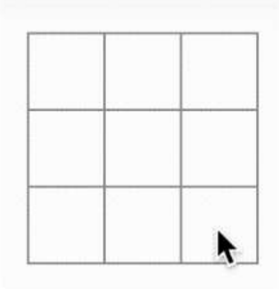
```
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
    </>
  );
}
```

Now you'll change `Square` to display an "X" when clicked. Replace the `console.log("clicked!");` event handler with `setValue('X');`. Now your `Square` component looks like this:

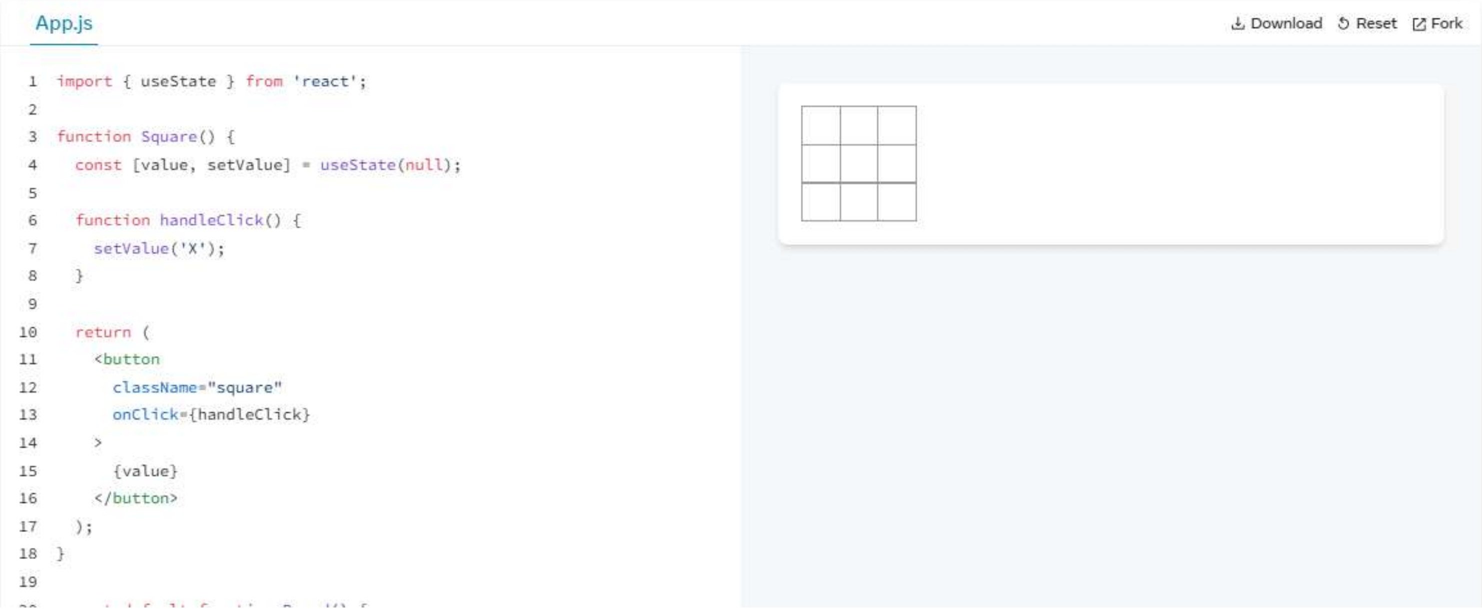
```
function Square() {
  const [value, setValue] = useState(null);

  function handleClick() {
    setValue('X');
  }

  return (
    <button
      className="square"
      onClick={handleClick}
    >
      {value}
    </button>
  );
}
```

By calling this `set` function from an `onClick` handler, you're telling React to re-render that `Square` whenever its `<button>` is clicked. After the update, the `Square`'s `value` will be `'X'`, so you'll see the "X" on the game board. Click on any Square, and "X" should show up:

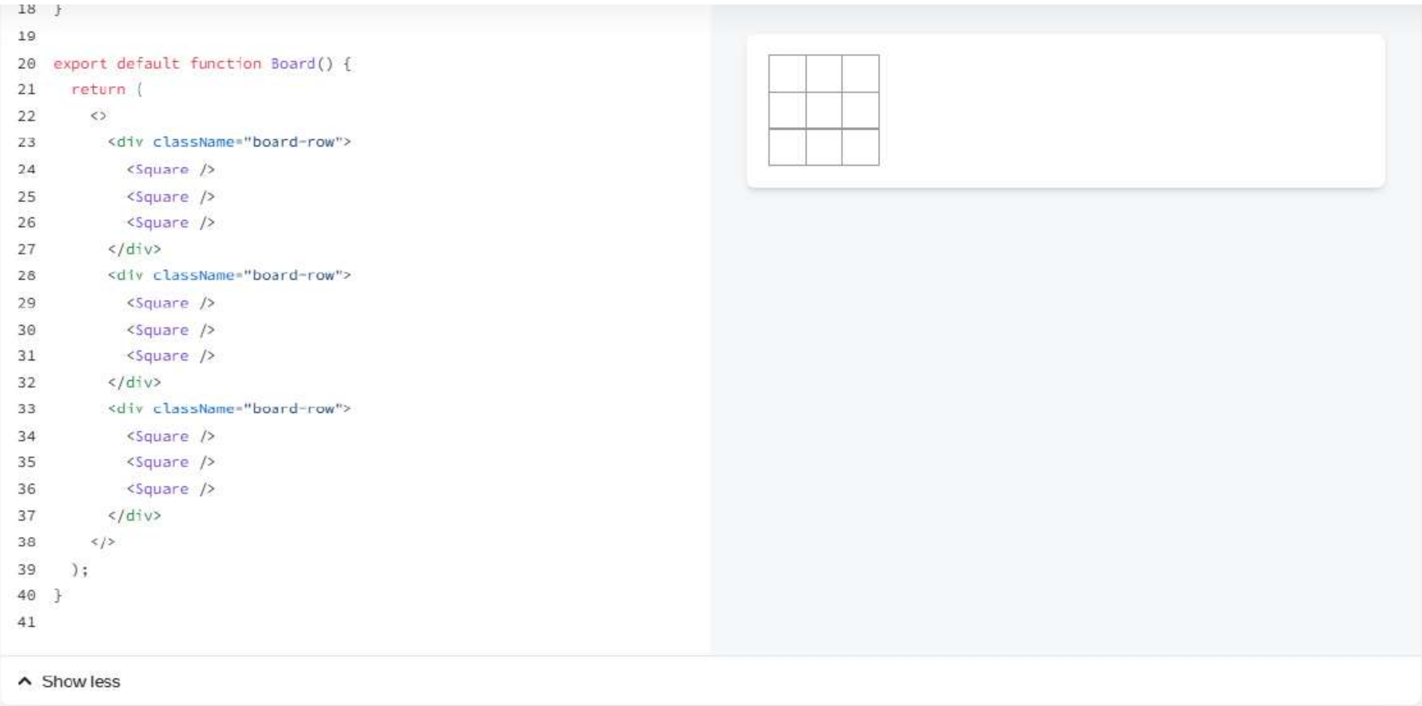


Each Square has its own state: the `value` stored in each Square is completely independent of the others. When you call a `set` function in a component, React automatically updates the child components inside too.

After you've made the above changes, your code will look like this:



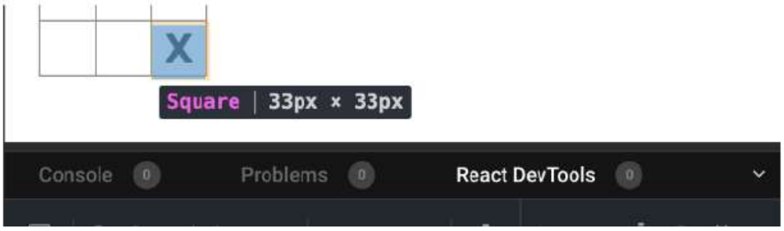
```
import { useState } from 'react';

function Square() {
  const [value, setValue] = useState(null);

  function handleClick() {
    setValue('X');
  }

  return (
    <button
      className="square"
      onClick={handleClick}
    >
      {value}
    </button>
  );
}

export default function Board() {
```

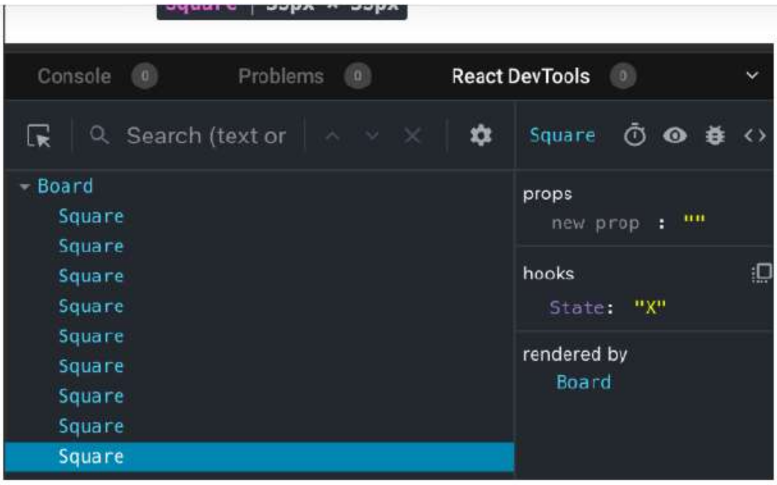


```
}

export default function Board() {
  return (
    <>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
    </>
  );
}
```

React Developer Tools

React DevTools let you check the props and the state of your React components. You can find the React DevTools tab at the bottom of the *browser* section in CodeSandbox:

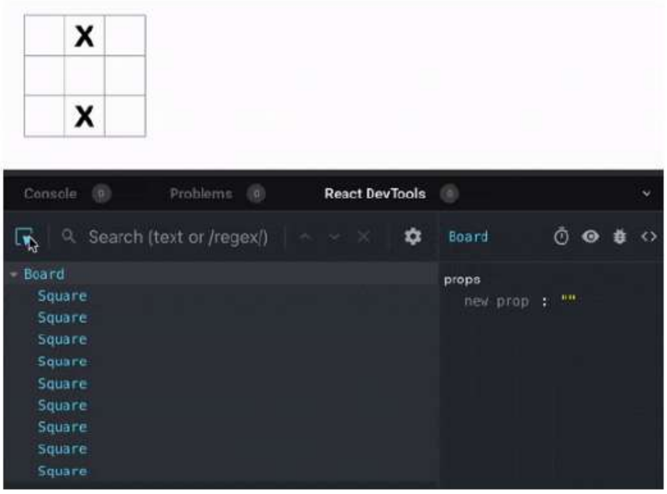




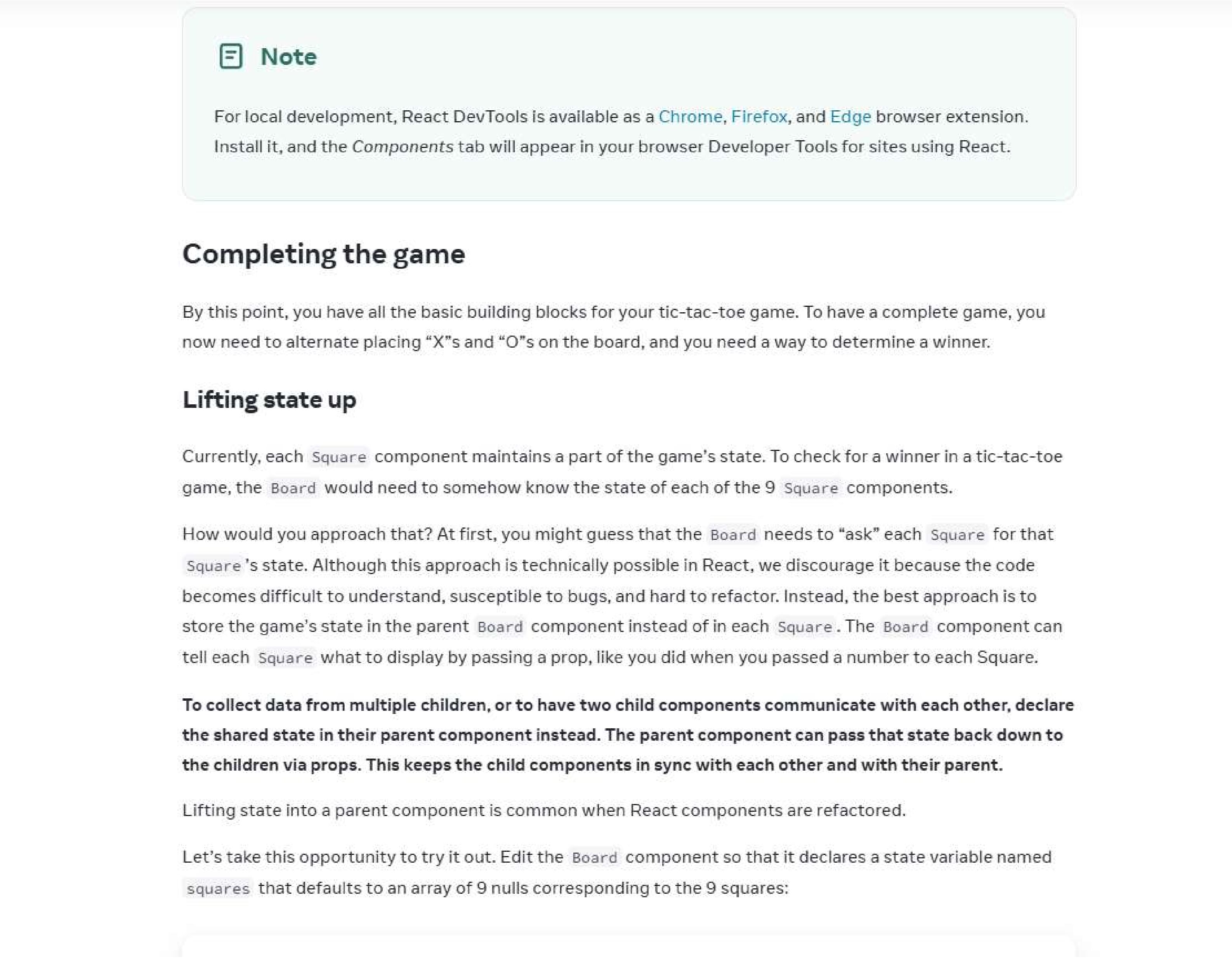
new prop : ""

State: "X"

To inspect a particular component on the screen, use the button in the top left corner of React DevTools:



new prop : ""



Note

For local development, React DevTools is available as a Chrome, Firefox, and Edge browser extension. Install it, and the *Components* tab will appear in your browser Developer Tools for sites using React.

Completing the game

By this point, you have all the basic building blocks for your tic-tac-toe game. To have a complete game, you now need to alternate placing “X”s and “O”s on the board, and you need a way to determine a winner.

Lifting state up

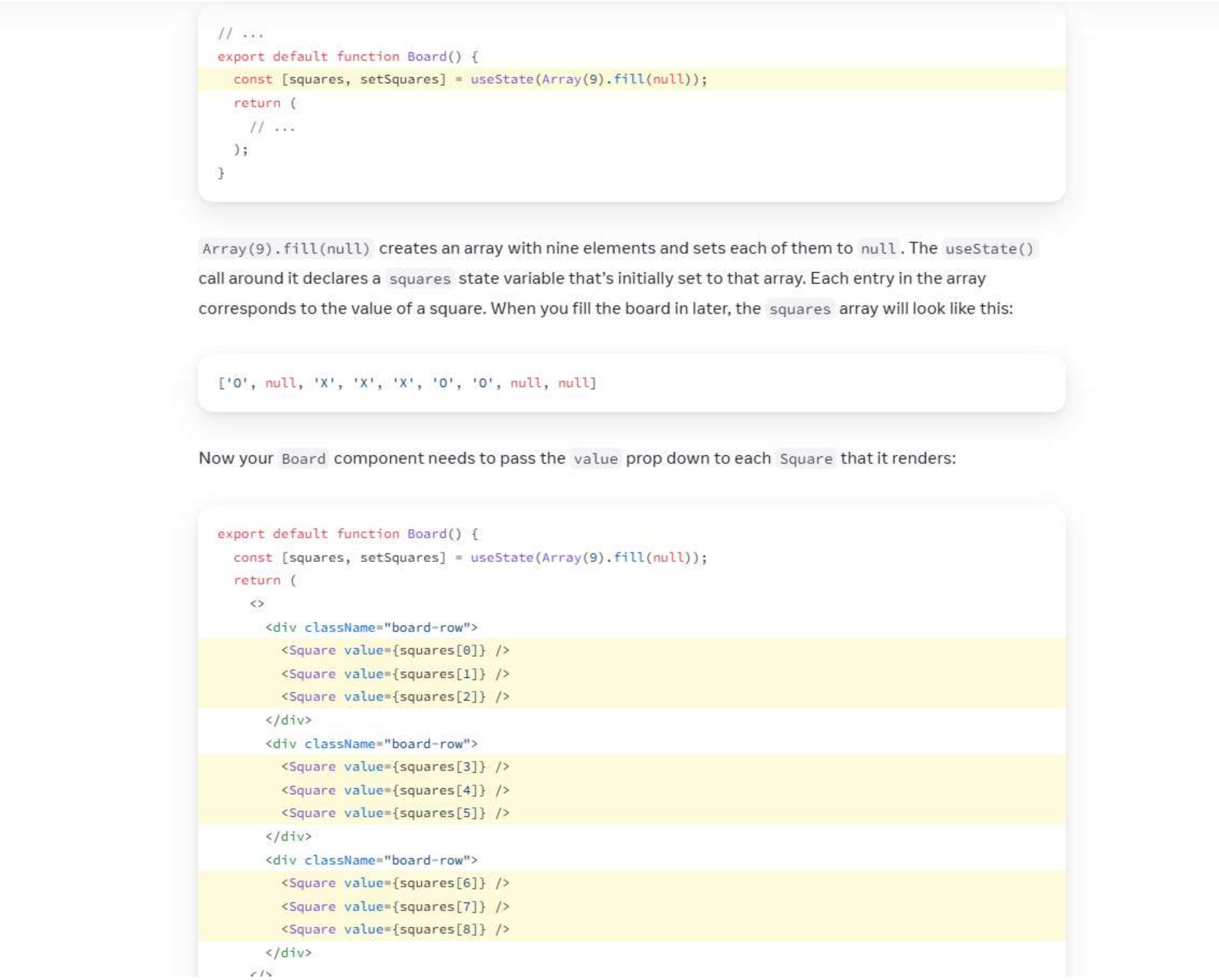
Currently, each `Square` component maintains a part of the game’s state. To check for a winner in a tic-tac-toe game, the `Board` would need to somehow know the state of each of the 9 `Square` components.

How would you approach that? At first, you might guess that the `Board` needs to “ask” each `Square` for that `Square`’s state. Although this approach is technically possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard to refactor. Instead, 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, like you did when you passed a number to each Square.

To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with their parent.

Lifting state into a parent component is common when React components are refactored.

Let’s take this opportunity to try it out. Edit the `Board` component so that it declares a state variable named `squares` that defaults to an array of 9 nulls corresponding to the 9 squares:



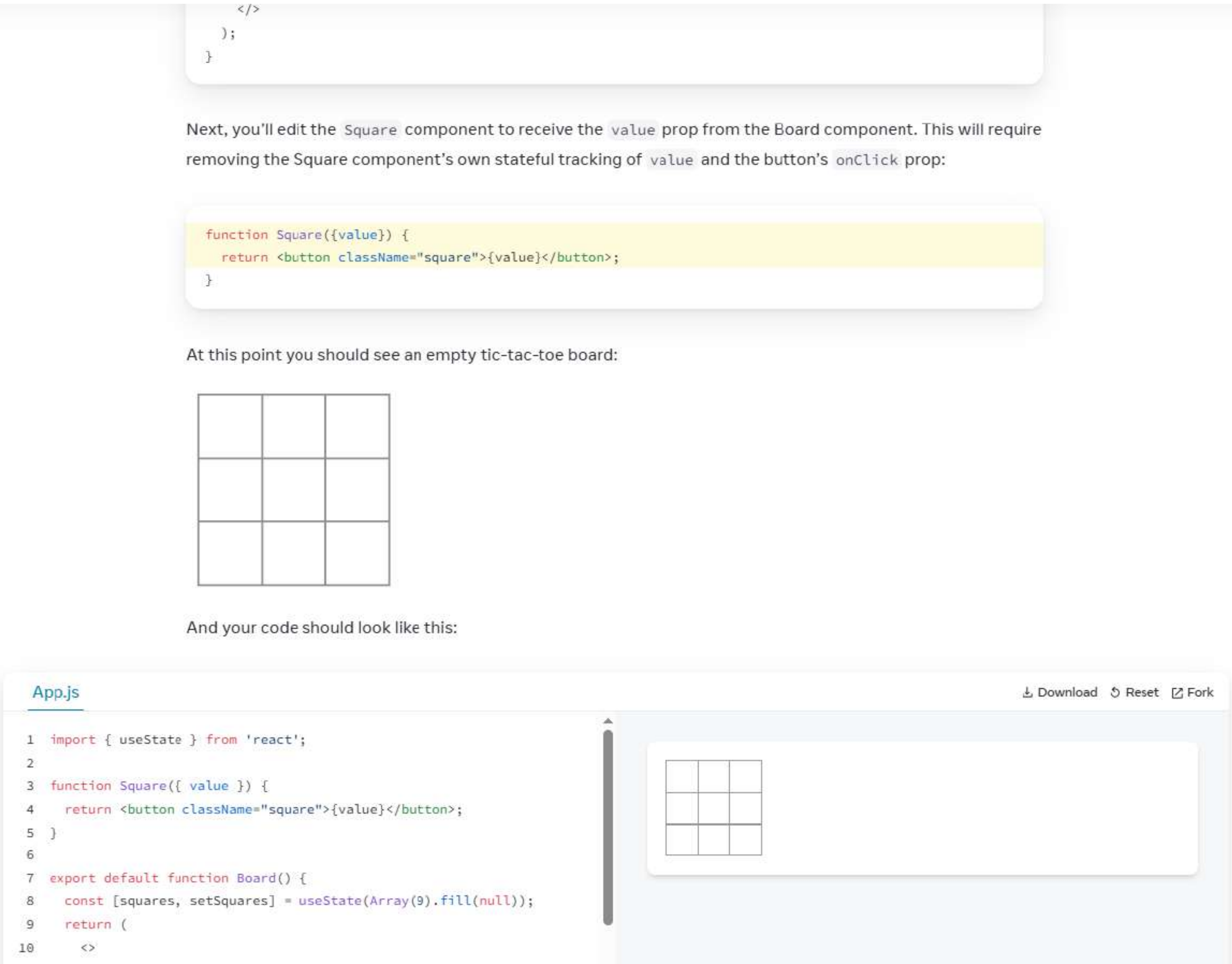
```
// ...
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    // ...
  );
}
```

`Array(9).fill(null)` creates an array with nine elements and sets each of them to `null`. The `useState()` call around it declares a `squares` state variable that's initially set to that array. Each entry in the array corresponds to the value of a square. When you fill the board in later, the `squares` array will look like this:

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

Now your `Board` component needs to pass the `value` prop down to each `Square` that it renders:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    <>
      <div className="board-row">
        <Square value={squares[0]} />
        <Square value={squares[1]} />
        <Square value={squares[2]} />
      </div>
      <div className="board-row">
        <Square value={squares[3]} />
        <Square value={squares[4]} />
        <Square value={squares[5]} />
      </div>
      <div className="board-row">
        <Square value={squares[6]} />
        <Square value={squares[7]} />
        <Square value={squares[8]} />
      </div>
    </>
```



```
    </>
  );
}
```

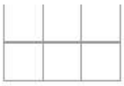
Next, you'll edit the `Square` component to receive the `value` prop from the Board component. This will require removing the Square component's own stateful tracking of `value` and the button's `onClick` prop:

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

At this point you should see an empty tic-tac-toe board:

And your code should look like this:

```
import { useState } from 'react';

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

export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    <>
    <>
      <div className="board-row">
        <Square value={squares[0]} />
        <Square value={squares[1]} />
        <Square value={squares[2]} />
      </div>
      <div className="board-row">
        <Square value={squares[3]} />
        <Square value={squares[4]} />
        <Square value={squares[5]} />
      </div>
      <div className="board-row">
        <Square value={squares[6]} />
        <Square value={squares[7]} />
        <Square value={squares[8]} />
      </div>
    </>
  );
}

```



Show less

Each Square will now receive a `value` prop that will either be `'X'`, `'O'`, or `null` for empty squares.

Next, you need to change what happens when a `Square` is clicked. The `Board` component now maintains which squares are filled. You'll need to create a way for the `Square` to update the `Board`'s state. Since state is private to a component that defines it, you cannot update the `Board`'s state directly from `Square`.

Instead, you'll pass down a function from the `Board` component to the `Square` component, and you'll have `Square` call that function when a square is clicked. You'll start with the function that the `Square` component will call when it is clicked. You'll call that function `onSquareClick`:

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

Next, you'll add the `onSquareClick` function to the `Square` component's props:

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

Now you'll connect the `onSquareClick` prop to a function in the `Board` component that you'll name `handleClick`. To connect `onSquareClick` to `handleClick` you'll pass a function to the `onSquareClick` prop of the first `Square` component:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  return (
    <>
      <div className="board-row">
        <Square value={squares[0]} onSquareClick={handleClick} />
        //...
  );
}
```

Lastly, you will define the `handleClick` function inside the Board component to update the `squares` array holding your board's state:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick() {
    const nextSquares = squares.slice();
    nextSquares[0] = "X";
    setSquares(nextSquares);
  }

  return (
    // ...
  )
}
```

The `handleClick` function creates a copy of the `squares` array (`nextSquares`) with the JavaScript `slice()` Array method. Then, `handleClick` updates the `nextSquares` array to add `X` to the first (`[0]` index) square.

Calling the `setSquares` function lets React know the state of the component has changed. This will trigger a re-render of the components that use the `squares` state (`Board`) as well as its child components (the `Square` components that make up the board).

Note

JavaScript supports closures which means an inner function (e.g. `handleClick`) has access to variables and functions defined in a outer function (e.g. `Board`). The `handleClick` function can read the `squares` state and call the `setSquares` method because they are both defined inside of the `Board` function.

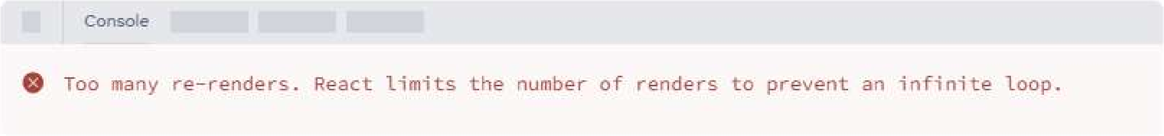
Now you can add X's to the board… but only to the upper left square. Your `handleClick` function is hardcoded to update the index for the upper left square (`0`). Let's update `handleClick` to be able to update any square. Add an argument `i` to the `handleClick` function that takes the index of the square to update:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    const nextSquares = squares.slice();
    nextSquares[i] = "X";
    setSquares(nextSquares);
  }

  return (
    // ...
  )
}
```

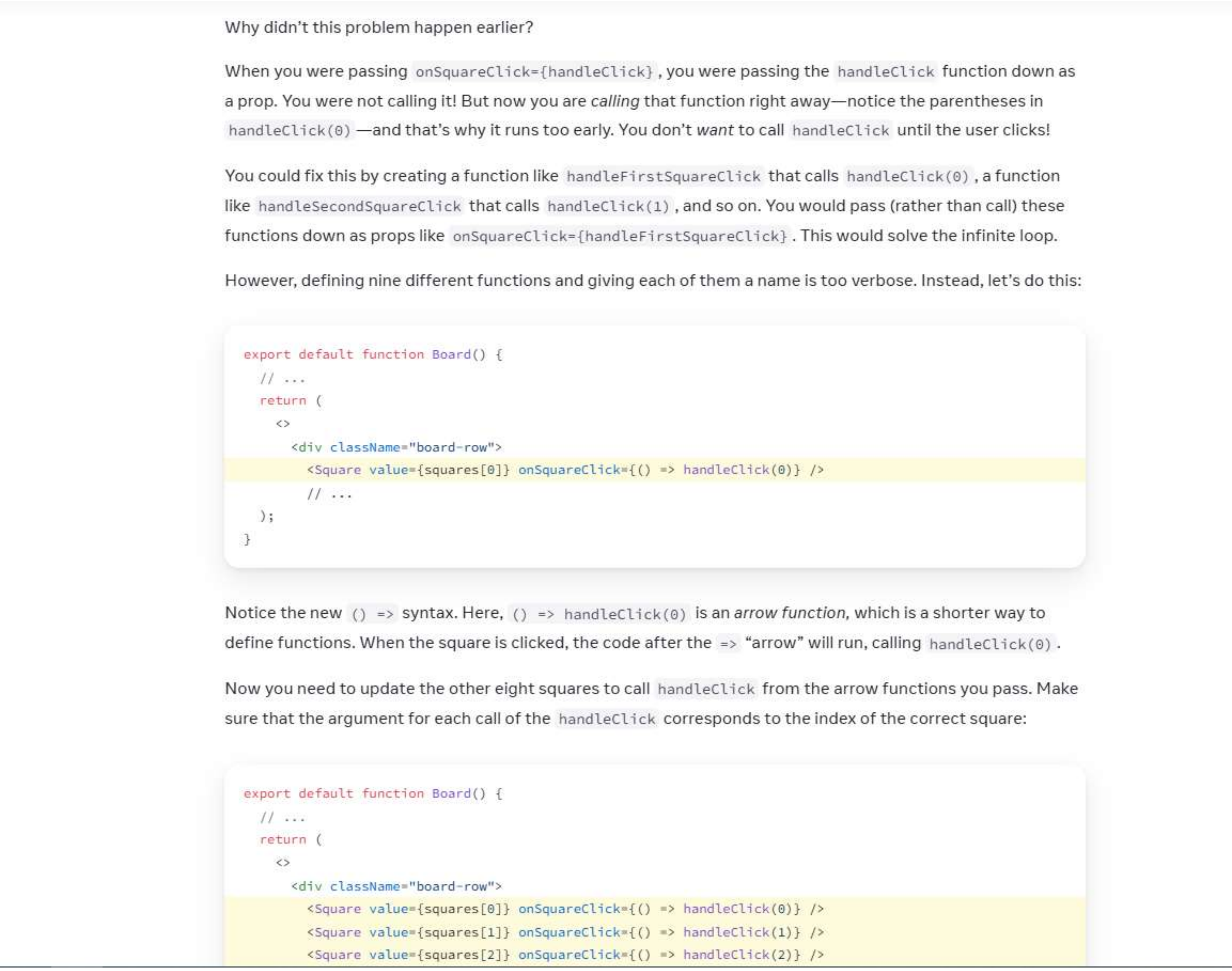
Next, you will need to pass that `i` to `handleClick`. You could try to set the `onSquareClick` prop of square to be `handleClick(0)` directly in the JSX like this, but it won't work:

```
<Square value={squares[0]} onSquareClick={handleClick(0)} />
```

Here is why this doesn't work. The `handleClick(0)` call will be a part of rendering the board component. Because `handleClick(0)` alters the state of the board component by calling `setSquares`, your entire board component will be re-rendered again. But this runs `handleClick(0)` again, leading to an infinite loop:



```
Too many re-renders. React limits the number of renders to prevent an infinite loop.
```



Why didn't this problem happen earlier?

When you were passing `onSquareClick={handleClick}`, you were passing the `handleClick` function down as a prop. You were not calling it! But now you are *calling* that function right away—notice the parentheses in `handleClick(0)`—and that's why it runs too early. You don't *want* to call `handleClick` until the user clicks!

You could fix this by creating a function like `handleFirstSquareClick` that calls `handleClick(0)`, a function like `handleSecondSquareClick` that calls `handleClick(1)`, and so on. You would pass (rather than call) these functions down as props like `onSquareClick={handleFirstSquareClick}`. This would solve the infinite loop.

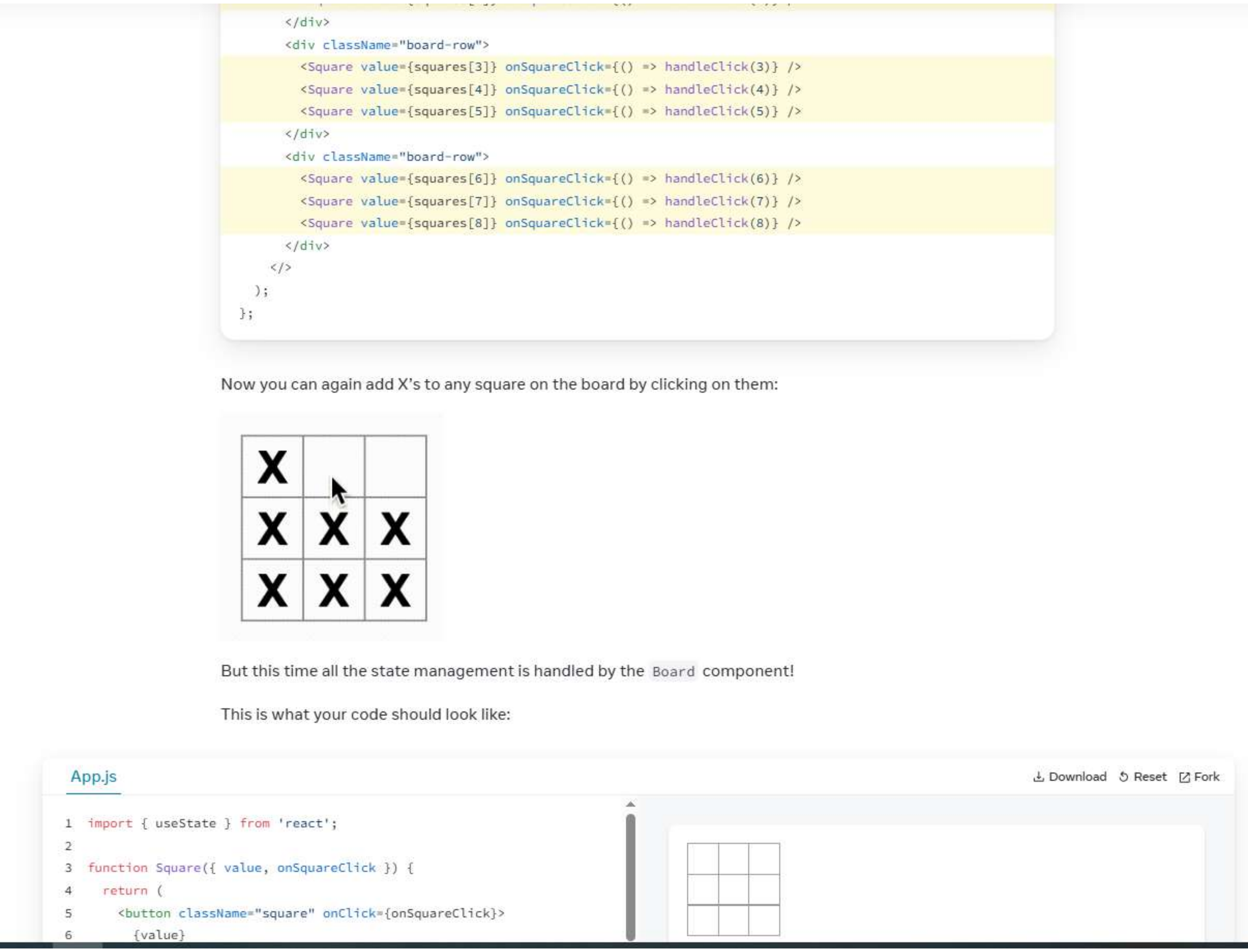
However, defining nine different functions and giving each of them a name is too verbose. Instead, let's do this:

```
export default function Board() {
  // ...
  return (
    <>
      <div className="board-row">
        <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
        // ...
  );
}
```

Notice the new `() =>` syntax. Here, `() => handleClick(0)` is an *arrow function,* which is a shorter way to define functions. When the square is clicked, the code after the `=>` "arrow" will run, calling `handleClick(0)`.

Now you need to update the other eight squares to call `handleClick` from the arrow functions you pass. Make sure that the argument for each call of the `handleClick` corresponds to the index of the correct square:

```
export default function Board() {
  // ...
  return (
    <>
      <div className="board-row">
        <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
        <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
        <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
```



```
      </div>
      <div className="board-row">
        <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
        <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
        <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
      </div>
      <div className="board-row">
        <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
        <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
        <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
      </div>
    </>
  );
};
```

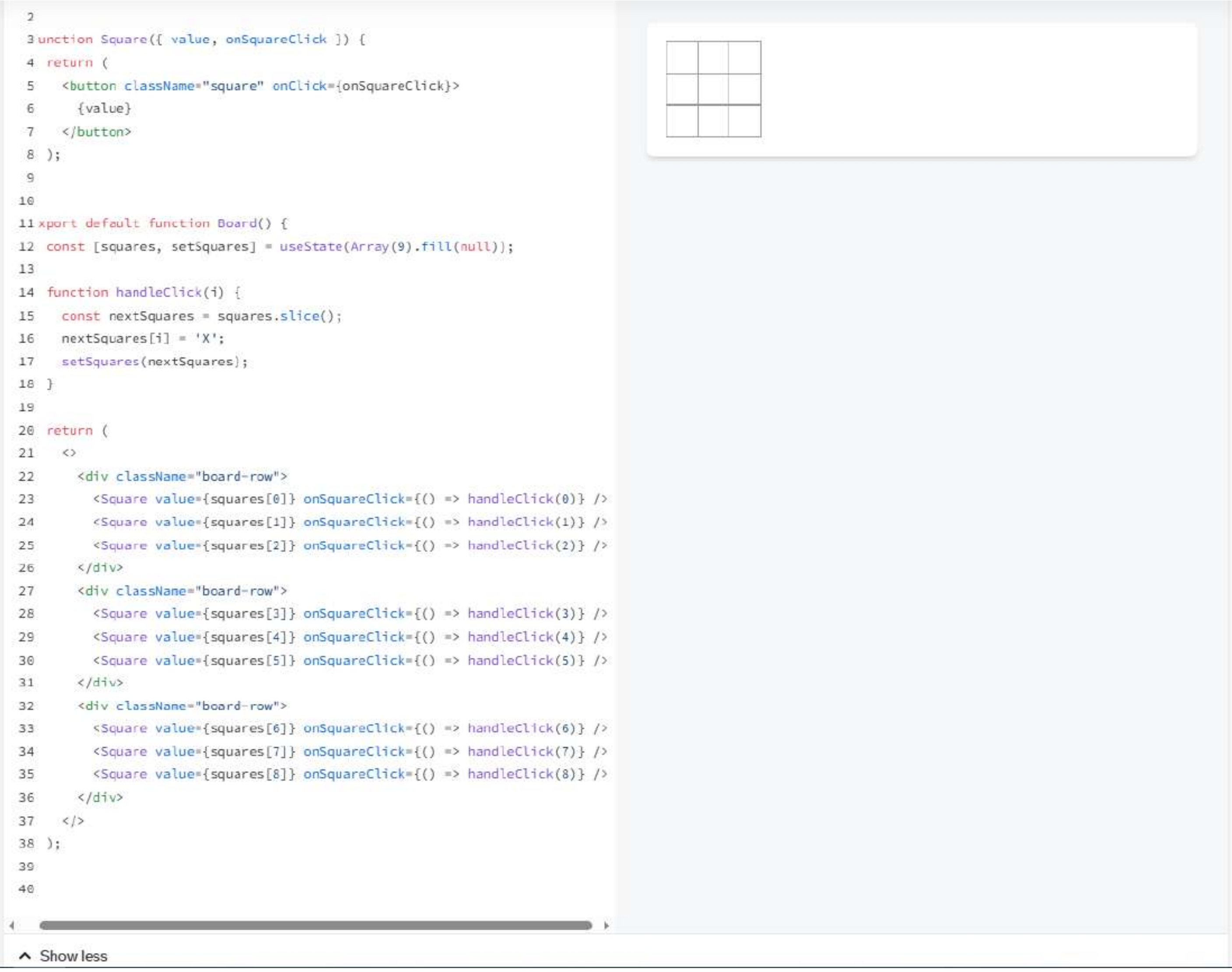
Now you can again add X's to any square on the board by clicking on them:

X		
X	X	X
X	X	X

But this time all the state management is handled by the `Board` component!

This is what your code should look like:

```
import { useState } from 'react';

function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
```



```

unction Square({ value, onSquareClick }) {
 return (
   <button className="square" onClick={onSquareClick}>
     {value}
   </button>
 );


xport default function Board() {
 const [squares, setSquares] = useState(Array(9).fill(null));

 function handleClick(i) {
   const nextSquares = squares.slice();
   nextSquares[i] = 'X';
   setSquares(nextSquares);
 }

 return (
   <>
     <div className="board-row">
       <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
       <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
       <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
     </div>
     <div className="board-row">
       <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
       <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
       <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
     </div>
     <div className="board-row">
       <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
       <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
       <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
     </div>
   </>
 );


```

Now that your state handling is in the `Board` component, the parent `Board` component passes props to the child `Square` components so that they can be displayed correctly. When clicking on a `Square`, the child `Square` component now asks the parent `Board` component to update the state of the board. When the `Board`'s state changes, both the `Board` component and every child `Square` re-renders automatically. Keeping the state of all squares in the `Board` component will allow it to determine the winner in the future.

Let's recap what happens when a user clicks the top left square on your board to add an `X` to it:

1. Clicking on the upper left square runs the function that the `button` received as its `onClick` prop from the `Square`. The `Square` component received that function as its `onSquareClick` prop from the `Board`. The `Board` component defined that function directly in the JSX. It calls `handleClick` with an argument of `0`.
2. `handleClick` uses the argument (`0`) to update the first element of the `squares` array from `null` to `X`.
3. The `squares` state of the `Board` component was updated, so the `Board` and all of its children re-render. This causes the `value` prop of the `Square` component with index `0` to change from `null` to `X`.

In the end the user sees that the upper left square has changed from empty to having a `X` after clicking it.

Note

The DOM `<button>` element's `onClick` attribute has a special meaning to React because it is a built-in component. For custom components like Square, the naming is up to you. You could give any name to the `Square`'s `onSquareClick` prop or `Board`'s `handleClick` function, and the code would work the same. In React, it's conventional to use `onSomething` names for props which represent events and `handleSomething` for the function definitions which handle those events.

Why immutability is important

Note how in `handleClick`, you call `.slice()` to create a copy of the `squares` array instead of modifying the existing array. To explain why, we need to discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to *mutate* the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes. Here is what it would look like if you mutated the `squares` array:

```
const squares = [null, null, null, null, null, null, null, null, null];
squares[0] = 'X';
// Now `squares` is ["X", null, null, null, null, null, null, null, null];
```

And here is what it would look like if you changed data without mutating the `squares` array:

```
const squares = [null, null, null, null, null, null, null, null, null];
const nextSquares = ['X', null, null, null, null, null, null, null, null];
// Now `squares` is unchanged, but `nextSquares` first element is 'X' rather than `null`
```

The result is the same but by not mutating (changing the underlying data) directly, you gain several benefits.

Immutability makes complex features much easier to implement. Later in this tutorial, you will implement a "time travel" feature that lets you review the game's history and "jump back" to past moves. This functionality isn't specific to games—an ability to undo and redo certain actions is a common requirement for apps. Avoiding direct data mutation lets you keep previous versions of the data intact, and reuse them later.

There is also another benefit of immutability. By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren't affected by the change. Although re-rendering is not by itself noticeable to the user (you shouldn't actively try to avoid it!), you might want to skip re-rendering a part of the tree that clearly wasn't affected by it for performance reasons. Immutability makes it very cheap for components to compare whether their data has changed or not. You can learn more about how React chooses when to re-render a component in the `memo` API reference.

Taking turns

It's now time to fix a major defect in this tic-tac-toe game: the "O"s cannot be marked on the board.

You'll set the first move to be "X" by default. Let's keep track of this by adding another piece of state to the Board component:

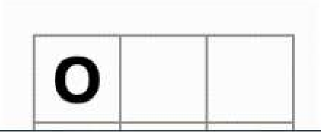
```
function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  // ...
}
```

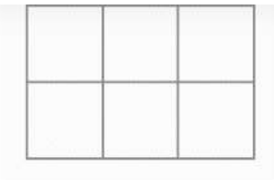
Each time a player moves, `xIsNext` (a boolean) will be flipped to determine which player goes next and the game's state will be saved. You'll update the `Board`'s `handleClick` function to flip the value of `xIsNext`:

```
export default function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = "X";
    } else {
      nextSquares[i] = "O";
    }
    setSquares(nextSquares);
    setXIsNext(!xIsNext);
  }

  return (
    //...
  );
}
```

Now, as you click on different squares, they will alternate between `X` and `O`, as they should!

But wait, there's a problem. Try clicking on the same square multiple times:



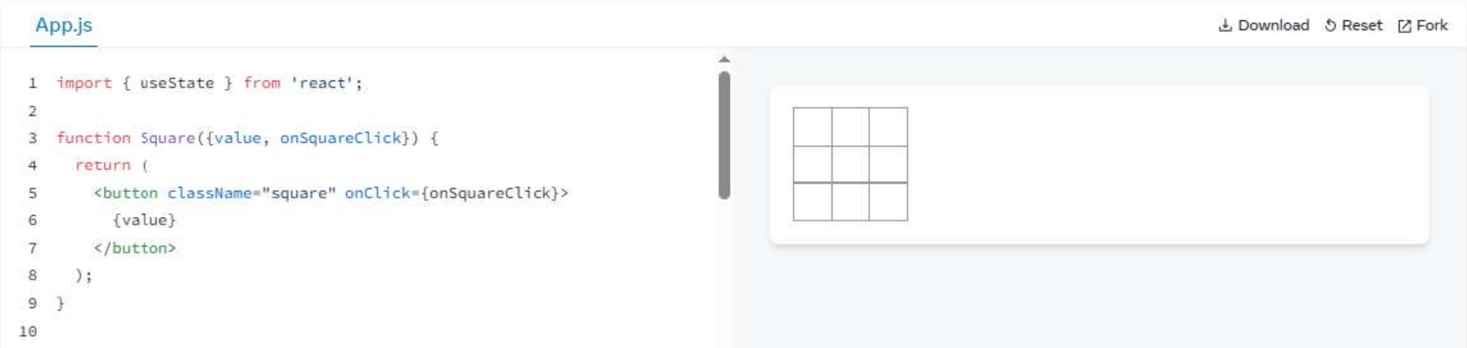


The `X` is overwritten by an `O`! While this would add a very interesting twist to the game, we're going to stick to the original rules for now.

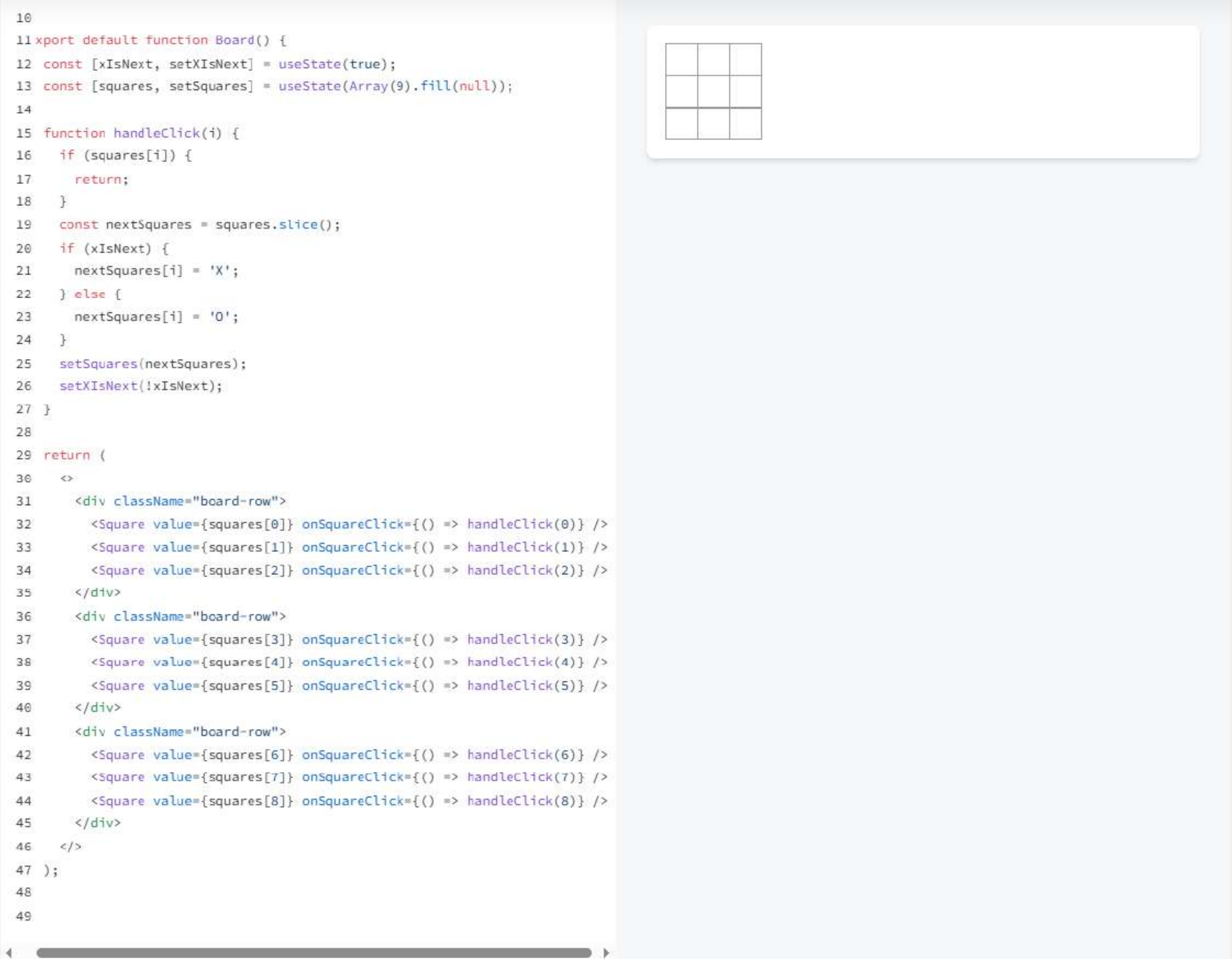
When you mark a square with a `X` or an `O` you aren't first checking to see if the square already has a `X` or `O` value. You can fix this by *returning early*. You'll check to see if the square already has a `X` or an `O`. If the square is already filled, you will `return` in the `handleClick` function early—before it tries to update the board state.

```
function handleClick(i) {
  if (squares[i]) {
    return;
  }
  const nextSquares = squares.slice();
  //...
}
```

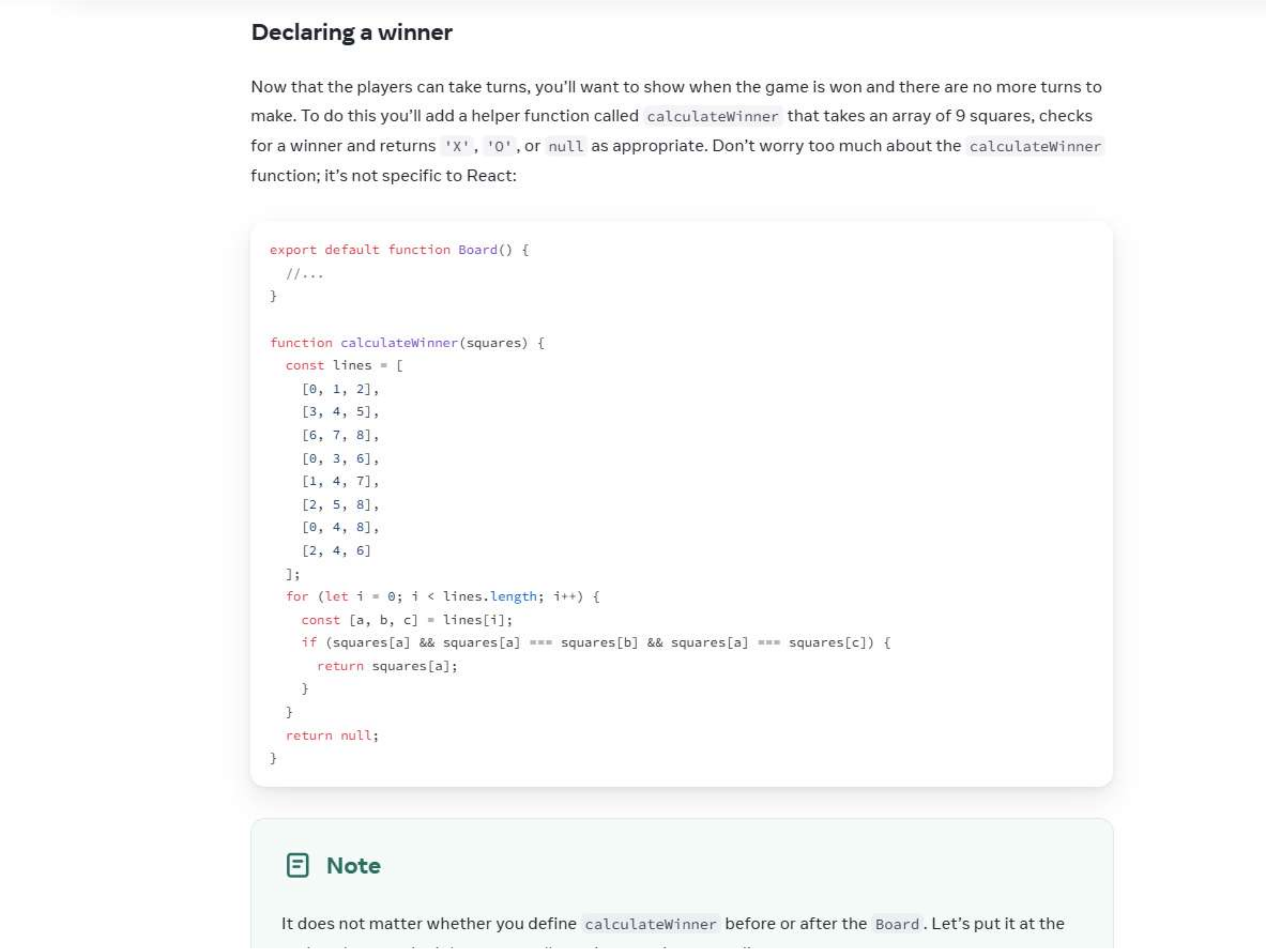
Now you can only add `X`'s or `O`'s to empty squares! Here is what your code should look like at this point:



```
import { useState } from 'react';

function Square({value, onSquareClick}) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}

```



```
xport default function Board() {
 const [xIsNext, setXIsNext] = useState(true);
 const [squares, setSquares] = useState(Array(9).fill(null));

 function handleClick(i) {
   if (squares[i]) {
     return;
   }
   const nextSquares = squares.slice();
   if (xIsNext) {
     nextSquares[i] = 'X';
   } else {
     nextSquares[i] = 'O';
   }
   setSquares(nextSquares);
   setXIsNext(!xIsNext);
 }

 return (
   <>
     <div className="board-row">
       <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
       <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
       <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
     </div>
     <div className="board-row">
       <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
       <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
       <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
     </div>
     <div className="board-row">
       <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
       <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
       <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
     </div>
   </>
 );
```



Declaring a winner

Now that the players can take turns, you'll want to show when the game is won and there are no more turns to make. To do this you'll add a helper function called `calculateWinner` that takes an array of 9 squares, checks for a winner and returns `'X'`, `'O'`, or `null` as appropriate. Don't worry too much about the `calculateWinner` function; it's not specific to React:

```
export default function Board() {
  //...
}

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;
}
```

Note

It does not matter whether you define `calculateWinner` before or after the `Board`. Let's put it at the