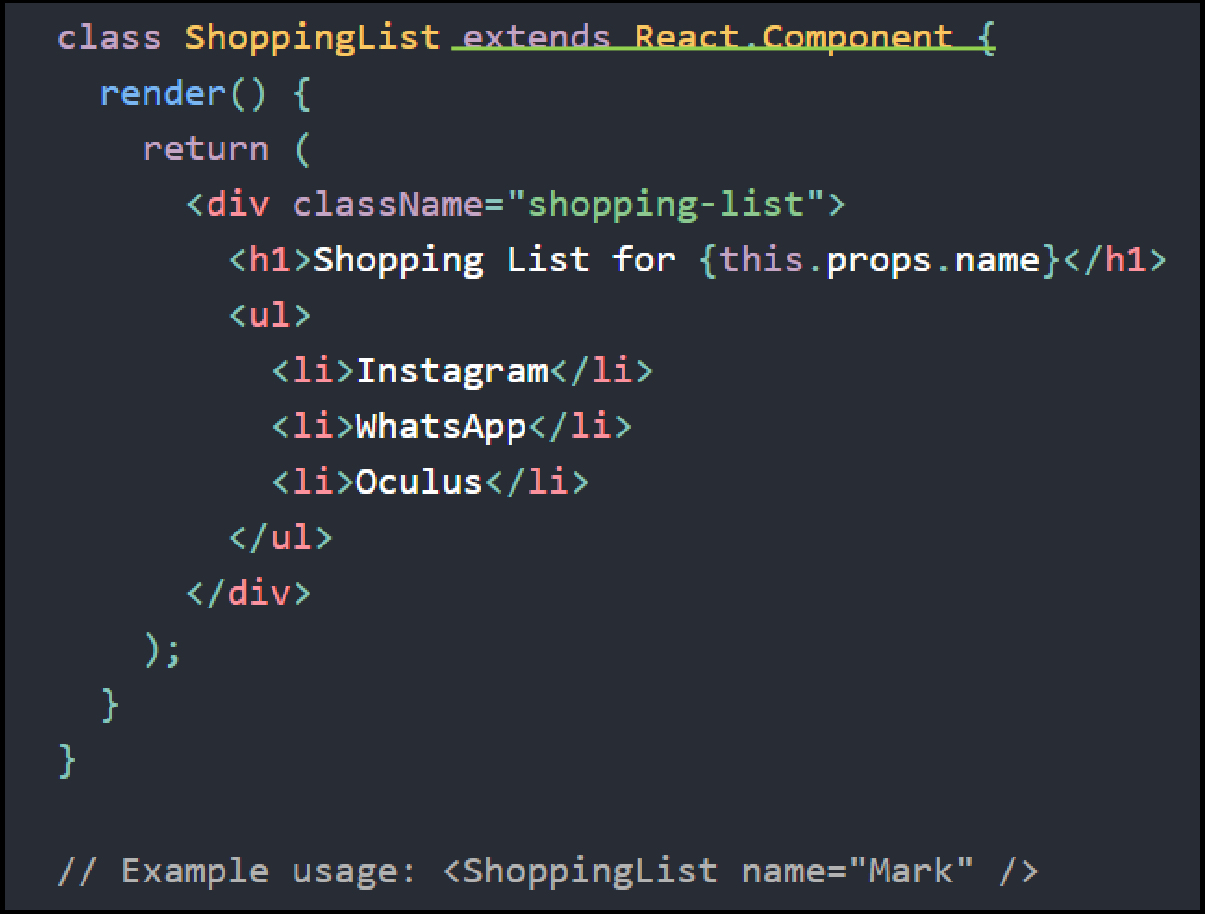
## What are we going to build?

- A Tic-Tac-Toe game



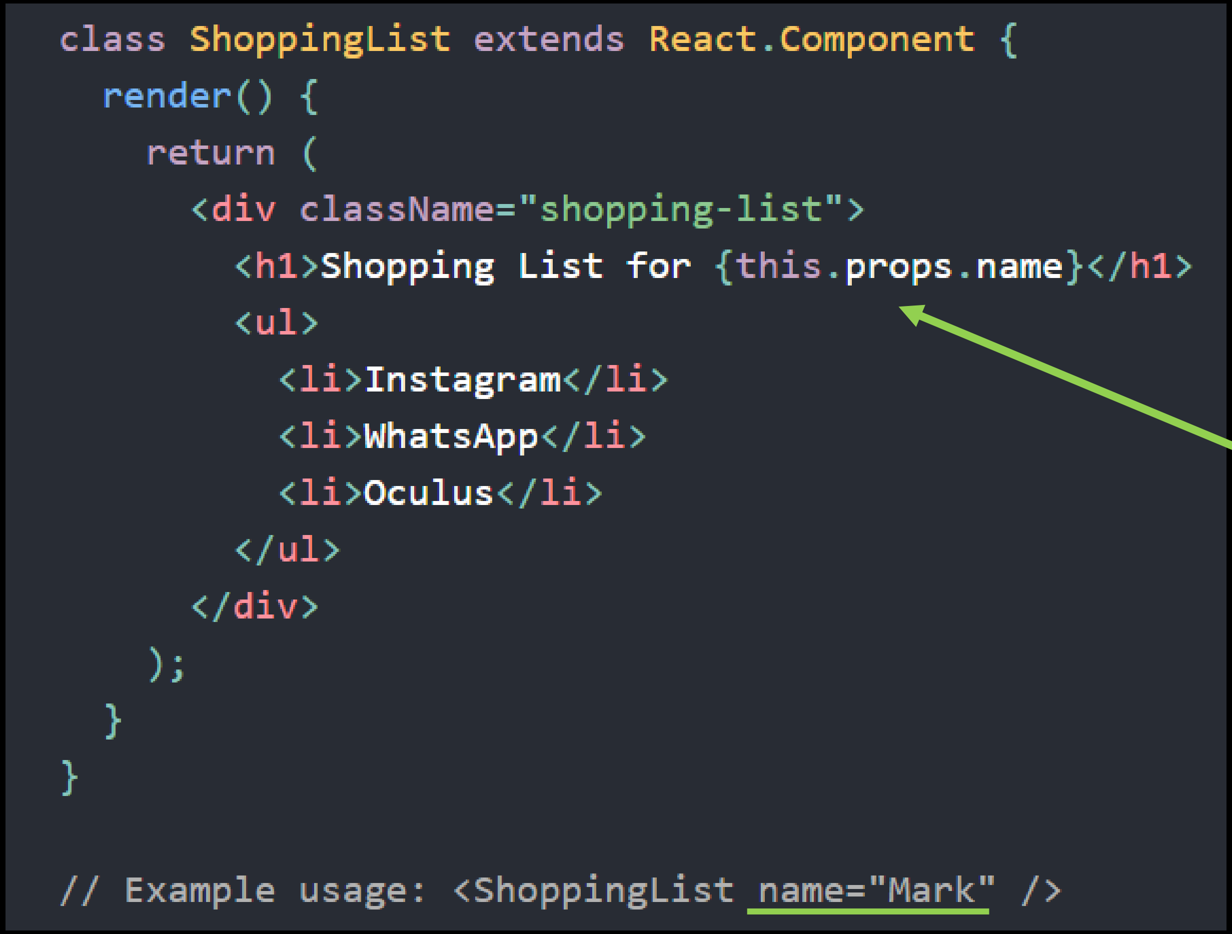
## What do we need to know to get started?

- A component should extends React.Component:



```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

## What do we need to know to get started?



```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

- A component takes in parameters, called props (short for "properties"), and returns a hierarchy of views to display via the render method.

## What do we need to know to get started?

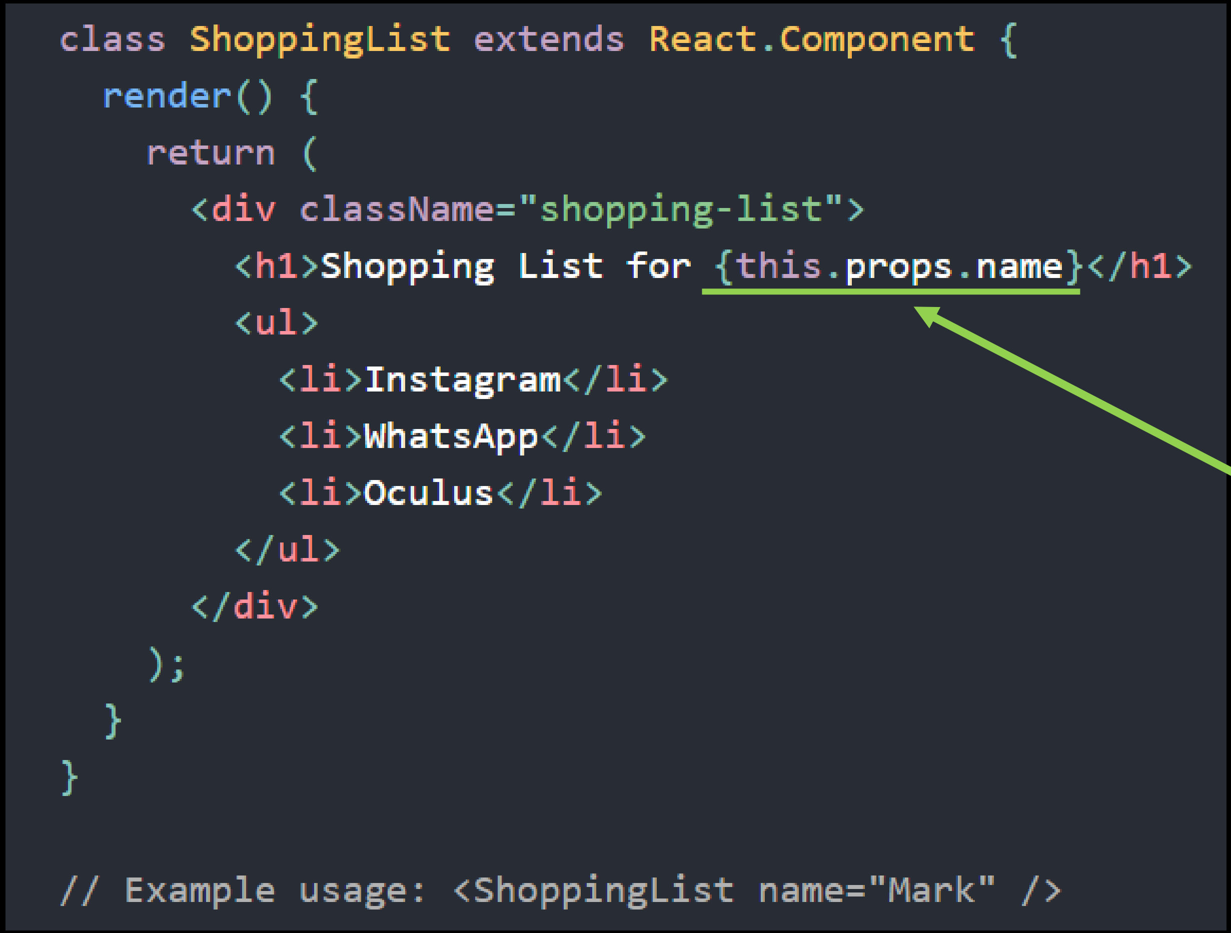
- The render method returns a description of what you want to see on the screen. React takes the description and displays the result. In particular, render returns a React element, which is a lightweight description of what to render.

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
```

- The <div /> syntax is transformed at build time to React.createElement('div’). This syntax is called “JSX”

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */)
);
```

# What do we need to know to get started?



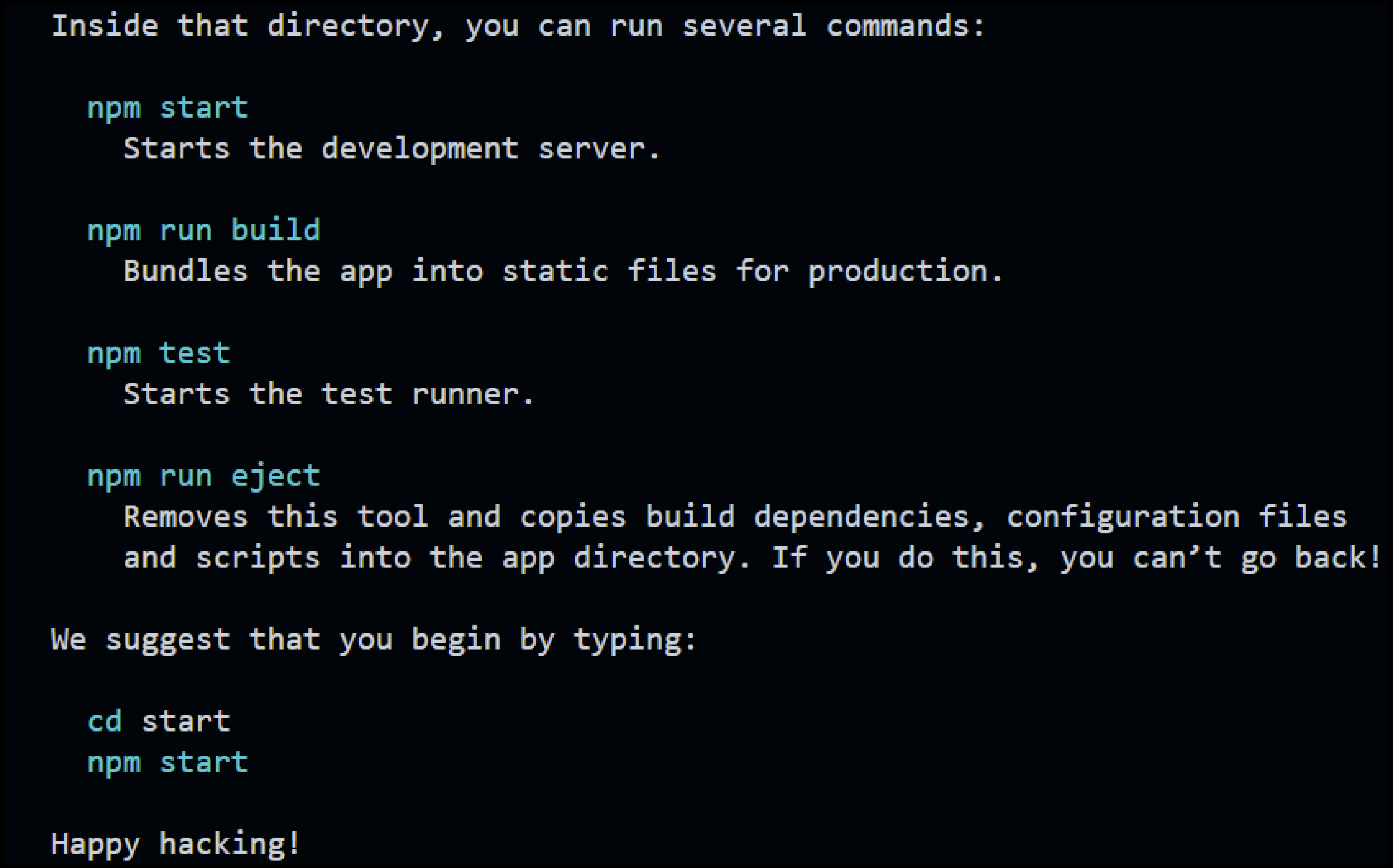
```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

- You can put any JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.

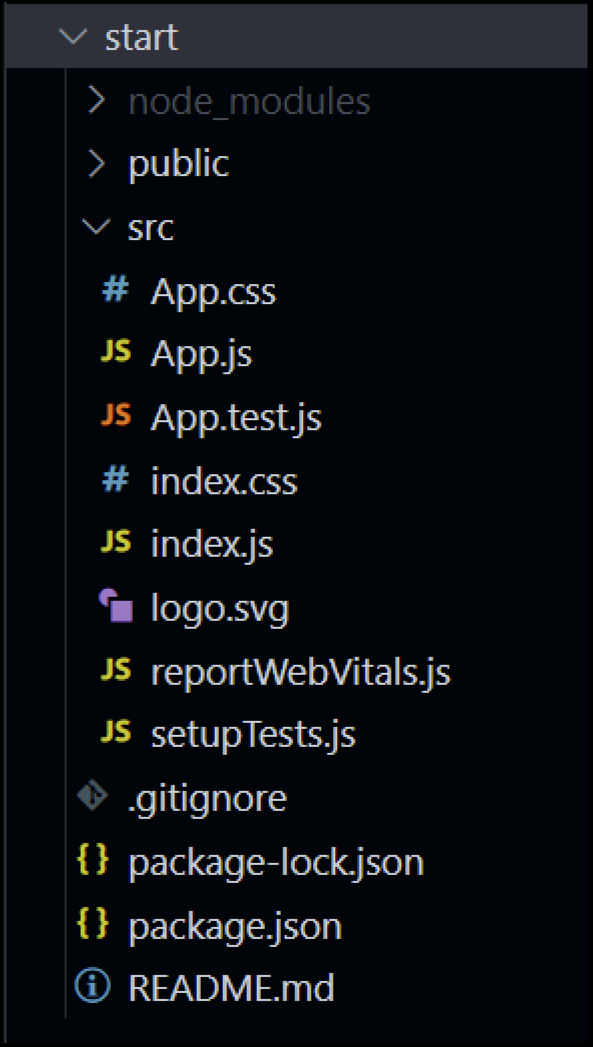
## Let's program

## Starting a new application

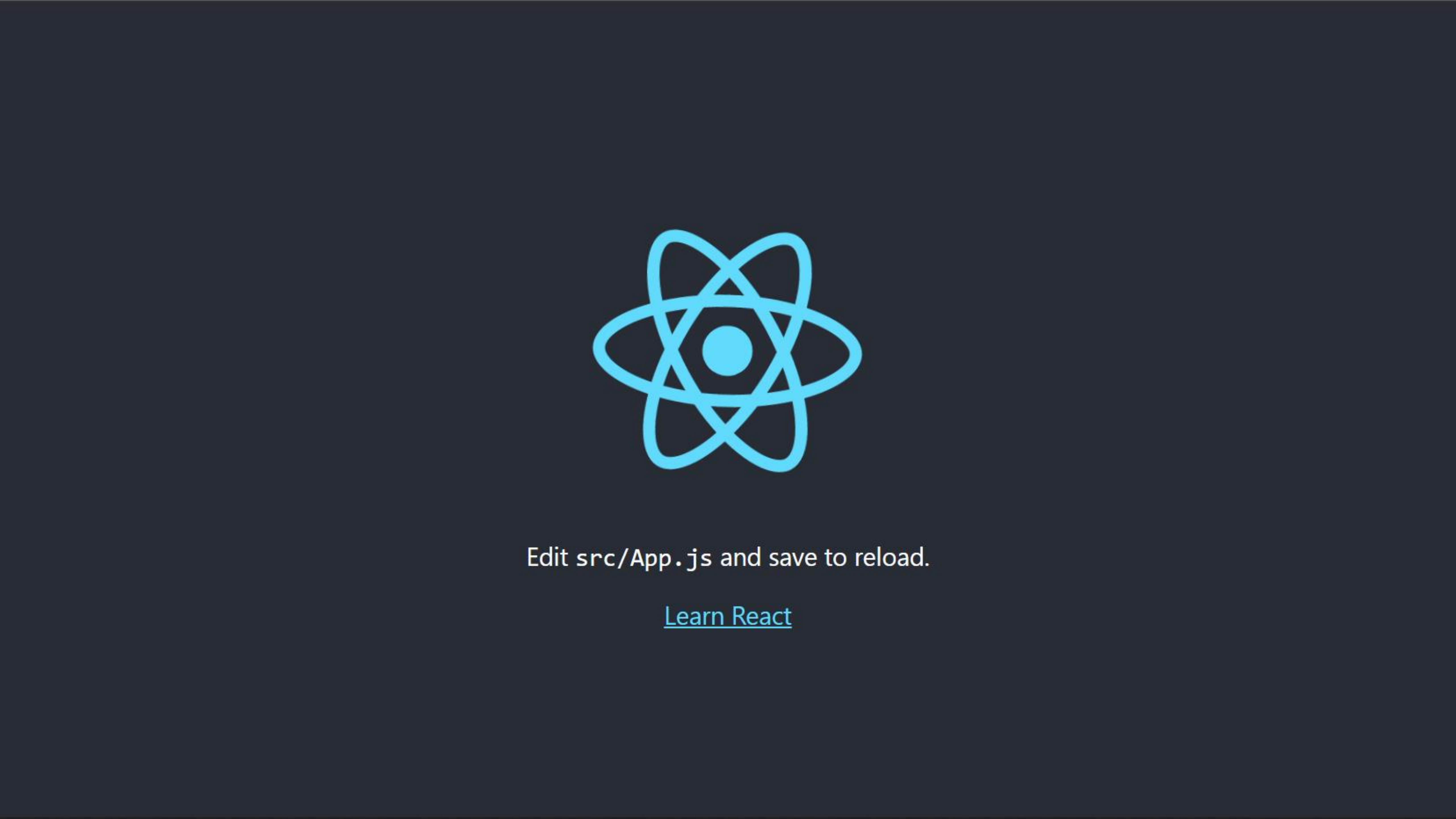
- > npx create-react-app [app-name]



```
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd start
  npm start

Happy hacking!
```



```
start
  node_modules
  public
  src
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

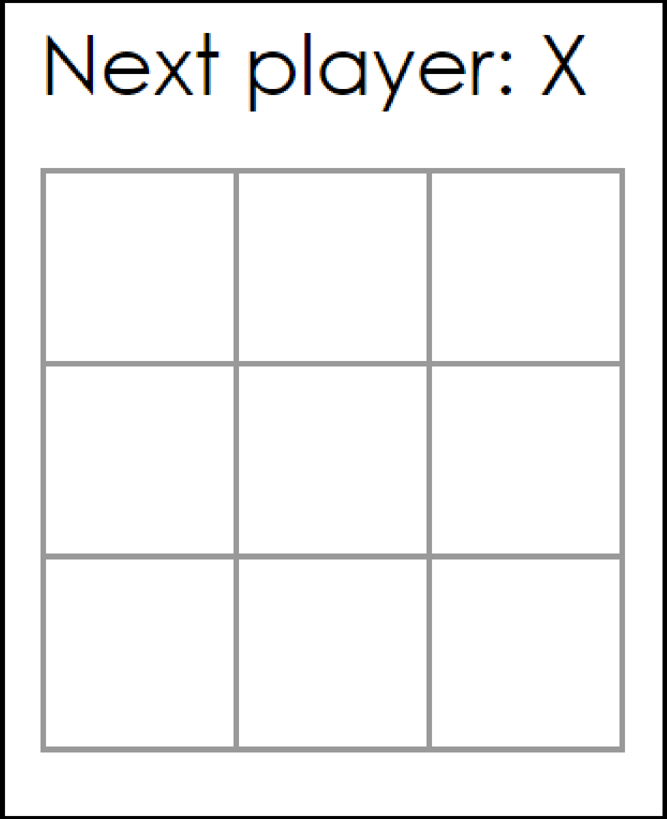


Edit `src/App.js` and save to reload.

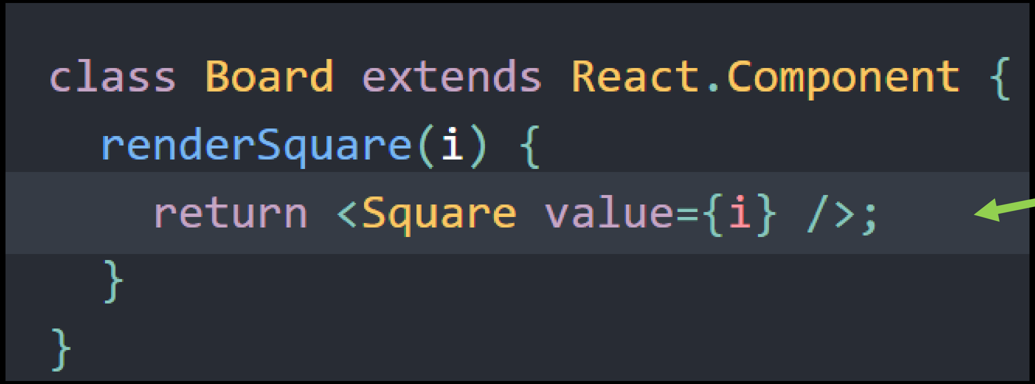
Learn React

## Building the application

- The Square component renders a single <button> and the Board renders 9 squares.
- The Game component renders a board with placeholder values which we'll modify later. There are currently no interactive components.


Next player: X

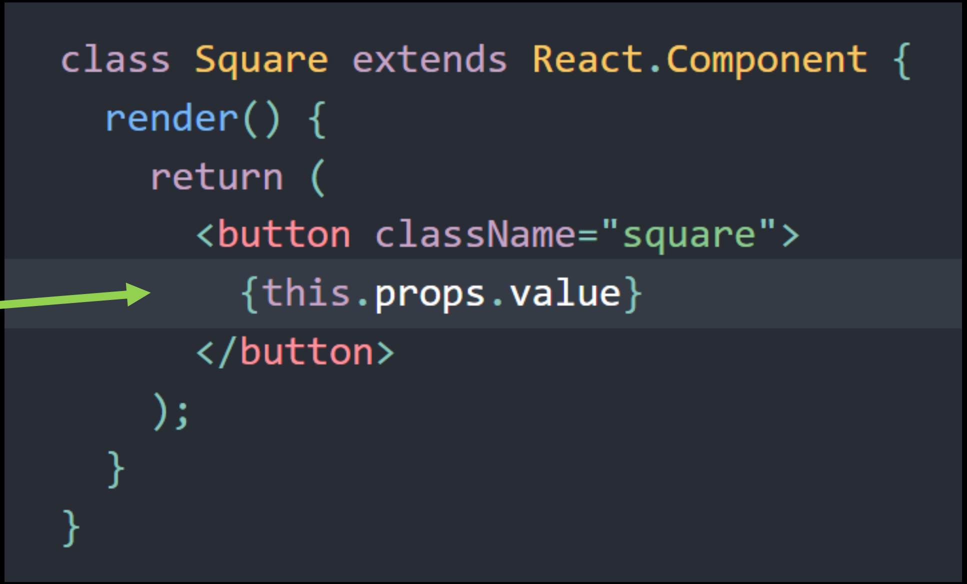
# Passing Data Through Props



```
class Board extends React.Component {
  renderSquare(i) {
    return <Square value={i} />;
  }
}
```

- In Board’s renderSquare method, change the code to pass a prop called value to the Square:

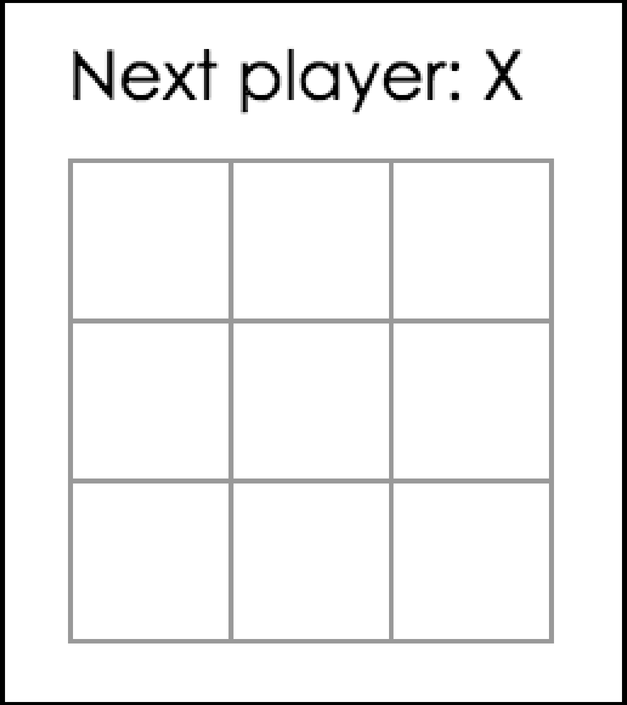
- Change Square’s render method to show that value by replacing {/* TODO */} with {this.props.value}:



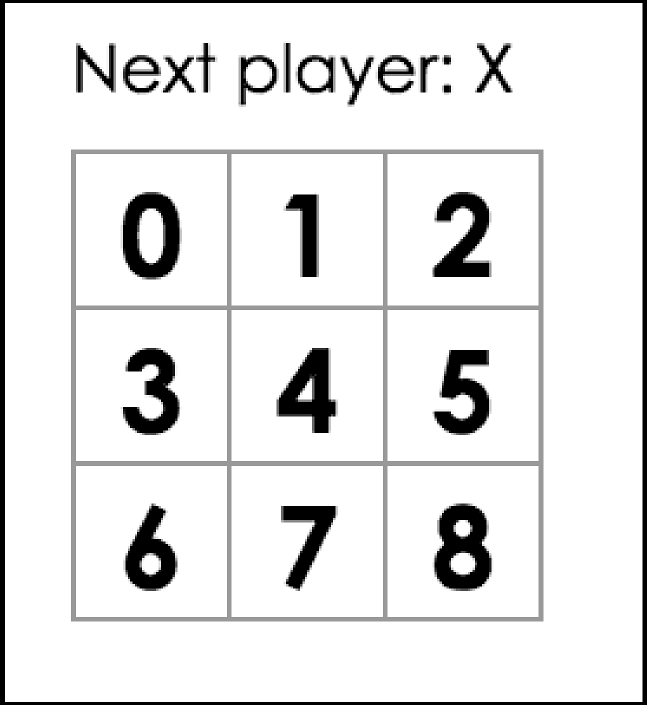
```
class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {this.props.value}
      </button>
    );
  }
}
```

## Building the application

**Before:**


Next player: X

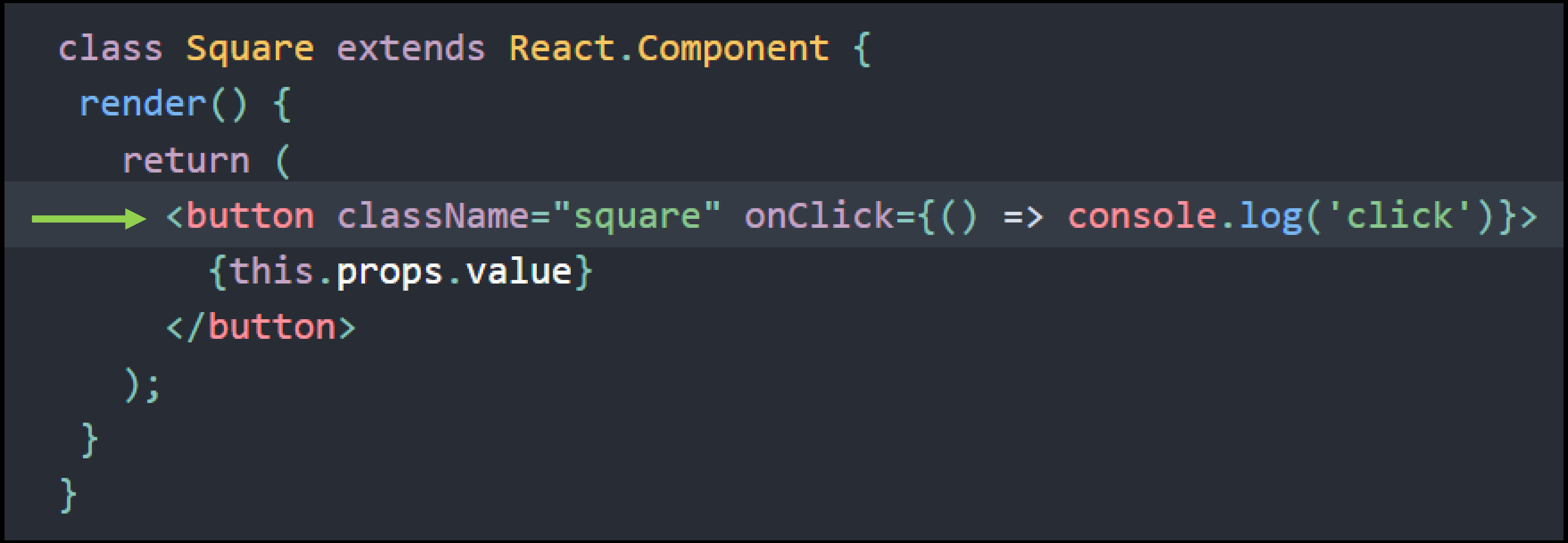
**After:**


Next player: X

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

## Making an Interactive Component

Let's fill the Square component with an "X" when we click it. First, change the button tag that is returned from the Square component's render() function to this:



```
class Square extends React.Component {
 render() {
   return (
     <button className="square" onClick={() => console.log('click')}>
       {this.props.value}
     </button>
   );
 }
}
```

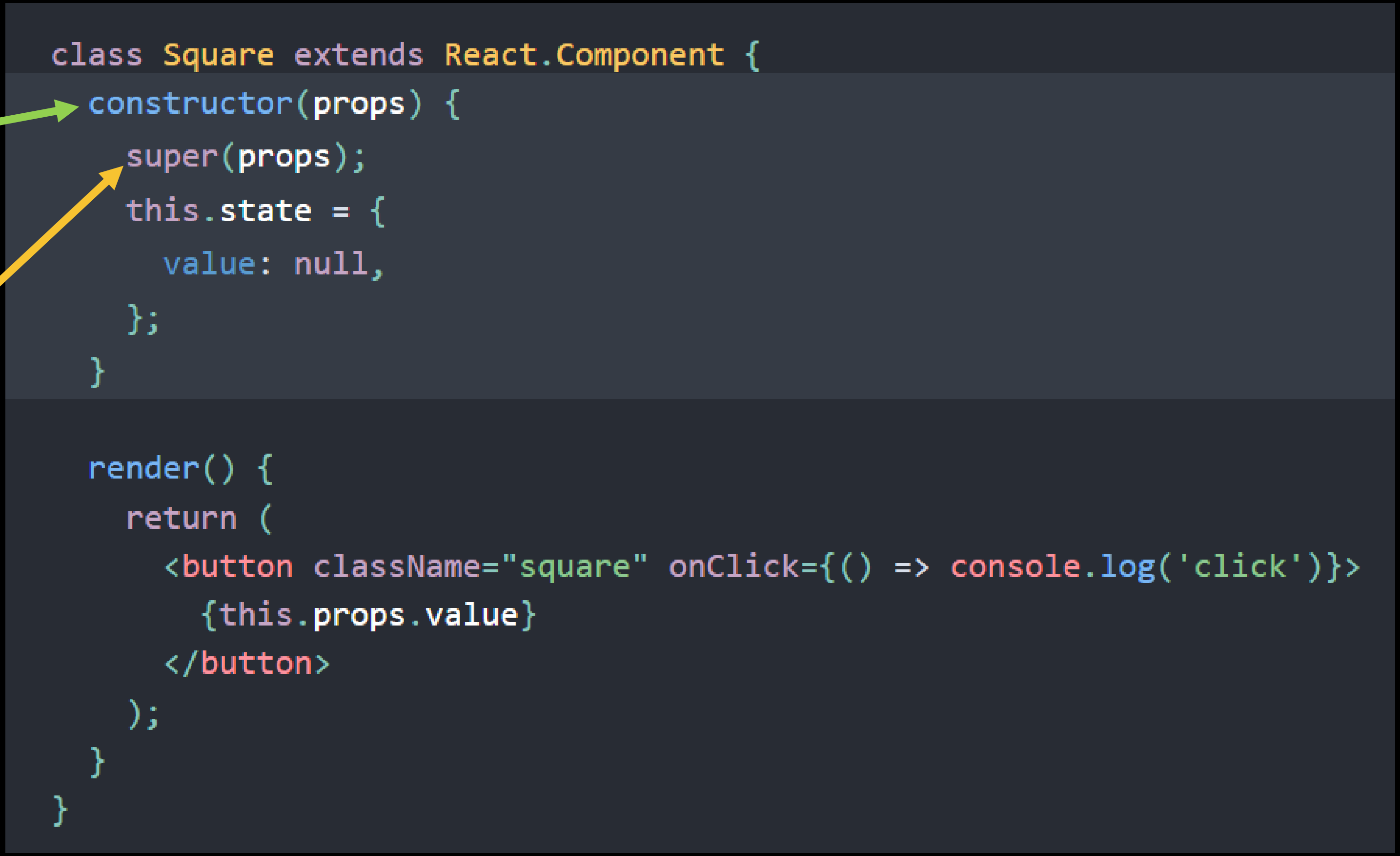
## Making an Interactive Component

- As a next step, we want the Square component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things, components use state.
- React components can have state by setting this.state in their constructors. this.state should be considered as private to a React component that it’s defined in. Let’s store the current value of the Square in this.state, and change it when the Square is clicked.

## Making an Interactive Component

- First, we'll add a constructor to the class to initialize the state:

In JavaScript classes, you need to always call super when defining the constructor of a subclass. All React component classes that have a constructor should start with a super(props) call.

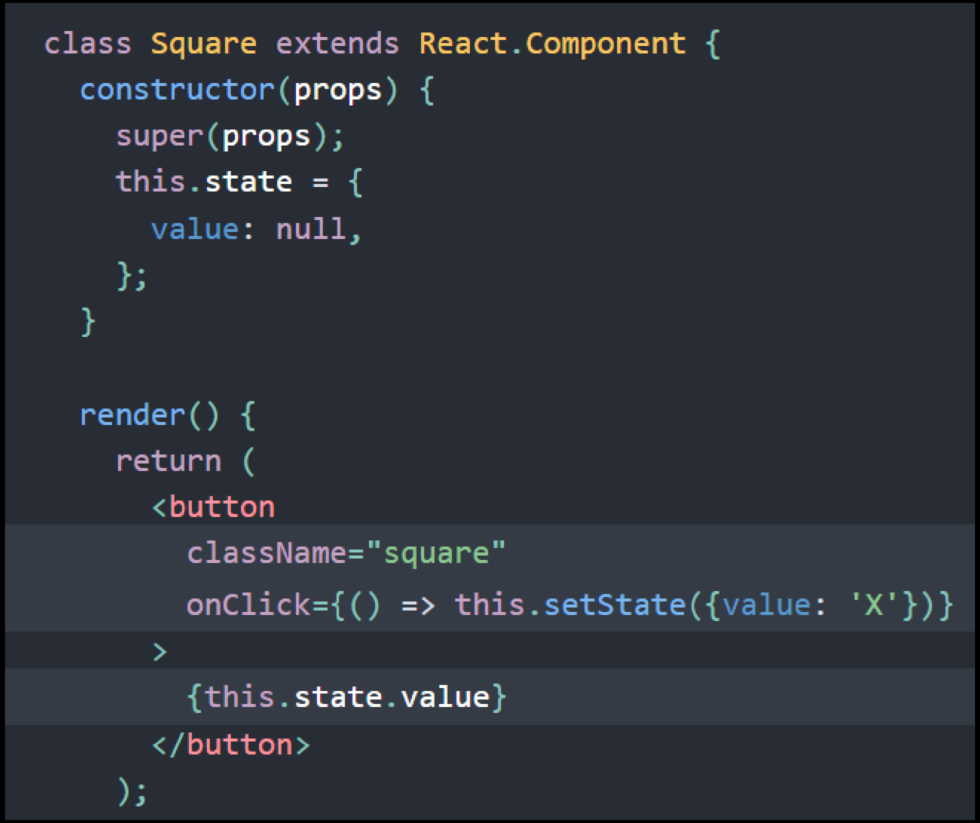


```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button className="square" onClick={() => console.log('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

# Making an Interactive Component

Now we'll change the Square's render method to display the current state's value when clicked:

- Replace this.props.value with this.state.value inside the <button> tag.
- Replace the onClick={...} event handler with onClick={() => this.setState({value: 'X'})}.
- Put the className and onClick props on separate lines for better readability.



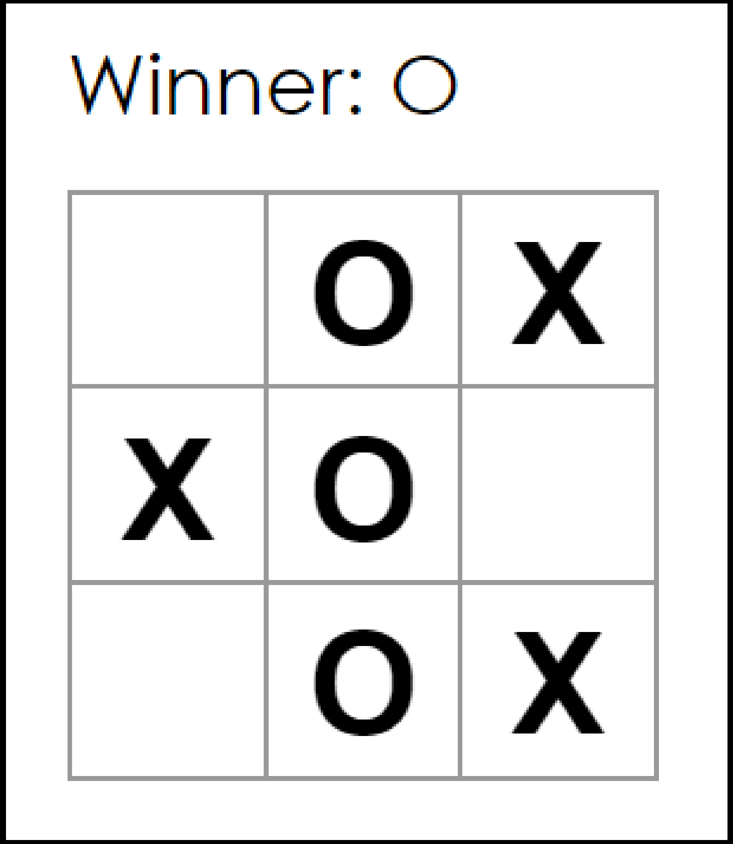
```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
```

## Making an Interactive Component

- By calling this.setState from an onClick handler in the Square's render method, we tell React to re-render that Square whenever its <button> is clicked. After the update, the Square's this.state.value will be 'X', so we'll see the X on the game board. If you click on any Square, an X should show up.
- When you call setState in a component, React automatically updates the child components inside of it too.

## Completing the Game

- We now have the basic building blocks for our tic-tac-toe game. To have a complete game, we now need to alternate placing "X"s and "O"s on the board, and we need a way to determine a winner.



## Lifting State Up

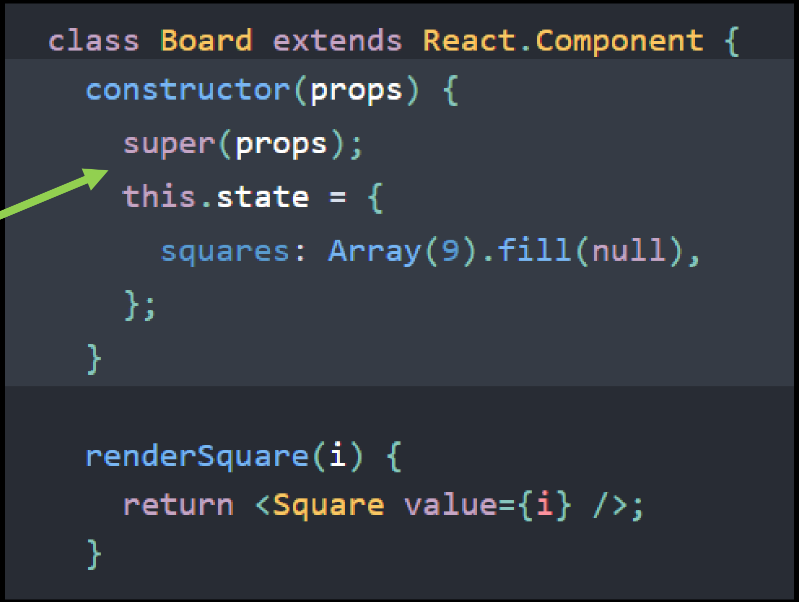
- Currently, each Square component maintains the game’s state. To check for a winner, we’ll maintain the value of each of the 9 squares in one location.
- We may think that Board should just ask each Square for the Square’s state. Although the best approach is to store the game’s state in the parent Board component instead of in each Square. The Board component can tell each Square what to display by passing a prop.

## Lifting State Up

- To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.

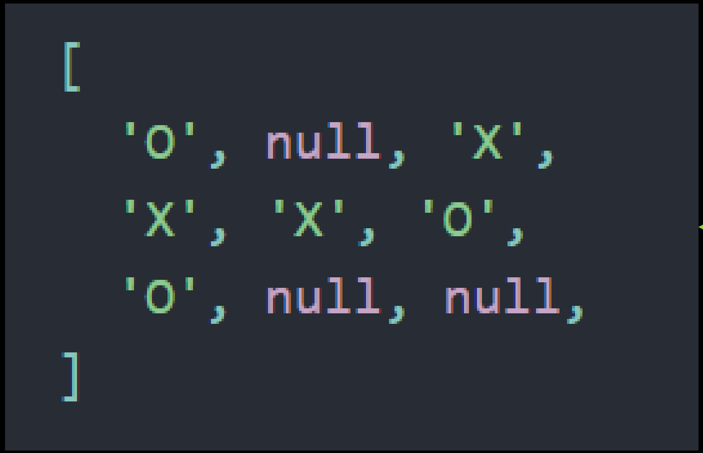
## Lifting State Up

- Add a constructor to the Board and set the Board's initial state to contain an array of 9 nulls corresponding to the 9 squares:



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  renderSquare(i) {
    return <Square value={i} />;
  }
```

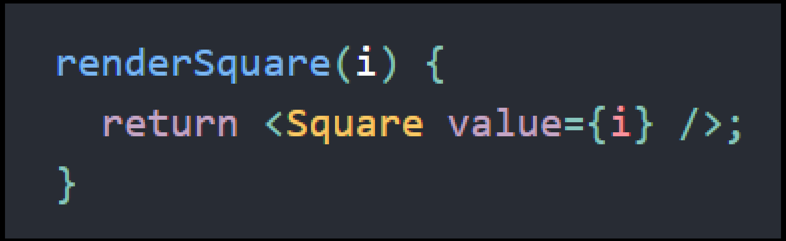
- When we fill the board in later, the this.state.squares array will look something like this:



```
[
  'O', null, 'X',
  'X', 'X', 'O',
  'O', null, null,
]
```

## Lifting State Up

- The Board’s renderSquare method currently looks like this:

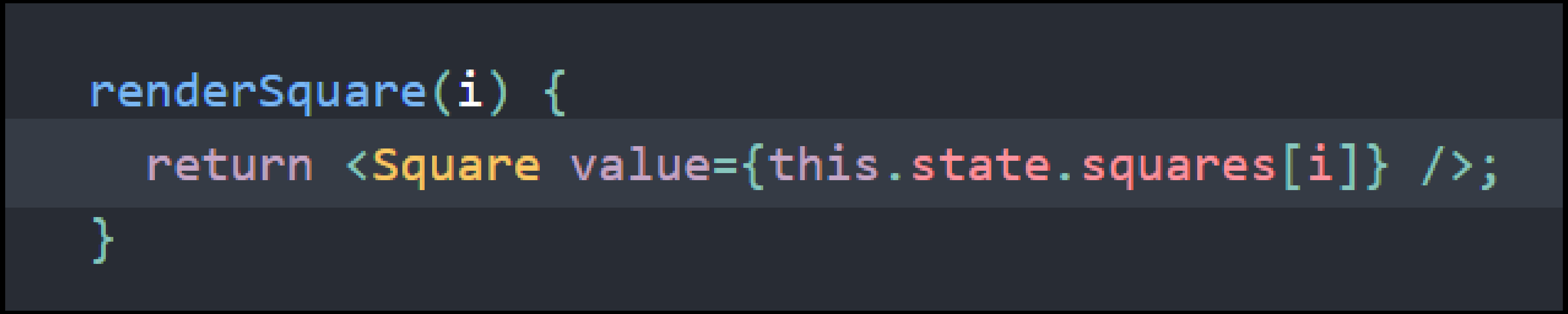


```
renderSquare(i) {
  return <Square value={i} />;
}
```

- In the beginning, we passed the value prop down from the Board to show numbers from 0 to 8 in every Square. In a different previous step, we replaced the numbers with an “X” mark determined by Square’s own state. This is why Square currently ignores the value prop passed to it by the Board.

## Lifting State Up

We will now use the prop passing mechanism again. We will modify the Board to instruct each individual Square about its current value ('X', 'O', or null). We have already defined the squares array in the Board's constructor, and we will modify the Board's renderSquare method to read from it:



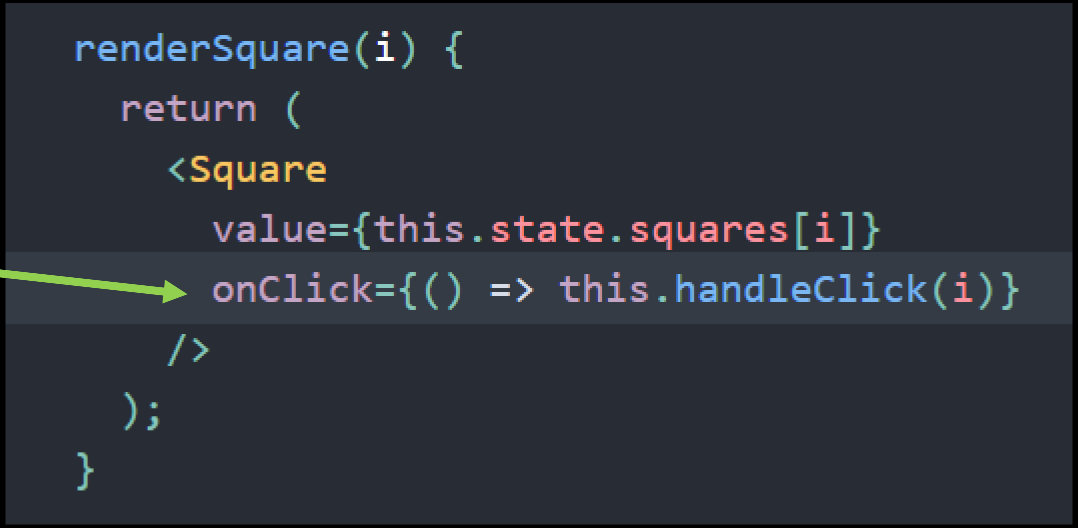
```
renderSquare(i) {
  return <Square value={this.state.squares[i]} />;
}
```

## Lifting State Up

- Each Square will now receive a value prop that will either be 'X', 'O', or null for empty squares.
- Next, we need to change what happens when a Square is clicked. The Board component now maintains which squares are filled. We need to create a way for the Square to update the Board’s state. Since state is considered to be private to a component that defines it, we cannot update the Board’s state directly from Square.

## Lifting State Up

- Instead, we'll pass down a function from the Board to the Square, and we'll have Square call that function when a square is clicked. We'll change the renderSquare method in Board to:

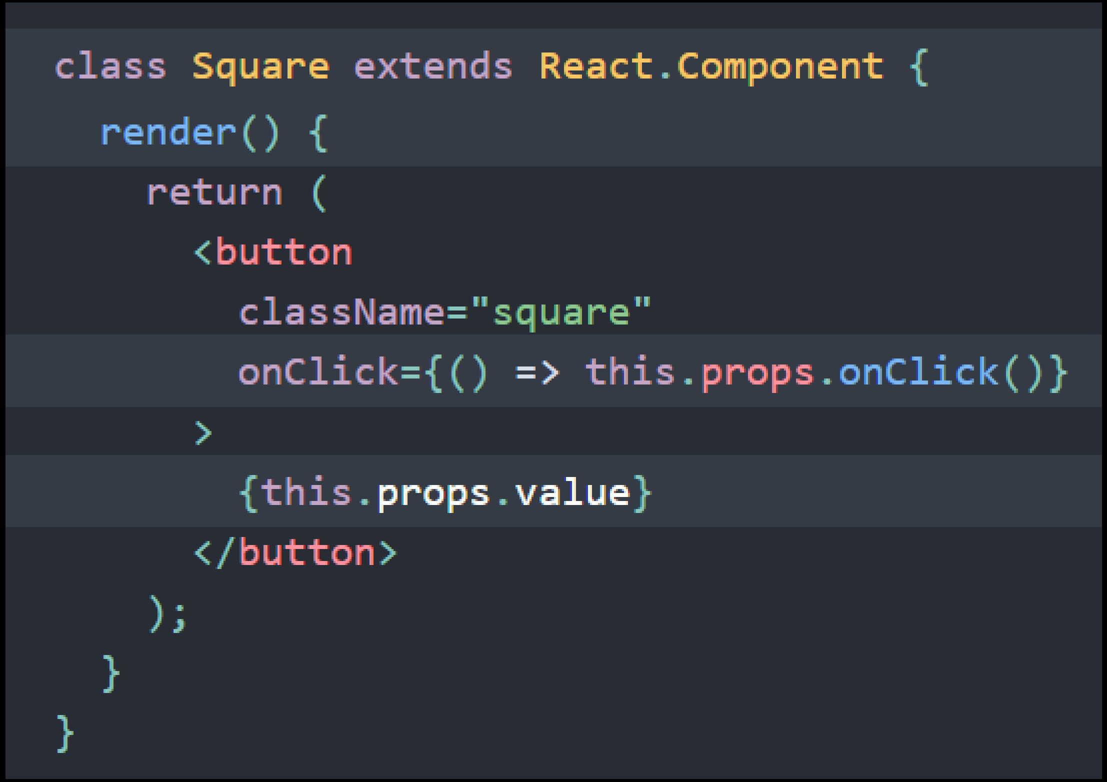


```
renderSquare(i) {
  return (
    <Square
      value={this.state.squares[i]}
      onClick={() => this.handleClick(i)}
    />
  );
}
```

## Lifting State Up

- Now we're passing down two props from Board to Square: value and onClick. The onClick prop is a function that Square can call when clicked. We'll make the following changes to Square:

- Replace this.state.value with this.props.value in Square's render method
- Replace this.setState() with this.props.onClick() in Square's render method
- Delete the constructor from Square because Square no longer keeps track of the game's state



```
class Square extends React.Component {
  render() {
    return (
      <button
        className="square"
        onClick={() => this.props.onClick()}
      >
        {this.props.value}
      </button>
    );
  }
}
```

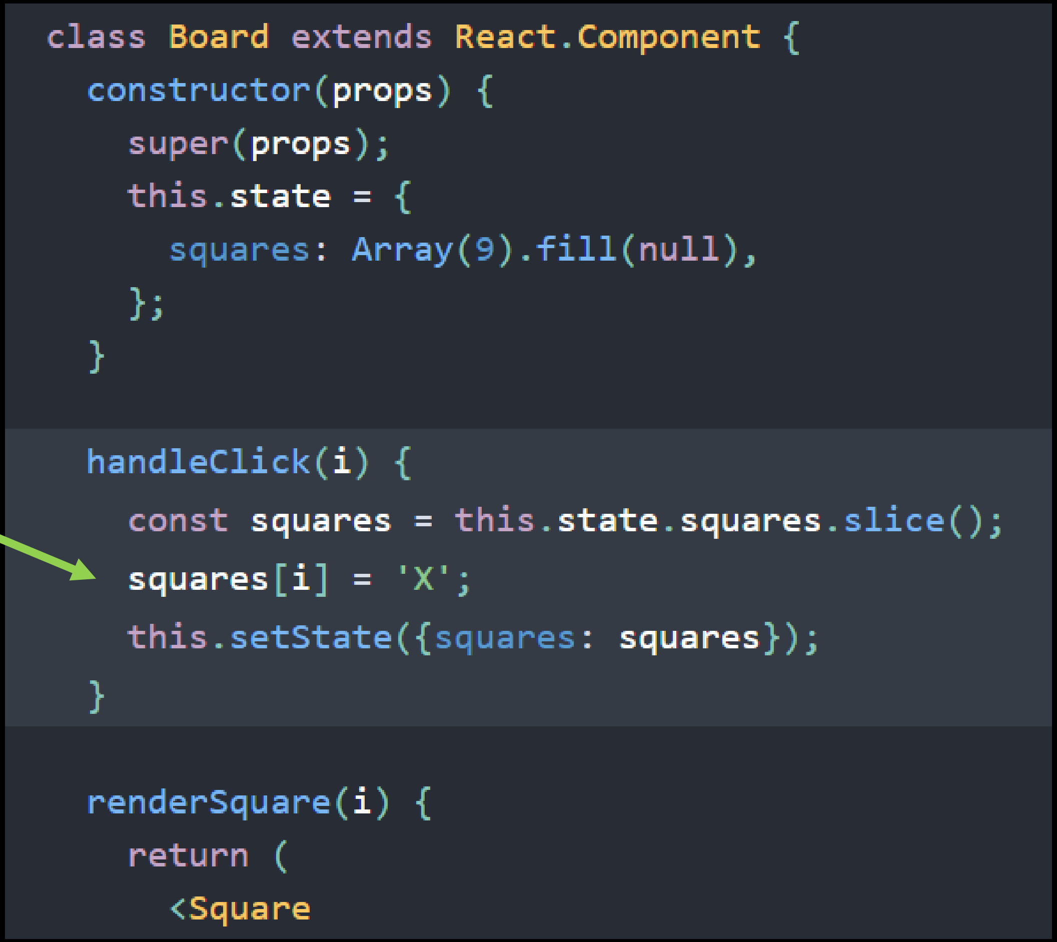
## Lifting State Up

- When a Square is clicked, the onClick function provided by the Board is called. Here's a review of how this is achieved:

1. The onClick prop on the built-in DOM <button> component tells React to set up a click event listener.
2. When the button is clicked, React will call the onClick event handler that is defined in Square's render() method.
3. This event handler calls this.props.onClick(). The Square's onClick prop was specified by the Board.
4. Since the Board passed onClick={() => this.handleClick(i)} to Square, the Square calls the Board's handleClick(i) when clicked.
5. We have not defined the handleClick() method yet, so our code crashes. If you click a square now, you should see a red error screen saying something like "this.handleClick is not a function".

## Lifting State Up

- When we try to click a Square, we should get an error because we haven’t defined handleClick yet. We’ll now add handleClick to the Board class:



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  handleClick(i) {
    const squares = this.state.squares.slice();
    squares[i] = 'X';
    this.setState({squares: squares});
  }

  renderSquare(i) {
    return (
      <Square
```

## Lifting State Up

- After these changes, we’re again able to click on the Squares to fill them, the same as we had before. However, now the state is stored in the Board component instead of the individual Square components. When the Board’s state changes, the Square components re-render automatically. Keeping the state of all squares in the Board component will allow it to determine the winner in the future.
- Since the Square components no longer maintain state, the Square components receive values from the Board component and inform the Board component when they’re clicked. In React terms, the Square components are now controlled components. The Board has full control over them.

## Function Components

- We'll now change the Square to be a function component.
- In React, function components are a simpler way to write components that only contain a render method and don't have their own state. Instead of defining a class which extends React.Component, we can write a function that takes props as input and returns what should be rendered. Function components are less tedious to write than classes, and many components can be expressed this way.

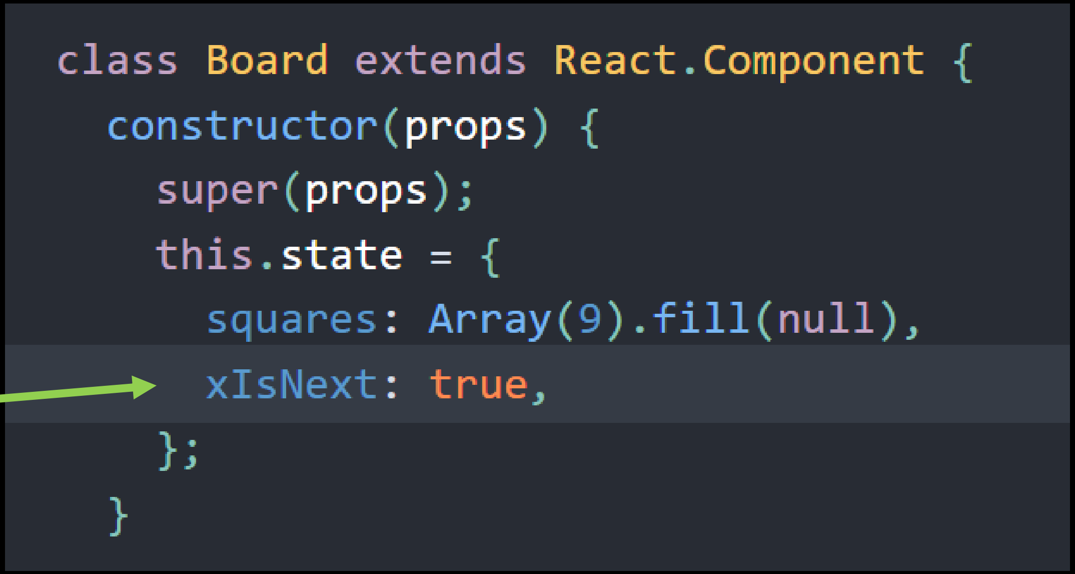
## Function Components

- Replace the Square class with this function:

```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

## Taking Turns

- We now need to fix an obvious defect in our tic-tac-toe game: the “O”s cannot be marked on the board.
- We’ll set the first move to be “X” by default. We can set this default by modifying the initial state in our Board constructor:



```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
      xIsNext: true,
    };
  }
```

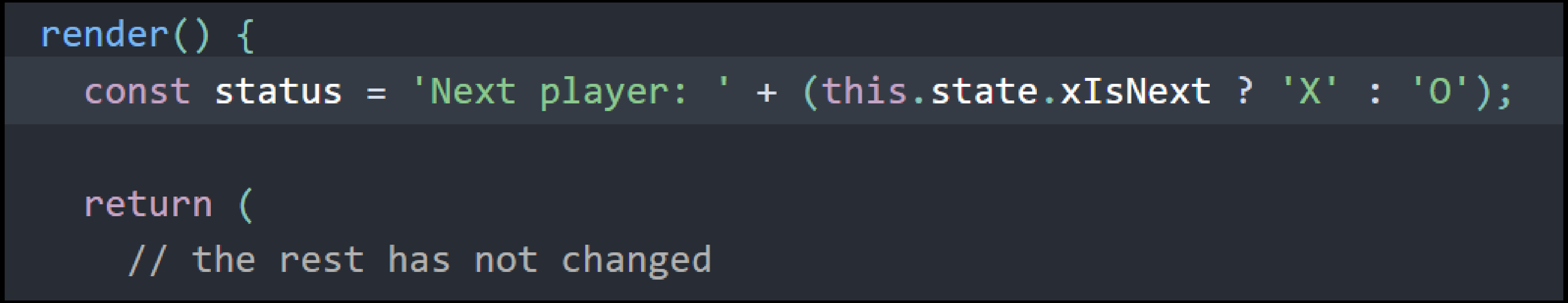
## Taking Turns

Each time a player moves, xIsNext (a boolean) will be flipped to determine which player goes next and the game's state will be saved. We'll update the Board's handleClick function to flip the value of xIsNext:

```
handleClick(i) {
  const squares = this.state.squares.slice();
  squares[i] = this.state.xIsNext ? 'X' : 'O';
  this.setState({
    squares: squares,
    xIsNext: !this.state.xIsNext,
  });
}
```

## Taking Turns

- Let’s also change the “status” text in Board’s render so that it displays which player has the next turn:



```
render() {
  const status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');

  return (
    // the rest has not changed
```

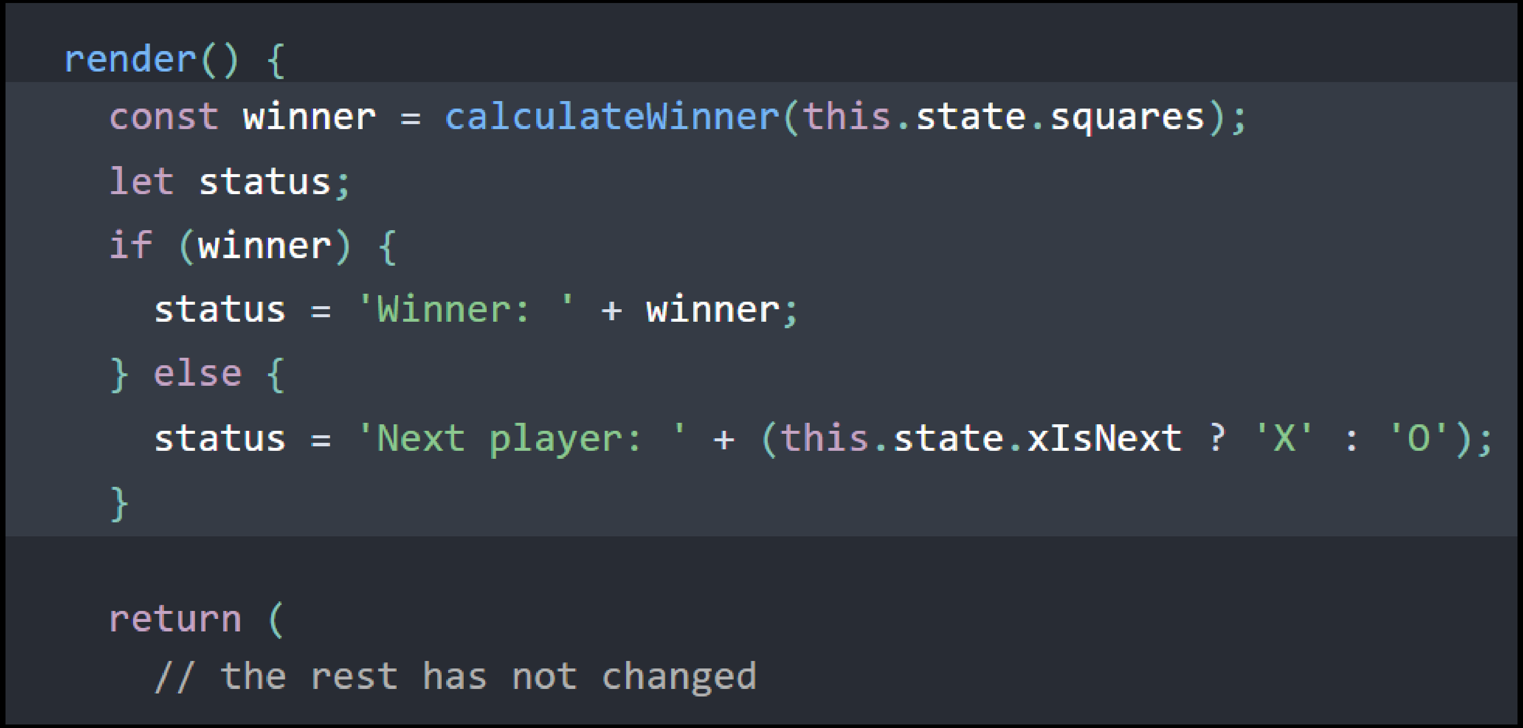
## Declaring a Winner

- Now that we show which player’s turn is next, we should also show when the game is won and there are no more turns to make. Copy this helper function and paste it at the end of the file:

```
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

## Declaring a Winner

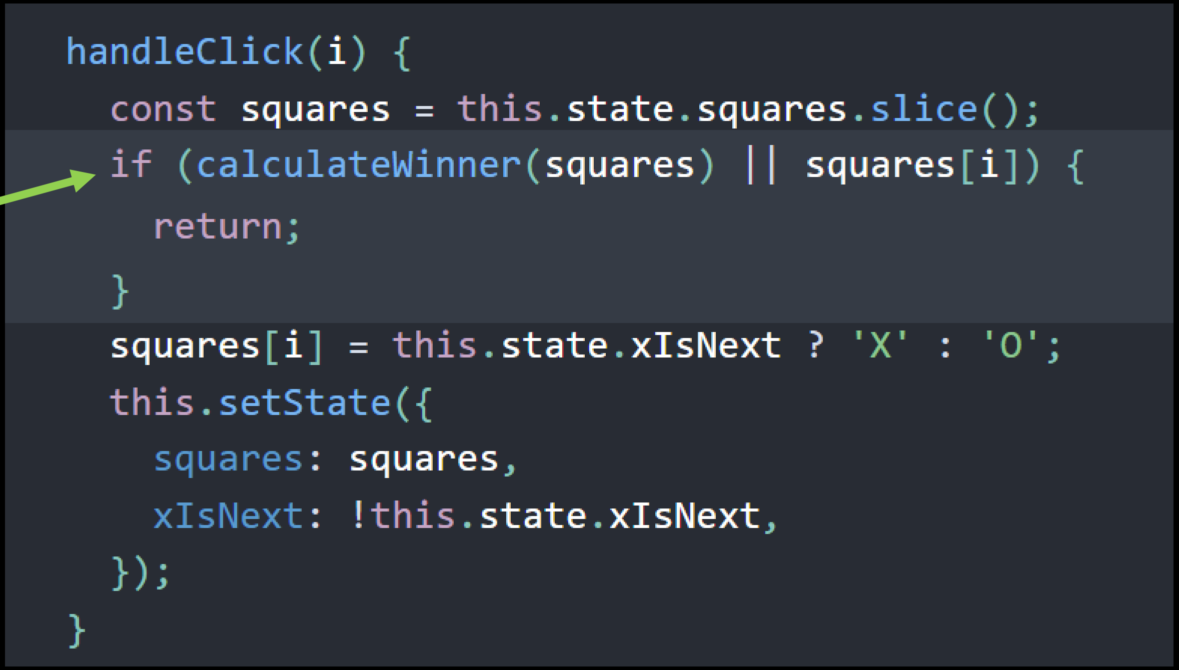
- Given an array of 9 squares, this function will check for a winner and return 'X', 'O', or null as appropriate.
- We will call calculateWinner (squares) in the Board's render function to check if a player has won. If a player has won, we can display text such as "Winner: X" or "Winner: O". We'll replace the status declaration in Board's render function with this code:



```
render() {
  const winner = calculateWinner(this.state.squares);
  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
  }

  return (
    // the rest has not changed
```

## Declaring a Winner

- We can now change the Board’s handleClick function to return early by ignoring a click if someone has won the game or if a Square is already filled:



```
handleClick(i) {
  const squares = this.state.squares.slice();
  if (calculateWinner(squares) || squares[i]) {
    return;
  }
  squares[i] = this.state.xIsNext ? 'X' : 'O';
  this.setState({
    squares: squares,
    xIsNext: !this.state.xIsNext,
  });
}
```

Congratulations! You now have a working tic-tac-toe game