```
94
95  function calculateWinner(squares) {
96    const lines = [
97      [0, 1, 2],
98      [3, 4, 5],
99      [6, 7, 8],
100     [0, 3, 6],
101     [1, 4, 7],
102     [2, 5, 8],
103     [0, 4, 8],
104     [2, 4, 6],
105   ];
106   for (let i = 0; i < lines.length; i++) {
107     const [a, b, c] = lines[i];
108     if (squares[a] && squares[a] === squares[b] && squares[a] === square
109       return squares[a];
110     }
111   }
112   return null;
113 }
114
115
```

Show less

Before you can implement `jumpTo`, you need the `Game` component to keep track of which step the user is currently viewing. To do this, define a new state variable called `currentMove`, defaulting to `0`:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const currentSquares = history[history.length - 1];
  //...
}
```

Next, update the `jumpTo` function inside `Game` to update that `currentMove`. You'll also set `xIsNext` to `true` if the number that you're changing `currentMove` to is even.

```
export default function Game() {
  // ...
  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
    setXIsNext(nextMove % 2 === 0);
  }
  //...
}
```

You will now make two changes to the `Game`'s `handlePlay` function which is called when you click on a square.

- If you "go back in time" and then make a new move from that point, you only want to keep the history up to that point. Instead of adding `nextSquares` after all items (`...` spread syntax) in `history`, you'll add it after all items in `history.slice(0, currentMove + 1)` so that you're only keeping that portion of the old history.
- Each time a move is made, you need to update `currentMove` to point to the latest history entry.

```
function handlePlay(nextSquares) {
  const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
  setHistory(nextHistory);
  setCurrentMove(nextHistory.length - 1);
  setXIsNext(!xIsNext);
}
```

Finally, you will modify the `Game` component to render the currently selected move, instead of always rendering the final move:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const currentSquares = history[currentMove];
```

```
  // ...
}
```

If you click on any step in the game's history, the tic-tac-toe board should immediately update to show what the board looked like after that step occurred.

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: O

X		

1. Go to game start
2. Go to move #1

```
  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>
    </>
  );
}

export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquar
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
    setXIsNext(!xIsNext);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
    setXIsNext(nextMove % 2 === 0);
  }

  const moves = history.map((squares, move) => {
    let description;
```

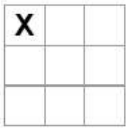
Next player: O

X		

1. Go to game start
2. Go to move #1

```
74 description;
75 move > 0) {
76 scription = 'Go to move #' + move;
77 se {
78 scription = 'Go to game start';
79
80 rn (
81 i key={move}>
82 <button onClick={() => jumpTo(move)}>{description}</button>
83 li>
84
85
86
87  (
88  className="game">
89 iv className="game-board">
90 <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
91 div>
92 iv className="game-info">
93 <ol>{moves}</ol>
94 div>
95 v>
96
97
98
99  calculateWinner(squares) {
100 lines = [
101 1, 2],
102 4, 5],
103 7, 8],
104 3, 6],
105 4, 7],
106 5, 8],
107 4, 8],
108 4, 6],
109
110 et i = 0; i < lines.length; i++) {
111 t [a, b, c] = lines[i];
112 squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
113 turn squares[a];
114
115
```

Next player: O



1. Go to game start
2. Go to move #1

```
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === square
      return squares[a];
    }
  }
  return null;
}

```

Show less

Final cleanup

If you look at the code very closely, you may notice that `xIsNext === true` when `currentMove` is even and `xIsNext === false` when `currentMove` is odd. In other words, if you know the value of `currentMove`, then you can always figure out what `xIsNext` should be.

There's no reason for you to store both of these in state. In fact, always try to avoid redundant state. Simplifying what you store in state reduces bugs and makes your code easier to understand. Change `Game` so that it doesn't store `xIsNext` as a separate state variable and instead figures it out based on the `currentMove`:

```
export default function Game() {
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xIsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }
  // ...
}
```

```
  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }
  // ...
}
```

You no longer need the `xIsNext` state declaration or the calls to `setXIsNext`. Now, there's no chance for `xIsNext` to get out of sync with `currentMove`, even if you make a mistake while coding the components.

Wrapping up

Congratulations! You've created a tic-tac-toe game that:

- Lets you play tic-tac-toe,
- Indicates when a player has won the game,
- Stores a game's history as a game progresses,
- Allows players to review a game's history and see previous versions of a game's board.

Nice work! We hope you now feel like you have a decent grasp of how React works.

Check out the final result here:

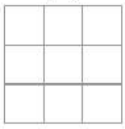
App.js

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;
```

```
      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');
  }

  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>
    </>
  );
}

```

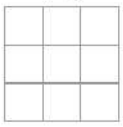
Next player: X



1. Go to game start

```
}

export default function Game() {
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xIsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquar
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }

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

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

Next player: X



1. Go to game start

```
}

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] === square
      return squares[a];
    }
  }
  return null;
}

```

Show less

If you have extra time or want to practice your new React skills, here are some ideas for improvements that you could make to the tic-tac-toe game, listed in order of increasing difficulty:

1. For the current move only, show "You are at move #..." instead of a button.
2. Rewrite `Board` to use two loops to make the squares instead of hardcoding them.
3. Add a toggle button that lets you sort the moves in either ascending or descending order.
4. When someone wins, highlight the three squares that caused the win (and when no one wins, display a message about the result being a draw).
5. Display the location for each move in the format (row, col) in the move history list.

Throughout this tutorial, you've touched on React concepts including elements, components, props, and state. Now that you've seen how these concepts work when building a game, check out Thinking in React to see how the same React concepts work when building an app's UI.