ANMOL GUPTA | 005723941 | tic-tac-toe.txt | Assignment 3
---

1. Setup Local Development Environment
    a. Use 'npx create-react-app tic-tac-toe' to create a new react project
    b. Delete existing files in src folder and create index.css and index.js files
    c. Copy the starter code into these files and use 'npm start' to run code in the file.
    * css file is made already and isn't a focus in tutorial.

> TESTING: We see an empty tic-tac-toe grid in the browser. (http://localhost:3000/)

2. Components and Props
    a. Small and isolated pieces of code are called components in React, among which the React.Component is subclass which can be rendered in React. Inspecting the starter code we see 3 components: square, board and game.
    b. Creating a component: (Similarly we have for game and board.)

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

    c. A component takes in parameters, called props (short for â□□propertiesâ□□).
    d. To pass a prop from a parent Board component to a chile Square component, I did the following. 'this.props.value' changes the square's render method.

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

    * Render function occurs in each component.

> TESTING: We see the tic-tac-toe grid filled with the square number in the browser.

3. Making Interactive Components
    a. Method 1: If we change the button tag of the Square component to:

```
<button className="square" onClick={function() { console.log('click'); }}>
```

    Method 2: Using the arrow function syntax to avoid confusing behavior:

```
<button className="square" onClick={() => console.log('click')}>
```

    > TESTING: We see the word 'click' in the dev tools console by both methods (go to http://localhost:3000/ and click inspect then console to see the 'click' count increment)

b. To remember things, components use states set using private property 'this.state' in their constructors.

To remember if square has been clicked or not we add a constructor and initialize state to null. To display current status when clicked, we change the square's render method to:

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

* All React component classes that have a constructor should start with a super(props) call.
* setState in component automatically updates child component inside of it

> TESTING: When we click on a square, an X appears. If we re-click, the X remains.

4. Maintaining Game State: We need to declare shared state in parent component to collect data which will be passed down to the children. So, we will store the game's state in the parent Board component instead of in each square. To do this:

a. add constructor to Board and initialize 9 squares' state

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

b. modify board's renderSquare method

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

> Each square has a value prop holding 'X', 'O' or null.

c. We cannot update Board's state directly from Square because state is private to the component. Instead, we make a new function handleClick in renderSquare and make changes to the square render function and remove square's constructor.

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

```
        }
```

and

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

> TESTING: We have not defined the handleClick() method yet, so our code crashes and displays an error in the console.

d. To fix this, we add handleClick() to the Board class.

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

* Square components are now controlled components. The Board has full control over them.
* Immutability makes new objects with different parameters rather than changing parameters of one object. It's important for actions requiring previous versions or the undo feature.

> TESTING: Functionality runs with no errors and we are able to see Xs when we click on squares.

5. Some Simplification

We can make classes that only have a render method simpler by making them functions.

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

> TESTING: Everything works the same as it did in the last test.

6. Taking Turns

a. We can set X as the default first move by adding 'xIsNext: true' in our Board constructor.

b. To switch turns, we can update Boardâ□□s handleClick function to flip the value of xIsNext by:

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

c. To change the text that appears to shows the player turn, we make the change in Board's render:

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

> TESTING: Players O and X alternately show up when we click squares and text changes. If we re-click a square, the X changes to an O and vice versa.

7. Declaring a Winner
   a. Using the helper function calculateWinner we can note the winning combination of squares.
   b. To display the winner using this helper function, we modify the text in Board's render method.

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

   c. To return early if someone has won, we add the following to the handleClick function.

```
    if (calculateWinner(squares) || squares[i]) {
    return;
    }
```

> TESTING: O and X take turns alternately and when a player scores a winning combination, the text shows the winner and stops further moves.

>> We've built the game correctly!
