Adapted from: https://legacy.reactjs.org/tutorial/tutorial.html &
https://react.dev/learn/tutorial-tic-tac-toe

Introduction

In this lab, we'll build an interactive tic-tac-toe game with React. You can check out a complete example here.

If the code doesn't make sense to you, or if you are unfamiliar with the code's syntax, don't worry! The goal of this lab is to help you understand React and its syntax. **You may work in a language for quite some time before it becomes second nature to you. This doesn't mean you are *bad* at coding. It means you are human and humans need repetition to learn.**

The lab is divided into four sections:

1. **Setup for the Lab** will give you a starting point to follow the tutorial.
2. **Overview** will teach you the fundamentals of React: components, props, and state.
3. **Completing the Game** will teach you the most common techniques in React development.
4. **Adding Time Travel** will give you a deeper insight into the unique strengths of React.

Prerequisites

- Some familiarity with HTML and JavaScript, but you should be able to follow along even if you're coming from a different programming language.
- It is assumed that you're familiar with programming concepts like functions, objects, arrays, and to a lesser extent, classes.

Note: If you need to review JavaScript, read this guide.

Setup for the Lab

1. Make sure you have a recent version of Node.js installed.
2. Follow the installation instructions for Create React App to make a new project.

```
npx create-react-app tic-tac-toe
```

3. Delete all files in the src/ folder of the new project
    a. **Note: Don't delete the entire src folder, just the original source files inside it. We'll replace the default source files with examples for this project in the next step.**

```
cd tic-tac-toe

cd src
```

```
# If you're using a Mac or Linux:

rm -f *

# Or, if you're on Windows:

del *

# Then, switch back to the project folder

cd ..

cd ..
```

4. Download the lab files here
5. Unzip the archive to the /src folder that we emptied earlier.
6. Open a terminal and cd to the directory you unzipped
7. Install the dependencies with npm install (they may already be installed but this is a safe practice)
8. Run npm start to start a local server and follow the prompts to view the code running in a browser
    a. In the files, you will see `App.js`, `index.js`, `styles.css` and a folder called `public`
    b. Open the root folder in VS Code or your preferred IDE to see the files
9. Select `app.js` in your IDE. The contents should look like:

```
export default function Square() {

    return <button className="square">X</button>;

}
```

10. The *browser* section should be displaying a square with a X in it like this:

11. Now let's have a look at the files in the starter code.
    a. **App.js**
    b. The code in App.js creates a *component*. In React, a component is a piece of reusable code that represents a part of a user interface. Components are used to render, manage, and update the UI elements in your application. Let's look at the component line by line to see what's going on:

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


}
```

    c. The second line returns a button. The return JavaScript keyword means whatever comes after is returned as a value to the caller of the function. <button> is a *JSX element*. A JSX element is a combination of JavaScript code and HTML tags that describes what you'd like to display. className="square" is a button property or *prop* that tells CSS how to style the button. X is the text displayed inside of the button and </button> closes the JSX element to indicate that any following content shouldn't be placed inside the button.

index.js

12. Click on the file labeled index.js in your IDE. You won't be editing this file during this lab 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';
```

13. Lines 1-5 bring all the necessary pieces together:
    a. React
    b. React's library to talk to web browsers (React DOM)
    c. the styles for your components
    d. the component you created in App.js.
14. The remainder of the file brings all the pieces together and injects the final product into `index.html` in the `public` folder.

Overview

Building the board

1. Let's get back to `App.js`. This is where you'll spend the rest of the tutorial.
2. Currently the board is only a single square, but you need nine! If you just try and copy paste your square to make two squares like this:

```
export default function Square() {

 return <button className="square">X</button><button
className="square">X</button>;

}
```

You'll get this error:

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

3. 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* (`<>` and `</>`) to wrap multiple adjacent JSX elements like this:
   export default function Square() {

```
return (

      <>

<button className="square">X</button>

<button className="square">X</button>

</>);}
```

4. Now you should see:

5. Great! Now you just need to copy-paste a few times to add nine squares and…

6. Oh no! The squares are all in a single line, not in a grid like you need for our board. To fix this you'll need to group your squares into rows with `div`s and add some CSS classes. While you're at it, you'll give each square a number to make sure you know where each square is displayed. This leads us to our first challenge:

Challenge 1: Creating a grid

1. In the `App.js` file, update the `Square` component to provide line breaks between each 3 squares.
    a. Hint: The CSS defined in `styles.css` styles the divs with the `className` of `board-row`.
2. Your completed page should look like this:

3. Try and sort this out before moving forward

1. Great work on the challenge! There are many ways to achieve the results so as long as your view looks like the image above and you have no console errors, move forward!

2. We now have a problem. Your component named Square, really isn't a just square anymore. Let's fix that by changing the name to Board:

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

Passing data through props

3. Next, you'll want to change the value of a square from empty to "X" when the user clicks on the square. With how you've built the board so far you would need to copy-paste the code that updates the square nine times (once for each square you have)! Instead of copy-pasting, React's component architecture allows you to create a reusable component to avoid messy, duplicated code.

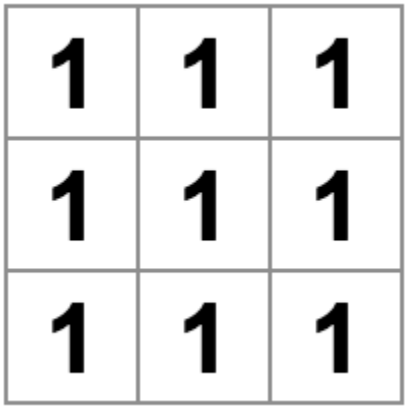
4. First, you are going to copy the line defining your first square (`<button classNam="square">1</button>`) from your `Board` component into a new `Square` component:

```
function Square() {

      return <button className="square">1</button>;

     }

     export default function Board() {

      // …

     }
```

5. Then you'll update the Board component to render that `Square` component using JSX syntax (code on following page):

```
// …
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>
      </>
 );
}
```

6. Note how unlike the browser `div`s, your own components `Board` and `Square` must start with a capital letter.
7. At this point your browser should show:



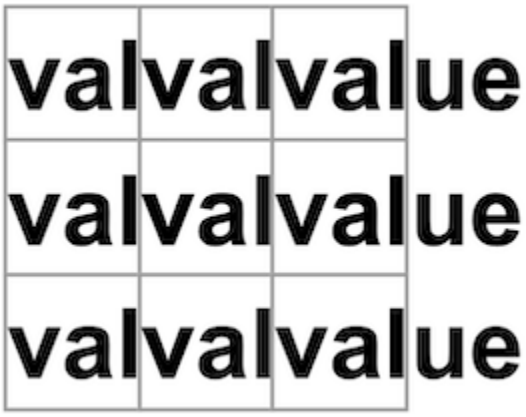
8. Oh no! You lost the numbered squares you had before. Now each square says “1”. To fix this, you will use *props* to pass the value each square should have from the parent component (`Board`) to its child (`Square`).
9. Update the `Square` component to read the `value` prop that you’ll pass from the `Board`

```
function Square({ value }) {

 return <button className="square">1</button>;

}
```

10. `function Square({ value })` indicates the Square component can be passed a prop called `value`.
11. Now you want to display that `value` instead of `1` inside every square. Try doing it like this:

```
function Square({ value }) {

 return <button className="square">value</button>;

}
```

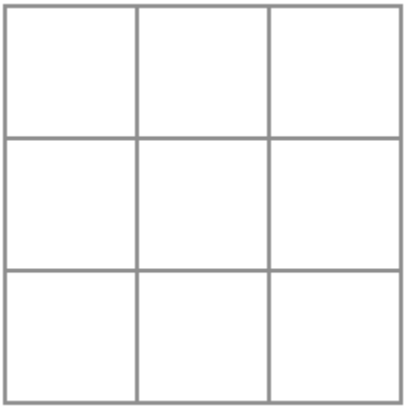
12. Oops, this is not what you wanted:



13. You wanted to render the JavaScript variable called `value` from your component, not the word “value”. To “escape into JavaScript” from JSX, you need curly braces. Add curly braces around `value` in JSX like so:

```
function Square({ value }) {

     return <button className="square">{value}</button>;

    }
```

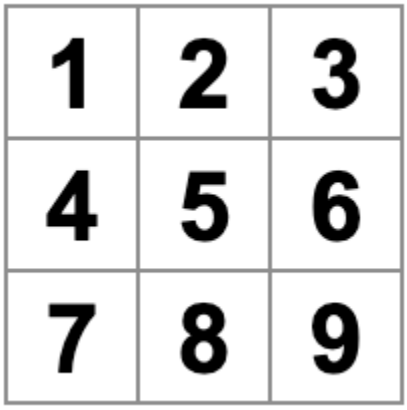
14. For now, you should see an empty board:



15. This is because the `Board` component hasn’t passed the `value` prop to each `Square` component it renders yet. To fix it you’ll add the `value` prop to each `Square` component rendered by the `Board` component:

```
export default function Board() {
      return (
       <>
        <div className="board-row">
         <Square value="1" />
         <Square value="2" />
        <Square value="3" />
       </div>
        <div className="board-row">
         <Square value="4" />
         <Square value="5" />
         <Square value="6" />
       </div>
        <div className="board-row">
         <Square value="7" />
         <Square value="8" />
         <Square value="9" />
       </div>
      </>
     );
     }
```

16. Now you should see a grid of numbers again:



Challenge 2: Make it interactive!

1. Let's fill the `Square` component with an `X` when you click it. Declare a function called `handleClick` inside of the `Square`. Then, add `onClick` to the props of the button JSX element returned from the `Square`.
    a. Hint: `onClick={handleClick}`

2. Next, add a console print to let us know it is working.
    a. Hint:
```
function handleClick() {
    console.log('clicked!');
}
```
3. If you click on a square now, you should see a log saying `"clicked!"` in the *Console* tab at the bottom of the *Browser* section in CodeSandbox. Clicking the square more than once will log `"clicked!"` again. Repeated console logs with the same message will not create more lines in the console. Instead, you will see an incrementing counter next to your first `"clicked!"` log.
    a. **Note: You need to open your browser's Console to see what is logged. For example, if you use the Chrome browser, you can view the Console with the keyboard shortcut Shift + Ctrl + i (on Windows/Linux) or Option + ⌘ + i (on macOS). Using j instead of i in the command will open the console in a new window if that is preferable.**

Conclusion

In our next lab, you will make the Square component "remember" that it got clicked, and fill it with an "X" mark. To "remember" things, components use *state*.

END OF PART A

Part B can be found at:

https://react.dev/learn/tutorial-tic-tac-toe

Start at the heading “Making an interactive component”