

Notes on React

Components:

A component is a piece of reusable code that represents a part of the user interface. Components are used to render, manage and update the UI elements in your application.

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

The First Line:

The function here calls Square().

Export:

Keyword in JavaScript which makes this function accessible outside of this file.

Default:

Keyword in JavaScript tells other files using your code that it's the main function in your file.

The Second Line:

Returns a button.

Return:

The return keyword in JavaScript means whatever comes after is returned as a value to the caller of the function.

<button> </button>

Is a JSX element. A JSX element is a combination of JavaScript code and HTML tags that describe what you'd like to display. </button> closes the JSX element to indicate that any following content shouldn't be placed inside the button.

className="square"

Is a button property or **prop that tells scc how to style the button.**

X:

Is the text displayed inside of the button

styles.css

This file defines the styles of your React app. The first two CSS ***** and **body** define the style of large parts of your app while the **.square** selector defines the style of any component where the className property is set to square.

In your code that would match the button form your Square component in the **App.js**.

index.js

This file is not edited in the tutorial but it is the bridge between the component you created in the App.js file and the web browser.

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App';
```

Lines 1-5 bring all the necessary pieces together:.

- React
- React's Library (to talk to the web browser "React DOM")
- The styles of your components
- The component your created in App.js

The remainder of the files brings all the pieces together and injects the final product into **index.html**

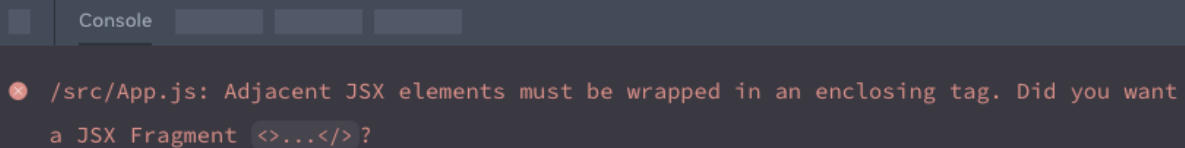
Building the Board:

The board only starts with a single Square but I will need nine.

However if you simply copy and past your Square you will get an error like this.

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

Here is the Error:

A screenshot of a web browser's developer console. The 'Console' tab is selected. A red error icon is on the left. The text of the error is: '/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX Fragment <>...</>?'.

```
/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want  
a JSX Fragment <>...</>?
```

Important:

React components need to return a single JSX element and not multiple adjacent JSX elements like two buttons.

To fix this you can use ~

Fragments

Such as `<>` and `</>` to wrap multiple adjacent JSX elements like this.

A screenshot of a code editor showing a function named 'Square'. The function returns a JSX fragment wrapped in `<>` and `</>` tags. Inside the fragment, there are nine `<button>` elements, each with `className="square"` and containing the text 'X'. Each button is followed by a comment indicating its index from 1 to 9.

```
1 export default function Square() {  
2   return (  
3     /* single lines needs ; while fragments do not */  
4     <>  
5       <button className="square">X</button> {/* 1 */}  
6       <button className="square">X</button> {/* 2 */}  
7       <button className="square">X</button> {/* 3 */}  
8       <button className="square">X</button> {/* 4 */}  
9       <button className="square">X</button> {/* 5 */}  
10      <button className="square">X</button> {/* 6 */}  
11      <button className="square">X</button> {/* 7 */}  
12      <button className="square">X</button> {/* 8 */}  
13      <button className="square">X</button> {/* 9 */}  
14    </>  
15  )  
16 }
```

However this only displays the boxes in a row next to each other.

To solve this we will need to group the squares into rows with

<div>

The div element in HTML when added to the DOM as a node. Can be used as both parents for grouping children components. However only div can appear on screen.

While using div and some CSS classes give each square a number to make sure you know where each square is displayed.

Next Problem

Now our Square component isn't really a square anymore.

Passing Data Through Props

Now we will want to change the values of a square from empty to "X" when the user clicks on the square.

With how you've built the board so far would need to copy-paste the code that updates the square nine times.

Once for each square.

Instead of copy and pasting Reacts components architecture allows you to create a reusable component to avoid messy duplicated code.

First we are going to copy the line defining your first square.

```
<button className="square"> 1 </button>
```

This is from the Board component. We will make this a new component outside of our board.

Like this:

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

```
Show usages
1 function Square() {
2   return <button className="square">1</button>;
3 }
4
```

Note unlike how the browser divs your own components board and square must start with capital letters

Now lets fix our numbers.

Props

Props are used to pass the value each square should have from the parent component (Board) to its child Square

Update Square component to read the value prop that you'll pass from the board like this.

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

Function Square ({ value }) indicates the square component can be passed a prop called value.

Now we want to display that value instead of 1 inside every square.

Ops

We did a syntax error and just displayed the text value.

Lets fix it.

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

Now we should see an empty board.

This is because the Board component hasn't passed the value prop to each Square component it renders yet. To fix this you will add the value prop to each Square component rendered by the board component.

Making an Interactive Component:

Let fill the Square component with an X when you click it. Declare a function called handleClick

If you click on a square now, you should see a log saying “clicked!”.

As a next step you want the Square component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things components use

state

React provides a special function called **useState** that you can call from your component to let it “remember things.

Let store the current value of the Square in state and change it when the Square is clicked.

You will need to import useState at the top of the file. Remove the value prop from the Square component instead add a new line at the start of the square that calls useState. Have it return a state variable called value:

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

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

  function handleClick() {
    //...
```

Value stores the value and the setValue is a function that can be used to change the value. The null passed to useState is used as the initial value for this state variable, so value here starts off equal to null.

Since the Square component no longer accept props anymore you will need to remove the value prop from all the nine squares.

Now we will change Square to display an “X” when clicked. Replaced the `console.log()` event handler with `setValue(“X”);`

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 will see the X on the game board.

Each square has its own state based on the Function Definition.

Completing the Game:

Given what we have now we have all the necessary building block for completing this project.

Lifting State Up:

Currently each square component maintains a part of the games 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 Squares components.

How to approach?

First we might guess that the Board needs to ask each square for the squares 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 games states 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.

Lets take the 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.

```
const [squares, setSquares] =  
useState(Array(9).fill(null));
```

Creates an array with nine elements and sets each of them to null.

The useState call around it declares squares state variable that 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.

Now we need to edit the Square component to receive the value prop form the Board component. This will required removing the Square components own stateful tracking of value and the buttons onClick prop

Now each Square will receive a value prop that will either be X O or null form empty squares.

To do this we need to change what happens when a square is clicked. The board component now maintains which square are filled you will need to create a way for the square to updates the Boards states.

Since state is private to a component that defines it you cannot update the boards state directly form Square.

Instead we will pass down a function from the board component to the Square component and we will have Square call that function when a square is clicked. We will start with the function that Square component will call when it is clicked. You'll call that function on SquareClick:

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

Next we will add onSquareClick function to the Square components 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} />
        //...
      </div>
    </>
  );
}
```

Lastly we will define the `handleClick` function inside the Board component to update the squares Array holding your boards 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 an 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.

Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

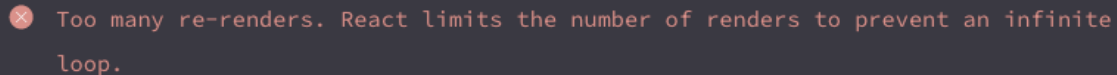
Now you can add X's to the board... but only to the upper left square. Your handleClick function is hardcoded to update the index for the upper left square (0). Let's update handleClick to be able to update any square. Add an argument i to the handleClick function that takes the index of the square to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

A dark gray rectangular box containing a red error message. The message starts with a red 'X' icon, followed by the text: "Too many re-renders. React limits the number of renders to prevent 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]} onClick={() => handleClick(0)} />  
        // ...  
      </div>  
    );  
  }  
}
```

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

With state handling working 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: 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.

`handleClick` uses the argument (0) to update the first element of the `squares` array from null to X.

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];
```

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:

Like so~

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

Declaring a Winner

Now that the player 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 can take an array of 9 squares checks for a winner and returns x or o or null as appropriate. Don't worry too much about the calculateWinner function it's not specific to React.

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 an O. We'd like to return early in both cases:

Here is what that would look like.

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

To let 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 section will display the winner if the game is over and if the game is ongoing you'll display which players 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:

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 divs 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 nulls. 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} />
        //...
      </div>
    </div>
  )
}
```

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:

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:

```
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);
  }

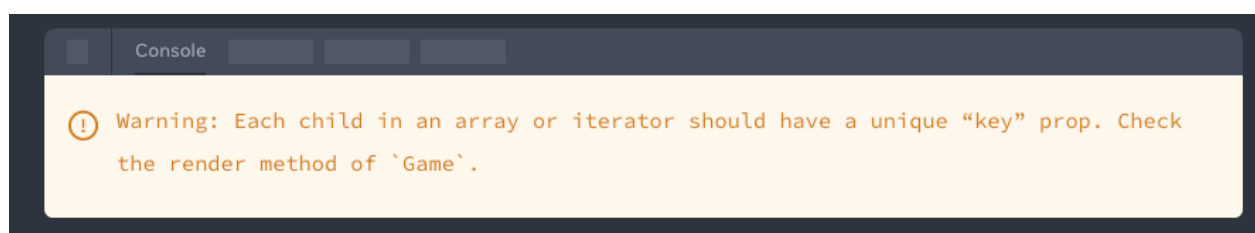
  function jumpTo(nextMove) {
    // TODO
  }

  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:



We will fix this error in the next section.

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 `` 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.

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>
  );
});
```

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:

Which will look like this

```
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 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.