Note

It does not matter whether you define `calculateWinner` before or after the `Board`. Let's put it at the end so that you don't have to scroll past it every time you edit your components.

You will call `calculateWinner(squares)` in the `Board` component's `handleClick` function to check if a player has won. You can perform this check at the same time you check if a user has clicked a square that already has a `X` or and `O`. We'd like to return early in both cases:

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

To let the players know when the game is over, you can display text such as "Winner: X" or "Winner: O". To do that you'll add a `status` section to the `Board` component. The status will display the winner if the game is over and if the game is ongoing you'll display which player's turn is next:

```
export default function Board() {
  // ...
  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = "Winner: " + winner;
  } else {
    status = "Next player: " + (xIsNext ? "X" : "O");
  }

  return (
    <>
```

```
      <div className="status">{status}</div>
      <div className="board-row">
        // ...
  )
}
```

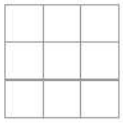
Congratulations! You now have a working tic-tac-toe game. And you've just learned the basics of React too. So *you* are the real winner here. Here is what the code should look like:

App.js

Download Reset Fork

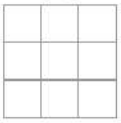
```
import { useState } from 'react';

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

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

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

```

Next player: X



```
28
29 const winner = calculateWinner(squares);
30 let status;
31 if (winner) {
32   status = 'Winner: ' + winner;
33 } else {
34   status = 'Next player: ' + (xIsNext ? 'X' : 'O');
35 }
36
37 return (
38   <>
39     <div className="status">{status}</div>
40     <div className="board-row">
41       <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
42       <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
43       <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
44     </div>
45     <div className="board-row">
46       <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
47       <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
48       <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
49     </div>
50     <div className="board-row">
51       <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
52       <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
53       <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
54     </div>
55   </>
56 );
57
58
59 unction calculateWinner(squares) {
60 const lines = [
61   [0, 1, 2],
62   [3, 4, 5],
63   [6, 7, 8],
64   [0, 3, 6],
65   [1, 4, 7],
66   [2, 5, 8],
67   [0, 4, 8],
68   [2, 4, 6],
69 ];
```

Next player: X



```
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;
```

Next player: X

Show less

Adding time travel

As a final exercise, let's make it possible to "go back in time" to the previous moves in the game.

Storing a history of moves

If you mutated the `squares` array, implementing time travel would be very difficult.

However, you used `slice()` to create a new copy of the `squares` array after every move, and treated it as immutable. This will allow you to store every past version of the `squares` array, and navigate between the turns that have already happened.

You'll store the past `squares` arrays in another array called `history`, which you'll store as a new state variable. The `history` array represents all board states, from the first to the last move, and has a shape like this:

```
[
  // Before first move
  [null, null, null, null, null, null, null, null, null],
  // After first move
  [null, null, null, null, 'X', null, null, null, null],
  // After second move
  [null, null, null, null, 'X', null, null, null, 'O'],
  // ...
]
```

Lifting state up, again

You will now write a new top-level component called `Game` to display a list of past moves. That's where you will place the `history` state that contains the entire game history.

Placing the `history` state into the `Game` component will let you remove the `squares` state from its child `Board` component. Just like you "lifted state up" from the `Square` component into the `Board` component, you will now lift it up from the `Board` into the top-level `Game` component. This gives the `Game` component full control over the `Board`'s data and lets it instruct the `Board` to render previous turns from the `history`.

First, add a `Game` component with `export default`. Have it render the `Board` component and some markup:

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

export default function Game() {
  return (
    <div className="game">
      <div className="game-board">
        <Board />
      </div>
      <div className="game-info">
        <ol>{/*TODO*/}</ol>
      </div>
    </div>
  );
}
```

Note that you are removing the `export default` keywords before the `function Board() {` declaration and adding them before the `function Game() {` declaration. This tells your `index.js` file to use the `Game` component as the top-level component instead of your `Board` component. The additional `div`s returned by the `Game` component are making room for the game information you'll add to the board later.

Add some state to the `Game` component to track which player is next and the history of moves:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  // ...
```

Notice how `[Array(9).fill(null)]` is an array with a single item, which itself is an array of 9 `null`s.

To render the squares for the current move, you'll want to read the last squares array from the `history`. You don't need `useState` for this—you already have enough information to calculate it during rendering:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];
  // ...
```

Next, create a `handlePlay` function inside the `Game` component that will be called by the `Board` component to update the game. Pass `xIsNext`, `currentSquares` and `handlePlay` as props to the `Board` component:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {
    // TODO
  }

  return (
    <div className="game">
      <div className="game-board">
        <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
        //...
  )
}
```

Let's make the `Board` component fully controlled by the props it receives. Change the `Board` component to take three props: `xIsNext`, `squares`, and a new `onPlay` function that `Board` can call with the updated squares array when a player makes a move. Next, remove the first two lines of the `Board` function that call `useState`:

```
function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    //...
  }
  // ...
}
```

Now replace the `setSquares` and `setXIsNext` calls in `handleClick` in the `Board` component with a single call to your new `onPlay` function so the `Game` component can update the `Board` when the user clicks a square:

```
function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = "X";
    } else {
      nextSquares[i] = "O";
    }
    onPlay(nextSquares);
  }
  //...
}
```

The `Board` component is fully controlled by the props passed to it by the `Game` component. You need to implement the `handlePlay` function in the `Game` component to get the game working again.

What should `handlePlay` do when called? Remember that Board used to call `setSquares` with an updated array; now it passes the updated `squares` array to `onPlay`.

The `handlePlay` function needs to update `Game`'s state to trigger a re-render, but you don't have a `setSquares` function that you can call any more—you're now using the `history` state variable to store this information. You'll want to update `history` by appending the updated `squares` array as a new history entry. You also want to toggle `xIsNext`, just as Board used to do:

```
export default function Game() {
  //...
  function handlePlay(nextSquares) {
    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }
  //...
}
```

Here, `[...history, nextSquares]` creates a new array that contains all the items in `history`, followed by `nextSquares`. (You can read the `...history` *spread syntax* as "enumerate all the items in `history`".)

For example, if `history` is `[[null,null,null], ["X",null,null]]` and `nextSquares` is `["X",null,"O"]`, then the new `[...history, nextSquares]` array will be `[[null,null,null], ["X",null,null], ["X",null,"O"]]`.

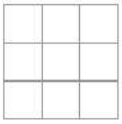
At this point, you've moved the state to live in the `Game` component, and the UI should be fully working, just as it was before the refactor. Here is what the code should look like at this point:

App.js

Download Reset Fork

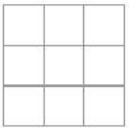
```
import { useState } from 'react';

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

function Board({ xIsNext, squares, onPlay }) {
```

Next player: X



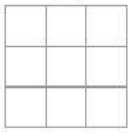
```
11 nction Board({ xIsNext, squares, onPlay }) {
12 function handleClick(i) {
13   if (calculateWinner(squares) || squares[i]) {
14     return;
15   }
16   const nextSquares = squares.slice();
17   if (xIsNext) {
18     nextSquares[i] = 'X';
19   } else {
20     nextSquares[i] = 'O';
21   }
22   onPlay(nextSquares);
23 }
24
25 const winner = calculateWinner(squares);
26 let status;
27 if (winner) {
28   status = 'Winner: ' + winner;
29 } else {
30   status = 'Next player: ' + (xIsNext ? 'X' : 'O');
31 }
32
33 return (
34   <>
35     <div className="status">{status}</div>
36     <div className="board-row">
37       <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
38       <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
39       <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
40     </div>
41     <div className="board-row">
42       <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
43       <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
44       <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
45     </div>
46     <div className="board-row">
47       <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
48       <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
49       <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
50     </div>
51   </>
```

Next player: X

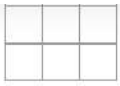


```
52  );
53 }
54
55 export default function Game() {
56   const [xIsNext, setXIsNext] = useState(true);
57   const [history, setHistory] = useState([Array(9).fill(null)]);
58   const currentSquares = history[history.length - 1];
59
60   function handlePlay(nextSquares) {
61     setHistory([...history, nextSquares]);
62     setXIsNext(!xIsNext);
63   }
64
65   return (
66     <div className="game">
67       <div className="game-board">
68         <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePl
69       </div>
70       <div className="game-info">
71         <ol>{/*TODO*/}</ol>
72       </div>
73     </div>
74   );
75 }
76
77 function calculateWinner(squares) {
78   const lines = [
79     [0, 1, 2],
80     [3, 4, 5],
81     [6, 7, 8],
82     [0, 3, 6],
83     [1, 4, 7],
84     [2, 5, 8],
85     [0, 4, 8],
86     [2, 4, 6],
87   ];
88   for (let i = 0; i < lines.length; i++) {
89     const [a, b, c] = lines[i];
90     if (squares[a] && squares[a] === squares[b] && squares[a] === squares[
91       return squares[a];
92     }
93   }
```

Next player: X



```
93  }
94  return null;
95 }
96
```



Show less

Showing the past moves

Since you are recording the tic-tac-toe game's history, you can now display a list of past moves to the player.

React elements like `<button>` are regular JavaScript objects; you can pass them around in your application. To render multiple items in React, you can use an array of React elements.

You already have an array of `history` moves in state, so now you need to transform it to an array of React elements. In JavaScript, to transform one array into another, you can use the array `map` method:

```
[1, 2, 3].map((x) => x * 2) // [2, 4, 6]
```

You'll use `map` to transform your `history` of moves into React elements representing buttons on the screen, and display a list of buttons to "jump" to past moves. Let's `map` over the `history` in the Game component:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {
    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }

  function jumpTo(nextMove) {
    // TODO
  }
```

```
  function jumpTo(nextMove) {
    // TODO
  }

  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = 'Go to move #' + move;
    } else {
      description = 'Go to game start';
    }
    return (
      <li>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });

  return (
    <div className="game">
      <div className="game-board">
        <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
      </div>
      <div className="game-info">
        <ol>{moves}</ol>
      </div>
    </div>
  );
}
```

You can see what your code should look like below. Note that you should see an error in the developer tools console that says:

```
Console

Warning: Each child in an array or iterator should have a unique "key" prop. Check the
render method of `Game`.
```

You'll fix this error in the next section.

App.js

Download Reset Fork

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

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

function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }

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

```

Next player: X

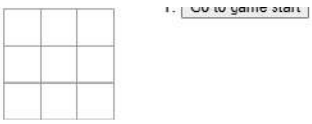
1. Go to game start

```
32
33    return (
34      <>
35        <div className="status">{status}</div>
36        <div className="board-row">
37          <Square value={squares[0]} onSquareClick={() => handleClick(0)}
38          <Square value={squares[1]} onSquareClick={() => handleClick(1)}
39          <Square value={squares[2]} onSquareClick={() => handleClick(2)}
40        </div>
41        <div className="board-row">
42          <Square value={squares[3]} onSquareClick={() => handleClick(3)}
43          <Square value={squares[4]} onSquareClick={() => handleClick(4)}
44          <Square value={squares[5]} onSquareClick={() => handleClick(5)}
45        </div>
46        <div className="board-row">
47          <Square value={squares[6]} onSquareClick={() => handleClick(6)}
48          <Square value={squares[7]} onSquareClick={() => handleClick(7)}
49          <Square value={squares[8]} onSquareClick={() => handleClick(8)}
50        </div>
51      </>
52    );
53  }
54
55  export default function Game() {
56    const [xIsNext, setXIsNext] = useState(true);
57    const [history, setHistory] = useState([Array(9).fill(null)]);
58    const currentSquares = history[history.length - 1];
59
60    function handlePlay(nextSquares) {
61      setHistory([...history, nextSquares]);
62      setXIsNext(!xIsNext);
63    }
64
65    function jumpTo(nextMove) {
66      // TODO
67    }
68
69    const moves = history.map((squares, move) => {
70      let description;
71      if (move > 0) {
```

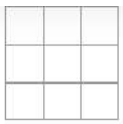
Next player: X

1. Go to game start

```
if (move > 0) {
  description = 'Go to move #' + move;
} else {
  description = 'Go to game start';
}
return (
  <li>
    <button onClick={() => jumpTo(move)}>{description}</button>
  </li>
);
);

eturn (
<div className="game">
  <div className="game-board">
    <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay
  </div>
  <div className="game-info">
    <ol>{moves}</ol>
  </div>
</div>
;


ction calculateWinner(squares) {
onst 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],
;
or (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];
```



```
}

eturn null;
```



Console (1)

Show less

As you iterate through `history` array inside the function you passed to `map`, the `squares` argument goes through each element of `history`, and the `move` argument goes through each array index: `0`, `1`, `2`, ... (In most cases, you'd need the actual array elements, but to render a list of moves you will only need indexes.)

For each move in the tic-tac-toe game's history, you create a list item `<li>` which contains a button `<button>`. The button has an `onClick` handler which calls a function called `jumpTo` (that you haven't implemented yet).

For now, you should see a list of the moves that occurred in the game and an error in the developer tools console. Let's discuss what the "key" error means.

Picking a key

When you render a list, React stores some information about each rendered list item. When you update a list, React needs to determine what has changed. You could have added, removed, re-arranged, or updated the list's items.

Imagine transitioning from

```
<li>Alexa: 7 tasks left</li>
<li>Ben: 5 tasks left</li>
```

to

```
<li>Ben: 9 tasks left</li>
<li>Claudia: 8 tasks left</li>
<li>Alexa: 5 tasks left</li>
```

In addition to the updated counts, a human reading this would probably say that you swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and does not know what you intended, so you need to specify a *key* property for each list item to differentiate each list item from its siblings. If your data was from a database, Alexa, Ben, and Claudia's database IDs could be used as keys.

```
<li key={user.id}>
  {user.name}: {user.taskCount} tasks left
</li>
```

When a list is re-rendered, React takes each list item's key and searches the previous list's items for a matching key. If the current list has a key that didn't exist before, React creates a component. If the current list is missing a key that existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved.

Keys tell React about the identity of each component, which allows React to maintain state between re-renders. If a component's key changes, the component will be destroyed and re-created with a new state.

`key` is a special and reserved property in React. When an element is created, React extracts the `key` property and stores the key directly on the returned element. Even though `key` may look like it is passed as props, React automatically uses `key` to decide which components to update. There's no way for a component to ask what `key` its parent specified.

It's strongly recommended that you assign proper keys whenever you build dynamic lists. If you don't have an appropriate key, you may want to consider restructuring your data so that you do.

If no key is specified, React will report an error and use the array index as a key by default. Using the array index as a key is problematic when trying to re-order a list's items or inserting/removing list items. Explicitly passing `key={i}` silences the error but has the same problems as array indices and is not recommended in most cases.

Keys do not need to be globally unique; they only need to be unique between components and their siblings.

Implementing time travel

Implementing time travel

In the tic-tac-toe game's history, each past move has a unique ID associated with it: it's the sequential number of the move. Moves will never be re-ordered, deleted, or inserted in the middle, so it's safe to use the move index as a key.

In the `Game` function, you can add the key as `<li key={move}>`, and if you reload the rendered game, React's "key" error should disappear:

```
const moves = history.map((squares, move) => {
  //...
  return (
    <li key={move}>
      <button onClick={() => jumpTo(move)}>{description}</button>
    </li>
  );
});
```

App.js

Download Reset Fork

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

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

function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
```

Next player: X

1. Go to game start

```
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }

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

  return (
    <>
      <div className="status">{status}</div>
      <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>
    </>
  );
}
```

Next player: X

1. Go to game start

```
55 export default function Game() {
56   const [xIsNext, setXIsNext] = useState(true);
57   const [history, setHistory] = useState([Array(9).fill(null)]);
58   const currentSquares = history[history.length - 1];
59
60   function handlePlay(nextSquares) {
61     setHistory([...history, nextSquares]);
62     setXIsNext(!xIsNext);
63   }
64
65   function jumpTo(nextMove) {
66     // TODO
67   }
68
69   const moves = history.map((squares, move) => {
70     let description;
71     if (move > 0) {
72       description = 'Go to move #' + move;
73     } else {
74       description = 'Go to game start';
75     }
76     return (
77       <li key={move}>
78         <button onClick={() => jumpTo(move)}>{description}</button>
79       </li>
80     );
81   });
82
83   return (
84     <div className="game">
85       <div className="game-board">
86         <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handle
87       </div>
88       <div className="game-info">
89         <ol>{moves}</ol>
90       </div>
91     </div>
92   );
93 }
94
```

Next player: X

1. Go to game start