React Training

with examples and
hands-on exercises

WEBUCATOR

Version: 2.0.2

The Authors

Chris Minnick

Chris Minnick, the co-founder of WatzThis?, has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies. A prolific writer, Chris has authored and co-authored books and articles on a wide range of Internet-related topics including HTML, CSS, mobile apps, e-commerce, e-business, Web design, XML, and application servers. His published books include Adventures in Coding, JavaScript For Kids For Dummies, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW E-Commerce Designer Certification Bible, and XHTML.

Nat Dunn (Editor)

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/REA101-2.0.2.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Table of Contents

LESSON 1

Introduction to React

Topics Covered

☑ The benefits of writing user interfaces with React.

☑ Data flow in a React user interface.

☑ React components.

☑ Create React App.

Introduction

React is a JavaScript library for creating dynamic website user interfaces. A dynamic website user interface is one that changes (or “reacts”) in response to events, such as the user clicking a button or typing into a form. This lesson introduces you to the most important concepts and terms that are used in React.

1.1. What is React?

React was created by Facebook to make it easier to make dynamic websites. A company as large as Facebook faces several challenges when designing and maintaining their website. The website must be:

1. Fast.
2. Scalable, so that it can grow and support increasing numbers of users.
3. Reliable.

Here’s how React addresses each of these challenges:

❖ 1.1.1. React is Fast

When you make use of content on the web, you use a program called a *user agent*. Most of the time, that user agent is a web browser.

The content that is displayed in a web browser can change over time using two different methods:

1. **Loading a new page**: This can be done by entering a new address into a browser's address bar, clicking a link, or submitting a form that loads a new HTML page.
2. **Using JavaScript to modify the current page**: Updating the web page using JavaScript can result in a better user experience, because the whole page doesn't need to be reloaded with each change. This allows for effects like "infinite scroll," in which images or posts are loaded as you scroll down a page (similar to a Facebook or Twitter feed), and for web applications like email readers and web-based financial programs to rival the user experience of "native" desktop and mobile applications.

The Document Object Model (DOM) is the Application Programming Interface (API) for the web browser. It gives JavaScript programs access to the properties and functionality of the web browser. Using the DOM, it is possible to add content, remove content, rearrange content, and load new content into a web browser. We call using the DOM to change the web page contents "DOM Manipulation" and it is at the heart of modern web applications.

However, all of this changing of content using the DOM has a cost. If you don't do it right, or if you do too much of it, your web page (or mobile app) will seem sluggish to the user. With React, Facebook rethought how DOM manipulation is done to minimize and optimize it as much as possible.

❖ 1.1.2. React is Scalable

React allows you to assemble (or "compose") user interfaces using reusable components. Every React application is made up a collection of components that you can use like HTML elements to build small, simple applications and large complex ones. As a programmer, you can think about your user interface as if the entire thing is constantly being re-rendered, and React manages actual updates to what appears in the browser so as to give the user the best and fastest experience possible.

❖ 1.1.3. React is Reliable

React is currently used in many of the largest and most visited websites and mobile apps in the world. It is well-supported by legions of developers and, when used correctly, it is a dependable and stable library. Because React is only concerned with the user interface, the actual React library is relatively small when compared with more full-featured frameworks such as Angular, Ember, or Dojo. When something does go wrong in a React application, it is usually because of the incorrect use of a JavaScript feature rather than a problem with React itself. In general, React requires developers to have a better understanding of JavaScript, and having this advanced JavaScript knowledge makes for more robust applications.

1.2. React Essentials

Let's examine the fundamental aspects of React, including how it renders components, manages data, and updates the browser.

Node, npm, and npx

React developers use Node to compile and test their applications. You should have installed Node when setting up for this course. Follow the instructions at `https://www.webucator.com/article/nodejs-and-node-package-manager-npm/` to check whether you have the latest version of Node installed and to install it if not.

Node comes with *npm*, a Node package manager, and *npx*, which downloads and executes a Node package in a single step. npx is used to create React applications.

❖ 1.2.1. Rendering in React

React components are made up of JavaScript modules that define React elements. A React component is a JavaScript function that renders a piece of the HTML user interface. The following demo shows a basic React component that simply prints out a message to the browser:

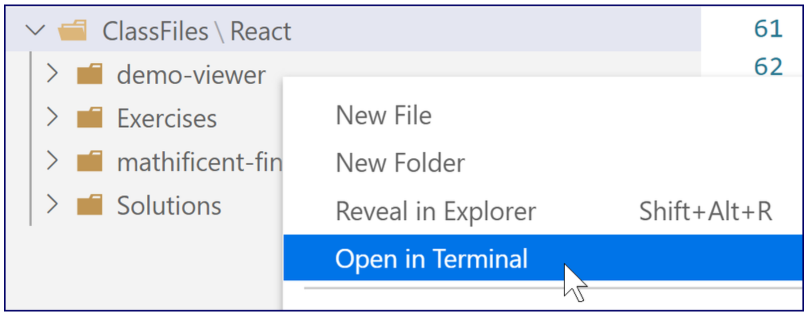
Demo 1.1: React/demo-viewer/src/Demos/SayHello.js

```
import React from 'react';

function SayHello() {
    return (
      <h1>Hello, world!</h1>
    );
}

export default SayHello;
```

Code Explanation

1. Each React component contains a `return` statement. When assembled together, these React components and their `return` statements define how the final user interface will look and behave.

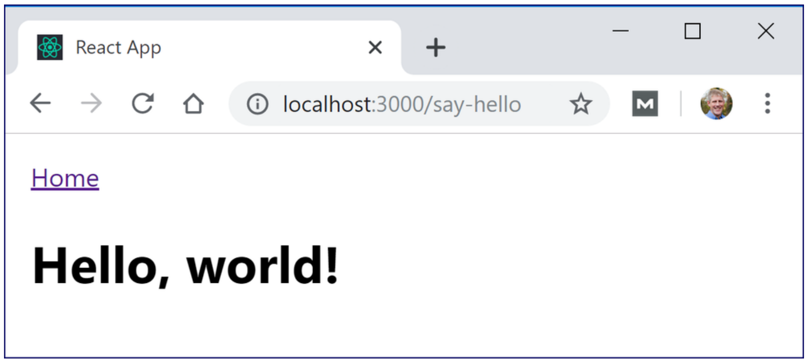
2. To run this file, open `React/demo-viewer` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



3. You must first install npm in the `demo-viewer` folder by running:

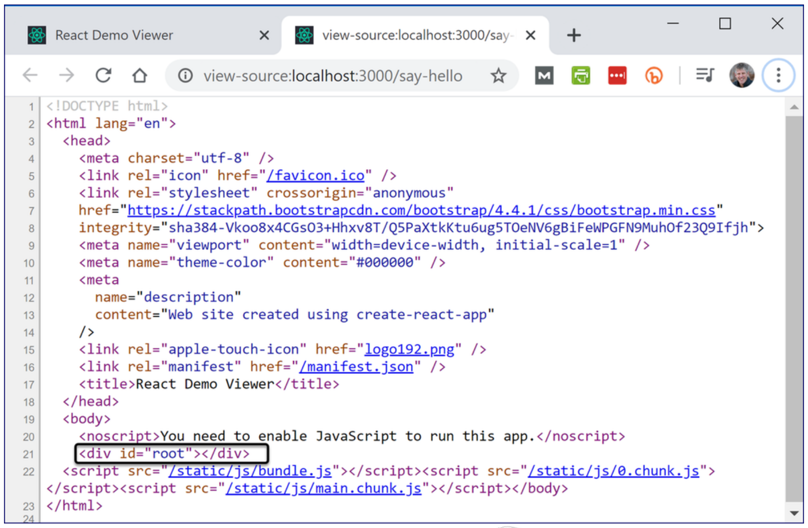
```
npm install
```

4. Once npm is installed, run `npm start`. This will launch your React app using a development server. A development server is a web server that runs on a single software developer's computer and makes it possible for the developer to test out code as it is written and modified, without having to make it available for use by the entire internet. After the development server is launched, a web browser window will open at `http://localhost:3000` showing our demo-viewer React application. Click the **SayHello** link under **Introduction to React**. You should see a page that looks like this:



Hello, world!

5. Right-click the page in the browser and select **View page source**. You should see the following:


React Demo Viewer
view-source:localhost:3000/say-
view-source:localhost:3000/say-hello

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="/favicon.ico" />
    <link rel="stylesheet" crossorigin="anonymous"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
    integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="logo192.png" />
    <link rel="manifest" href="/manifest.json" />
    <title>React Demo Viewer</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  <script src="/static/js/bundle.js"></script><script src="/static/js/0.chunk.js">
</script><script src="/static/js/main.chunk.js"></script></body>
</html>
```

Notice that the “Hello, world!” `div` does not show up in the page source. That is because it was added using DOM manipulation after the browser loaded the page. However, that happens so fast, that the user never sees that change.

6. Back in the terminal, press **CTRL+C** to stop the app.
7. Close the terminal by pressing the trash can icon.

Class Components

The component in the preceding demo is a *function component*, meaning it is created using a JavaScript function. You can also create React components using classes. Doing so requires more advanced knowledge of JavaScript. In modern versions of React (16.8 and later), function components are much more flexible. As such, many React developers are switching from writing class components to function components. In this course, we only teach function components.

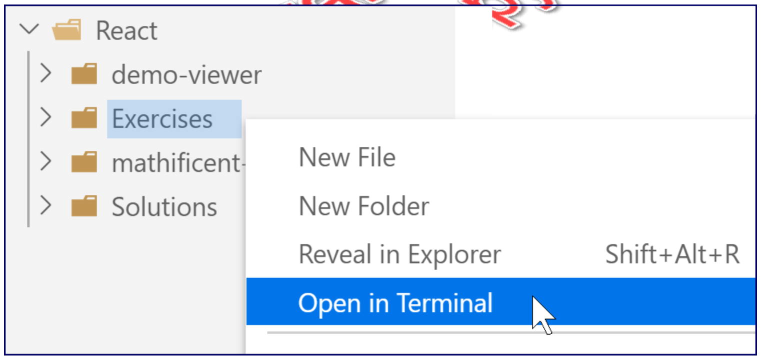
Exercise 1: Get Started with Create React App

15 to 25 minutes

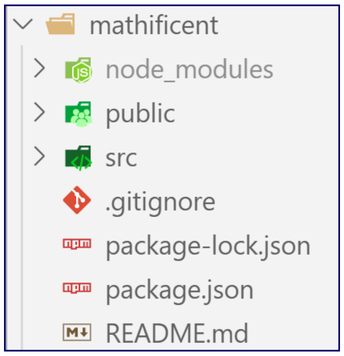
In this exercise, you will use a Node package called **Create React App** to create your first React application, which will serve as the starting point for the math game we'll be building in the rest of the course. After you've built your React application, you'll use npm to package and deploy the application to a development server and open it in a browser.

Best of all, most of the process of building a simple application and installing the Node packages and scripts that make it run is done by Create React App. This makes it easy for anyone with even the most basic knowledge of JavaScript and React to quickly start working on a project. So, let's jump in!

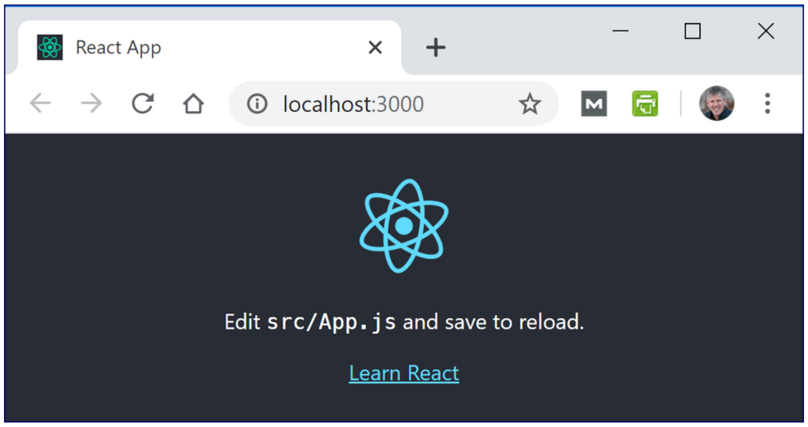
1. From your class files, open `React/Exercises` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



2. Run `npx create-react-app mathificent` in the terminal to create a starter project named `mathificent`. The dependencies will be downloaded and after a few minutes you'll have a new React project.
3. Run `cd mathificent` to make your new React project the working directory. At this point, your first React program has been created and you can look at the individual files that it is made of:



4. Run `npm start`. It will start the local development server and open a browser window with your new React website:



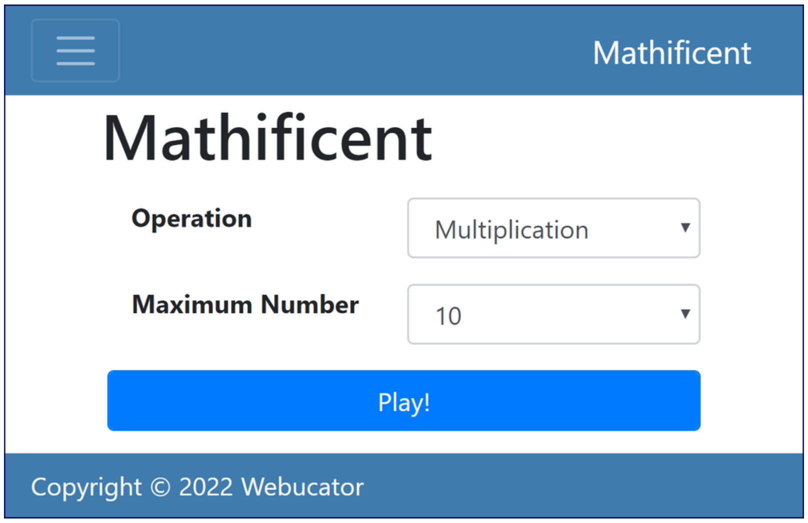
Edit `src/App.js` and save to reload.

5. Back in the terminal, press **CTRL+C** to stop the app.
6. Close the terminal window by pressing the trash can icon.

1.3. Introducing Our Project: Mathificent

Throughout these lessons, you'll be using the latest React syntax and techniques to build a single-page application for practicing arithmetic. The application you'll be building is based on the game Mathificent, which you can view at `https://www.mathificent.com`. It consists of the following three views:

Config View



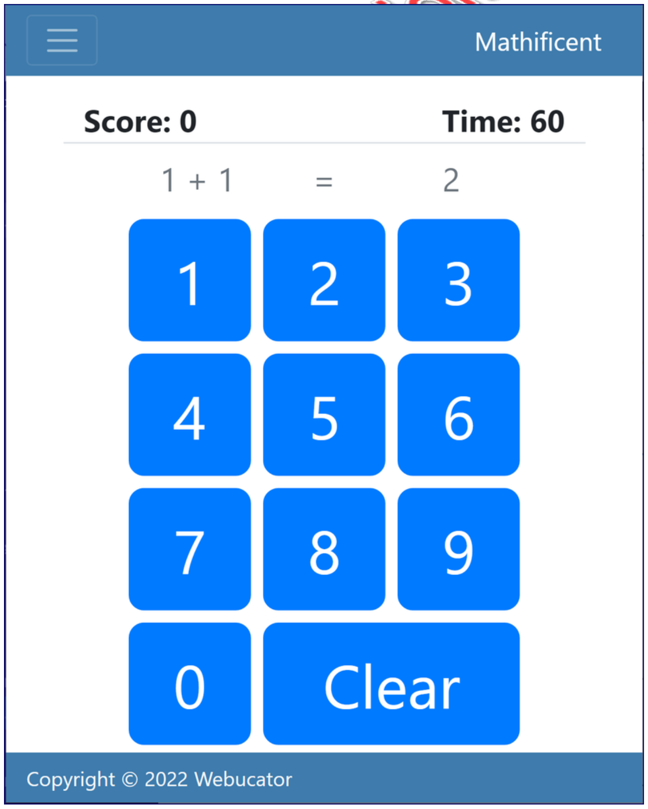
Mathificent

Operation: Multiplication

Maximum Number: 10

Copyright © 2022 Webucator

Game View

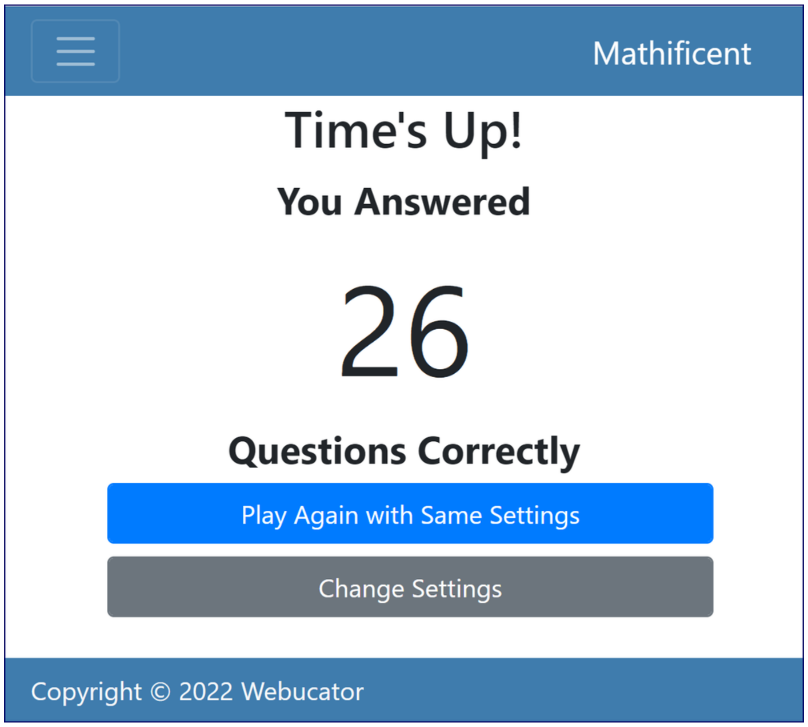


Score: 0 Time: 60

1 + 1 = 2

Copyright © 2022 Webucator

Times-Up View



Time's Up!

You Answered

26

Questions Correctly

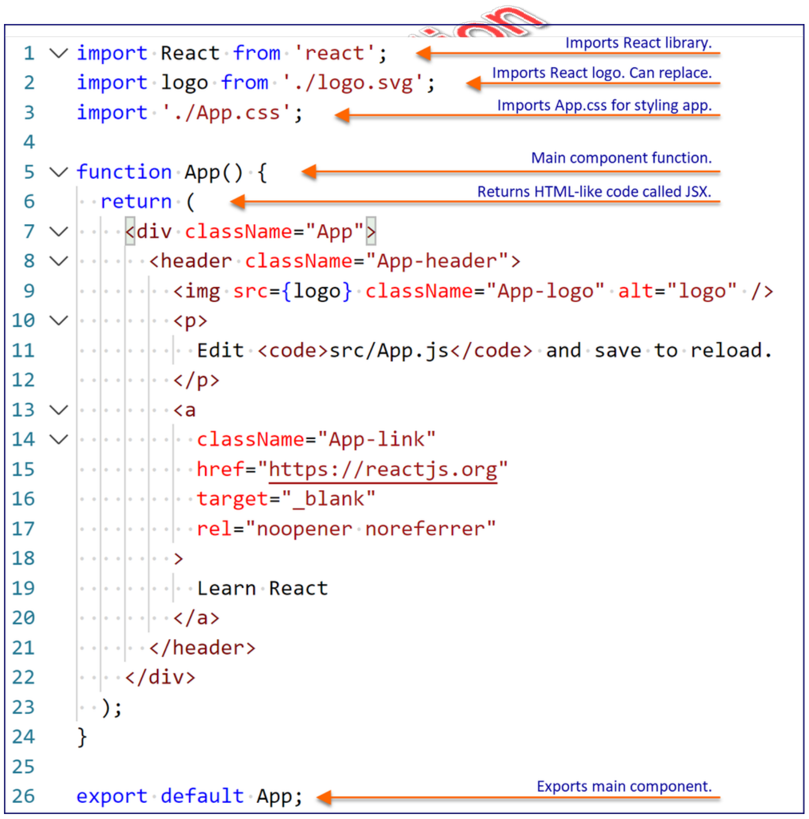
Copyright © 2022 Webucator

Exercise 2: Learning the Structure of a React App

15 to 25 minutes

In this exercise, you will start with the boilerplate Create React App project you created and make some modifications to learn about the different files involved in React applications and the different parts of those files.

1. From the `mathificent` directory, open `src/App.js` for editing.
2. Take a look at the structure of this file. It imports the React library and contains a function with a `return` statement. This function creates a component. The job of a component is to return (output) a piece of the user interface. The final statement in a React component is the `export` statement, which makes the file into a JavaScript module that can be imported into other files.



```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Imports React library.
Imports React logo. Can replace.
Imports App.css for styling app.
Main component function.
Returns HTML-like code called JSX.
Exports main component.

3. Open `src/index.js` in your editor. This is the main JavaScript file for the entire React application. This file is the only place in your application that imports and uses the `ReactDOM/client` library, which handles rendering of elements into the DOM:

```
import ReactDOM from 'react-dom/client';
ReactDOM.createRoot(document.getElementbyId('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

React.StrictMode

The `React.StrictMode` element is used to highlight potential problems with your application. It is not required. The preceding code could be written like this:

```
ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

For more information, see `https://reactjs.org/docs/strict-mode.html`

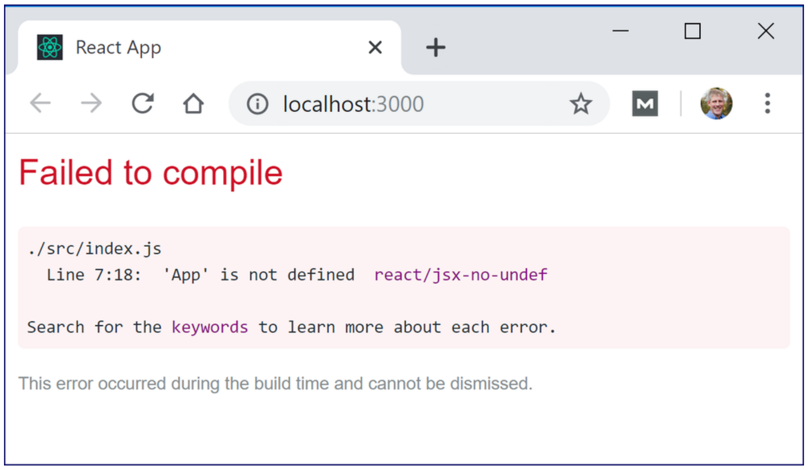
4. In `index.js`, comment out the `import` statement that imports `App`:

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

5. Save your file.
6. If your development server isn't already running, start it by running:

```
npm start
```

7. Once the development server starts up, your default web browser will load `http://localhost:3000`, where you'll see an error message, as shown here:



Failed to compile

```
./src/index.js
  Line 7:18:  'App' is not defined  react/jsx-no-undef

Search for the keywords to learn more about each error.
```

This error occurred during the build time and cannot be dismissed.

8. Read the error message, then return to your editor an remove the single-line comment before the `import` statement and save the file.

9. Return to your web browser and the application should refresh and be working again. Because React applications are made up of components that are linked together using `import` statements and any one file may have many `import` statements, one of the most common errors that you will see in React development is caused by a component or file not being imported, or not being imported correctly.

10. Look at the `ReactDOM.createRoot()` statement in `index.js`. `ReactDOM.createRoot()` is a method whose job it is to control the contents of an element in a web page. `ReactDOM.createRoot()` takes one argument, the location where you want to render a component. Once you've used `createRoot()` to specify this location, you can use the `render()` method to actually render a component. The `render()` method takes one argument: the component you want to render. Because every component in a React application is linked together, `render()` only needs to specifically render a single component, and that single component contains all the other components in the application. We call the single component that contains all the others the "root" component. In our application, `App` is the root component:

```
root.render(<App />);
```

11. Look at the argument passed to `ReactDOM.createRoot()`:

```
ReactDOM.createRoot(document.getElementById('root'));
```

This is a DOM method that locates an HTML element in the browser using the value of its `id` attribute. In a default Create React App program, the `id` attribute value

`ReactDOM.createRoot()` looks for is "root", but it can be anything, as long as that element exists and you reference it correctly in the `ReactDOM.createRoot()` method call.

12. Open `public/index.html` in your editor. This is the HTML file that is loaded when your web browser loads `http://localhost:3000`. Find the element with the "root" id:

    ```
    <div id="root"></div>
    ```

    This is where `index.js` will render the root component for your application.

13. Notice that `index.html` doesn't have any code that imports `index.js`. This is because `index.html` is a template. When you start the Create React App development server (using `npm start`), the code in `index.js` (and therefore everything that it imports) is linked to the `index.html` with `<script>` tags before the page opens in your web browser.

14. Open `App.js` in your editor.

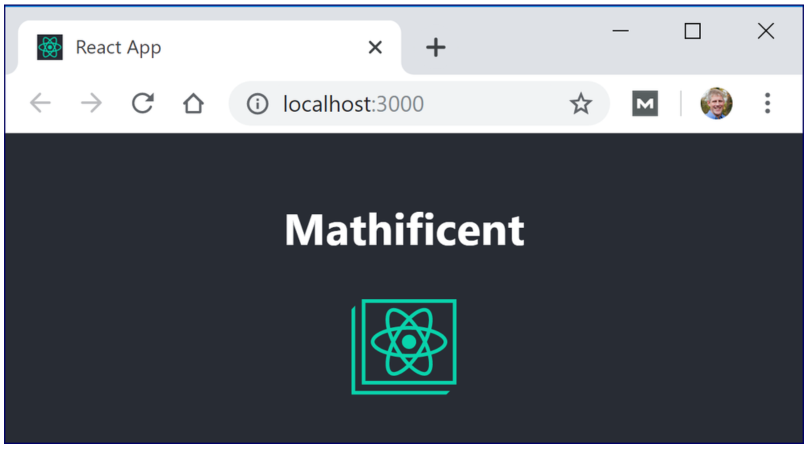
15. Find the `<header>` and `</header>` tags and delete everything between them except for the `img` tag. Your `App.js` file should now look like this:

    ```
    import React from 'react';
    import logo from './logo.svg';
    import './App.css';

    function App() {
      return (
        <div className="App">
          <header className="App-header">
            <img src={logo} className="App-logo" alt="logo" />
          </header>
        </div>
      );
    }

    export default App;
    ```

16. Insert an `<h1>` element above the image, containing the title of our app:

    ```
    <h1>Mathificent</h1>
    ```

17. Return to your web browser where the app is open and running. If you did everything correctly, you'll see the following:



Mathificent

.

18. If you see an error message, return to your `App.js` file and make sure that it matches the solution exactly.
19. Remember to stop the app (**CTRL+C**) and close the terminal when you are done.

Solution: React/Solutions/introduction-to-react/App.js

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>Mathificent</h1>
        <img src={logo} className="App-logo" alt="logo" />
      </header>
    </div>
  );
}

export default App;
```

Conclusion

In this lesson, you have learned about the essential components of a React application, including how React uses JavaScript modules to define components that can be assembled together using React elements.

LESSON 2
JSX and React Elements

Topics Covered

☑ The role of JSX in React.

☑ JSX vs. HTML.

☑ Expressions in JSX.

Introduction

JSX is the template language used by React to generate HTML user interfaces.

2.1. Using JSX in React

JSX is an XML-based template language that simplifies the use of React to produce HTML. When used in a React component's `return` statement for a function component, JSX looks and behaves similarly to HTML code. However, behind the scenes, JSX is actually just a shorthand for React's `createElement()` function. To illustrate, take a look at the following React component, which we saw in the previous lesson:

Demo 2.1: React/demo-viewer/src/Demos/SayHello.js

```
import React from 'react';

function SayHello() {
    return (
      <h1>Hello, world!</h1>
    );
}

export default SayHello;
```

This component does what you would think: it causes an HTML `h1` element containing the words "Hello, world!" to be rendered. The JSX code is everything inside of the `return` statement. Inside

React, this JSX code gets compiled into a JavaScript statement before it runs in the browser. Here's what it looks like compiled into JavaScript:

```
import React from 'react';

function SayHello() {
  return (
    React.createElement("h1", null, "Hello, world!");
  );
}

export default SayHello;
```

JSX was created by Facebook to give programmers an easier way to write `React.createElement()` statements. When you start to have nested elements with attributes, it quickly becomes apparent how much easier JSX is to read than the native React JavaScript. For example, consider the following component:

Demo 2.2: React/demo-viewer/src/Demos/JSXForm.js

```
import React from 'react';

function JSXForm() {
    return (
      <div>
        <h1>Your Name Please</h1>
        <form>
          <label htmlFor="first-name">First Name: </label>
          <input type="text" id="first-name" /><br/>
          <label htmlFor="last-name">Last Name: </label>
          <input type="text" id="last-name" /><br/>
          <button>Submit</button>
        </form>
      </div>
    );
}

export default JSXForm;
```

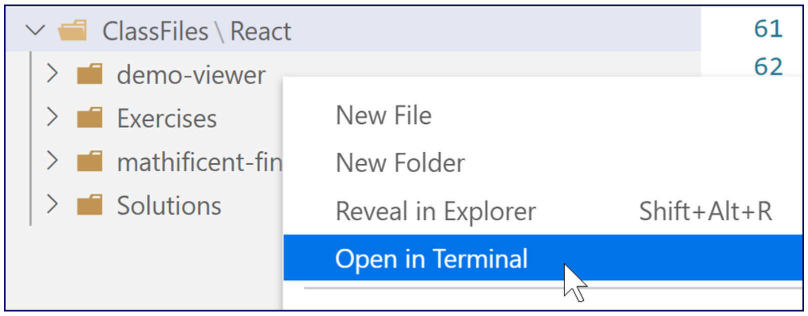
Code Explanation

The `JSXForm` component shown in the demo above returns an HTML form written in JSX. Written using native `React.createElement()` statements, that same form would be coded like this:

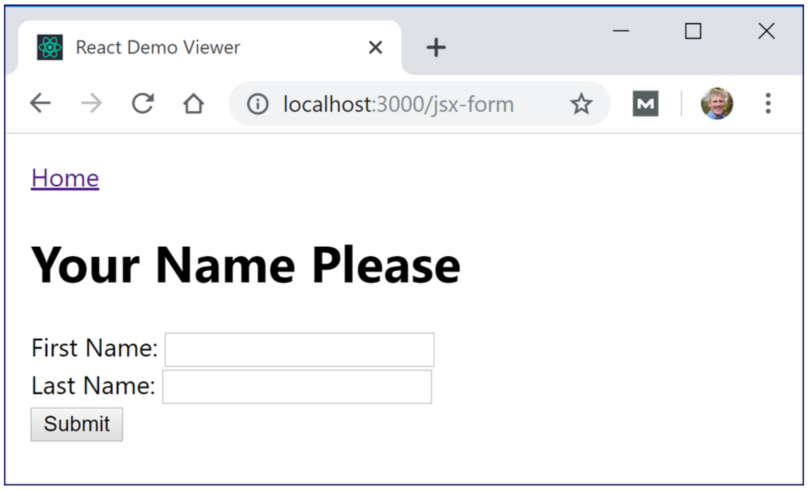
```
React.createElement("div", null,
  React.createElement("h1", null,  "Your Name Please"),
  React.createElement("form", null,
    React.createElement("label",
    {
      htmlFor: "first-name"
    }, "First Name: "),
    React.createElement("input",
    {
      type: "text",
      id: "first-name"
    }),
    React.createElement("br", null),
    React.createElement("label",
    {
      htmlFor: "last-name"
    }, "Last Name: "),
    React.createElement("input",
    {
      type: "text",
      id: "last-name"
    }),
    React.createElement("br", null),
    React.createElement("button",
    {
      onClick: "submitForm"
    }, "Submit")
  )
);
```

The JSX version is much easier to read and to visualize than the version written using `React.createElement()` statements.

1. To run this file, open `React/demo-viewer` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



2. Run `npm start` to launch the demo-viewer React application.
3. Click the **JSXForm** link under **JSX and React Elements**. You should see a page that looks like this:



Your Name Please

First Name:
Last Name:
Submit

4. Back in the terminal, press **CTRL+C** to stop the app.
5. Close the terminal by pressing the trash can icon.

2.2. JSX Rules

❖ 2.2.1. JSX Syntax

1. JSX is XML-based. That means all tags, including empty tags, must be closed. The following code is not valid JSX:

   ```
   <img src="logo.png" alt="Logo">
   ```

   Adding the shortcut close makes it valid:

   ```
   <img src="logo.png" alt="Logo" />
   ```

2. Tags that are meant to be output as HTML must be all lowercase.
3. Tags that reference React components must be UpperCamelCase.
4. JSX attributes are written in lowerCamelCase. For example, `onclick` becomes `onClick`.

❖ 2.2.2. JSX is an Extension of JavaScript

While JSX looks a lot like HTML, it is important to remember that it is in fact an extension of JavaScript. And because JavaScript and HTML share some of the same keywords, JSX cannot use all of the HTML keywords. For example, in the JSX example from earlier, the `label` element includes an attribute called `htmlFor`:

```
React.createElement("label",
{
  htmlFor: "first-name"
}
```

In HTML, this attribute, which indicates what the label relates to, is the `for` attribute. However, because JSX is actually JavaScript code and because `for` has a special meaning in JavaScript, it is necessary to use `htmlFor` instead of `for` in JSX.

In addition to `htmlFor`, the other common change that you need to be aware of when you write JSX is to the `class` attribute. The HTML `class` attribute is used to apply CSS classes to an HTML element. However, `class` has a completely different meaning in JavaScript. To avoid conflict, `class` needs to be changed to `className` in JSX.

❖ 2.2.3. Using Custom Elements in JSX

The result of defining a React component is a React element that has the same name as the component that can be used inside other components. Custom React JSX element names always start with a capital letter, as opposed to the HTML equivalent elements, which always start with lower case letters.

This is perhaps the most important thing to understand about JSX in React: when you write JSX code, you are not directly writing the HTML that will appear in the browser. Instead, you are writing the JavaScript that will write the HTML in the browser.

❖ 2.2.4. Using JavaScript in JSX

Since JSX is a template language for writing JavaScript, it is easy to include any bit of JavaScript code that you want to run inside your JSX. The only difference between writing JavaScript code inside of JSX and writing it in a function is that you need to surround any JavaScript inside of JSX code with curly braces (`{` and `}`) to indicate that it shouldn't be interpreted as JSX.

As we will see in the following exercise, it is easy to use variables and simple conditional logic inside of JSX code.

Exercise 3: Using JSX

15 to 25 minutes

In this exercise, you will use JSX to create a user interface in React.

1. To add style to the app, we will use Bootstrap:

   A. Open `index.html` from the `mathificent/public` directory in Visual Studio Code.

   B. Copy the following `<link>` tag:

   ```
   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/boot  ↵↵
   strap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhf  ↵↵
   TAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
   ```

   Paste it in the `head` in `index.html`.

   C. Copy the following `<script>` tag:

   ```
   <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/boot  ↵↵
   strap.bundle.min.js" integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wX  ↵↵
   gYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
   ```

   Paste it immediately before the close `</body>` in `index.html`.

   D. Change the `title` from “React App” to “Mathificent!”

2. Open `src/App.js` in your editor.

3. Inside the `return` statement of the function, replace the existing `<header>` element with the header navigation for our app:

```
<header>
  <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
    <div className="container-fluid">
      <button className="navbar-toggler" type="button"
        data-bs-toggle="collapse" data-bs-target="#navbarText">
          <span className="navbar-toggler-icon"></span>
      </button>
      <div className="collapse navbar-collapse" id="navbarText">
          <ul className="navbar-nav mr-auto text-left">
              <li className="nav-item active">
                  <a className="nav-link" href="/">Home</a>
              </li>
          </ul>
      </div>
      <a className="navbar-brand" href="/">Mathificent</a>
    </div>
  </nav>
</header>
```

A Shortcut: Copy and Paste

You can copy and paste from `Exercises/starter-code.txt` if you would prefer not to type this out. You will find both this `header` element and the `footer` element (shown in step 7) in that document. If you do so, be sure to review both carefully, so you understand what's going on. They both include some Bootstrap classes, and the `footer` includes some JavaScript enclosed in curly braces.

4. Because the app no longer uses the React logo, delete the `import` statement that imports the logo.
5. Save `App.js`.
6. Below the header, add the `<h1>` element containing the name of the app:

```
<h1>Mathificent</h1>
```

7. Under the `h1` element, type the footer:

```
<footer className="navbar fixed-bottom bg-dark">
    <div className="container-fluid">
      <a href="https://www.webucator.com" className="nav-link text-light">
        Copyright &copy; {new Date().getFullYear()} Webucator
      </a>
    </div>
  </footer>
```

The JavaScript in the footer will dynamically populate the copyright year. Notice that it is enclosed in curly braces. This lets React know that it should be interpreted.

8. If your app isn't already running, `cd` to your `mathificent` directory and run `npm start` and then open your browser to `http://localhost:3000` (if it doesn't automatically open). You should now have a header, main content of the page with the `h1` element, and a footer.

Solution: React/Solutions/jsx-react-elements/App.js

```
import React from 'react';
  import './App.css';

  function App() {
  return (
  <div className="App">
      <header>
      <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
        <div className="container-fluid">
          <button className="navbar-toggler" type="button"
            data-bs-toggle="collapse" data-bs-target="#navbarText">
            <span className="navbar-toggler-icon"></span>
          </button>
          <div className="collapse navbar-collapse" id="navbarText">
            <ul className="navbar-nav mr-auto text-left">
              <li className="nav-item active">
                <a className="nav-link" href="/">Home</a>
              </li>
            </ul>
          </div>
          <a className="navbar-brand" href="/">Mathificent</a>
        </div>
      </nav>
    </header>
      <h1>Mathificent</h1>
      <footer className="navbar fixed-bottom bg-dark">
        <div className="container-fluid">
          <a href="https://www.webucator.com" className="nav-link text-light">
            Copyright &copy; {new Date().getFullYear()} Webucator
          </a>
        </div>
      </footer>
  </div>
  );
}

export default App;
```

Code Explanation

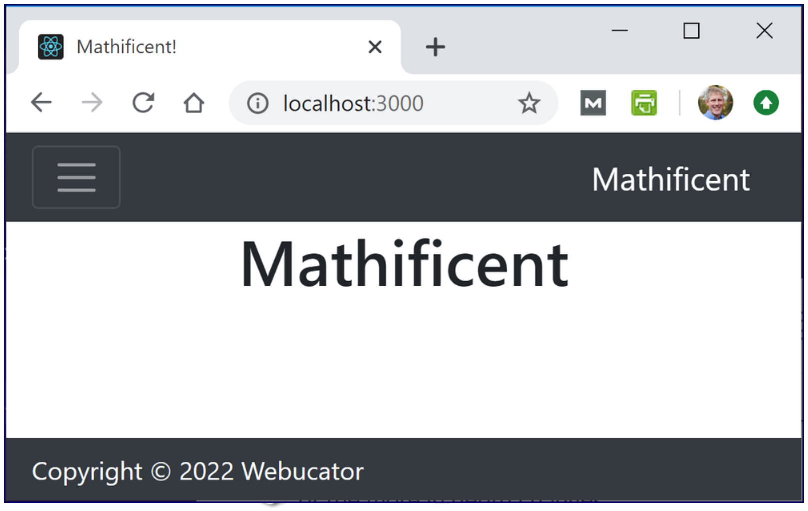
The preceding code shows the complete `App.js` file.

Solution: React/Solutions/jsx-react-elements/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
  rel="stylesheet"
  integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
  crossorigin="anonymous"
/>
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta
    name="description"
    content="Web site created using create-react-app"
  />
  <link rel="apple-touch-icon" href="logo192.png" />
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <title>Mathificent!</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <script
  src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
  integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
  crossorigin="anonymous"
></script>
</body>
</html>
```

Code Explanation

This is the HTML code that should be in `public/index.html`. It has the Bootstrap CSS and JavaScript added. When you run the Mathificent React app (using `npm start`), the page should now look like this:



Mathificent

Copyright © 2022 Webucator

Remember to stop the app (**CTRL+C**) and close the terminal when you are done.

Conclusion

In this lesson, you have learned about JSX, which is the template language used to change the DOM.

LESSON 3
React Components

Topics Covered

☑ Best practices for writing React components.

☑ Passing data between components with `props`.

Introduction

React components are the building blocks of React user interfaces. In this lesson, you'll learn how to put those building blocks together.

3.1. Assembling User Interfaces

Every React application consists of at least one React component, called the `root` component. Although it is fully possible to write an entire React application with only one component, doing so would defeat the purpose of React and make your code difficult to maintain.

❖ 3.1.1. Understanding F.I.R.S.T.

User interface design can be extremely complex. Different screens need to be shown based on where users are in a website or process, data needs to be sent to and received from a server, every user interaction can trigger an event, and modern web applications need to run on desktop, tablet, and smart phone devices. Rather than trying to understand and build an entire web application at once, web developers are increasingly adopting a technique that's long been standard practice in the rest of the software development world: namely, reusable components.

One common way of remembering the best practices for how to design React components is the rule of F.I.R.S.T. This rule says that React components should be:

- **F**lexible
- **I**ndependent
- **R**eusable
- **S**mall

- Testable

The benefits of designing components with the rule of F.I.R.S.T. in mind are that your components:

1. will be less likely to have errors.
2. will be easier to work with on a team.
3. will be easier to write.
4. will be easier to maintain.

In general, if you find yourself writing a React component that has more than one purpose, that's a good indicator that it is time to break that component into separate smaller components that each only do one thing.

❖ 3.1.2. Passing Data with Props

Once you've written components that comply with the rule of F.I.R.S.T., the next step in making useful components is to be able to pass data and functionality into them so that the same component can produce different results when given different data.

The `props` object is used for passing data into React components. Props are simply JSX attributes that you specify when you use a component. These attributes can have any valid name. Inside of the component, these attributes become accessible as properties in the `props` object.

For example, in the following component, we use several instances of a React component named `HelloMessage`. With each use of the `HelloMessage` component, we pass in a different value for the `helloTo` attribute.

Demo 3.1: React/demo-viewer/src/Demos/Greeting.js

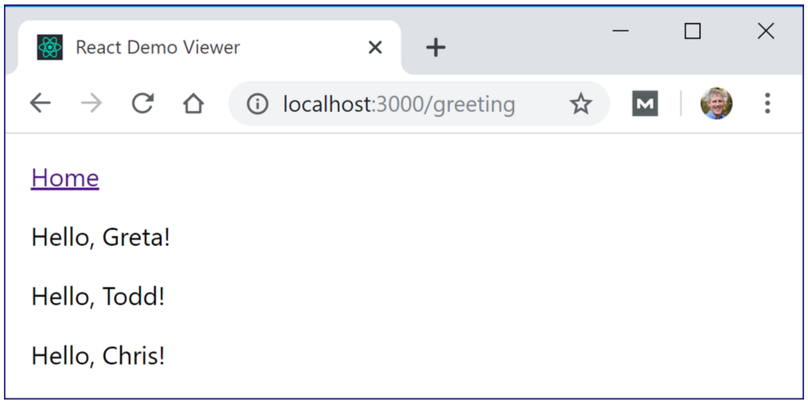
```
import React from 'react';
import HelloMessage from './HelloMessage';

function Greeting() {
    return (
      <div>
        <HelloMessage helloTo="Greta" />
        <HelloMessage helloTo="Todd" />
        <HelloMessage helloTo="Chris" />
      </div>
    );
}

export default Greeting;
```

Inside the `HelloMessage` component, you can now define a parameter in the function header called `props` and use the values passed into the component as properties inside the `props` object, like this:

Demo 3.2: React/demo-viewer/src/Demos/HelloMessage.js

```
import React from 'react';

function HelloMessage(props) {
    return (
     <p>Hello, {props.helloTo}!</p>
    );
}

export default HelloMessage;
```

When the `Greeting` component is rendered in a browser, it will look like this:



Hello, Greta!

Hello, Todd!

Hello, Chris!

Note that when you pass variables from a parent to a child component, you wrap the variable in curly braces instead of quotation marks:

Demo 3.3: React/demo-viewer/src/Demos/Greeting2.js

```
import React from 'react';
import HelloMessage from './HelloMessage';

function Greeting2() {
    const firstName = 'Nat';
    return (
      <div>
        <HelloMessage helloTo="Greta" />
        <HelloMessage helloTo="Todd" />
        <HelloMessage helloTo="Chris" />
        <HelloMessage helloTo={firstName} />
      </div>
    );
}

export default Greeting2;
```

Exercise 4: Breaking an App into Components

45 to 60 minutes

In this exercise, you will break up the user interface of the Mathificent game into subcomponents.

Although it is possible to write an entire React application in a single component, what makes React useful is that it allows you to break your user interface into components which can be reused. So, let's make some components!

1. Create a new file in the `mathificent/src` directory named `Header.js`. Note that the name of this file starts with a capital letter. In React, the names of custom components always start with a capital letter.
2. Inside this new file, import React and then define a function named `Header`:

```
import React from 'react';

function Header() {
  return (

  )
}
```

3. After the function definition, export this function so that it becomes a JavaScript module that can be imported into other JavaScript files:

```
export default Header;
```

4. Open `App.js` in your editor.
5. Cut the full `header` element (including the `<header>` and `</header>` tags) from `App.js` and paste it inside the `return` statement of the function of your new `Header` component.
6. Back in `App.js`, add an additional `import` statement to the beginning of the file to import the `Header` component:

```
import Header from './Header';
```

7. Notice that when we import the `Header` component, we need to use `./` before the file name and we don't include the `.js` at the end of the file name.

8. Inside the `return` statement, replace the existing `<header>` element with your new custom component, `<Header>`. To do this, just delete everything from the opening `<header>` tag to the closing `</header>` tag and type in a single self-closing `<Header />` tag.

```
<div className="App">
  <Header />
  <h1>Mathificent</h1>
  <footer> ... </footer>
</div>
```

9. Next we'll make the footer into a component. Start by making a new file named `Footer.js` in the `src` directory.

10. Import React, define a function named `Footer`, and export the function just as you did with the `Header` component.

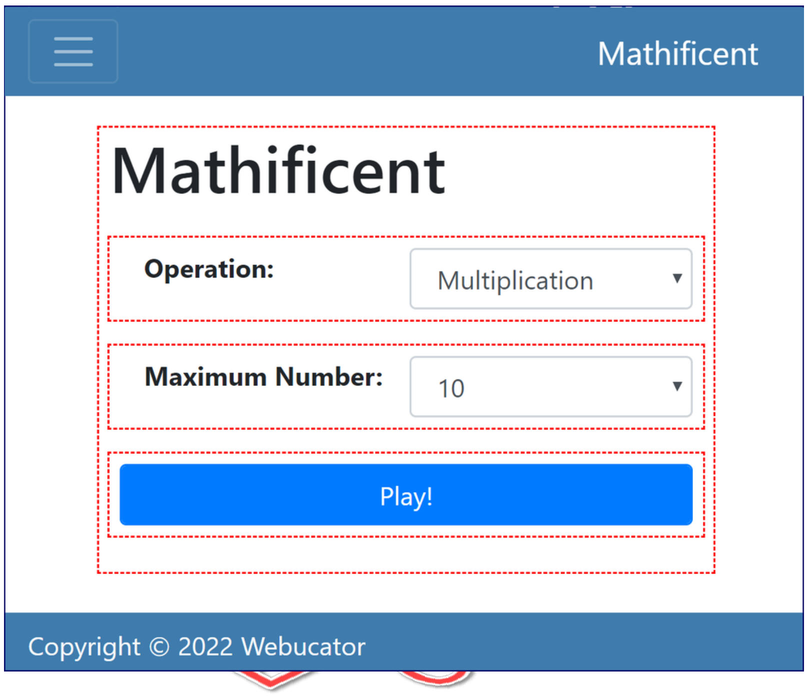
11. Cut the the full `footer` element (including the `<footer>` and `</footer>` tags) from `App.js` and paste it inside the `return` statement of the function of your new `Footer` component.

12. Import the `Footer` component into `App.js` and put the new custom `Footer` component at the end of the `return` statement, before the `</div>` tag. Your `return` statement in `App.js` should now look like this:

```
return (
  <div className="App">
    <Header />
    <h1>Mathificent</h1>
    <Footer />
  </div>
);
```

13. At this point, your web page should look the same as it did when you started the exercise, but the code is broken into components, making it easier to maintain.

14. Take a look at the finished Mathificent program and think about how you might split up the main content of the app into components. Here's one way it could be done:



15. The first step in the React development process is to create a static (meaning without any functionality) version of the app. Create a new functional component for each of the unique components in the following outline that you haven't already created:

- `App`
  - `Header`
  - *`Main`*
    - *`SelectInput`*
    - *`PlayButton`*
  - `Footer`

16. In the `return` statement for each component, put a placeholder element containing the name of the component for now. For example, here's what the `SelectInput` component should look like:

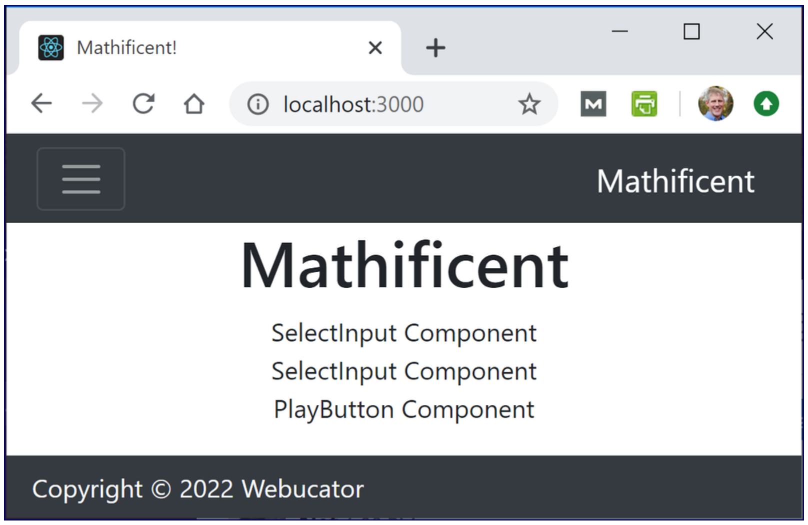
```
import React from 'react';

function SelectInput() {
  return(
    <div>SelectInput Component</div>
  )
}

export default SelectInput;
```

17. Now that you have all the components, it is time to put them together in the right order. Think about the hierarchy of components in your app:

    A. `App` contains `Header`, `Main`, and `Footer`.

    B. `Main` contains two instances of `SelectInput` and one `PlayButton`.

18. Import the correct components into `App.js` and `Main.js` and then modify the `return` statements of these two components to include the correct sub-components (also known as "child" components). Remember that only one base element can be returned in the `return` statement. The `Main` component should return a `main` element rather than a `div` element. When you're done, it should look like this in your browser:



Mathificent

Mathificent

SelectInput Component
SelectInput Component
PlayButton Component

Copyright © 2022 Webucator

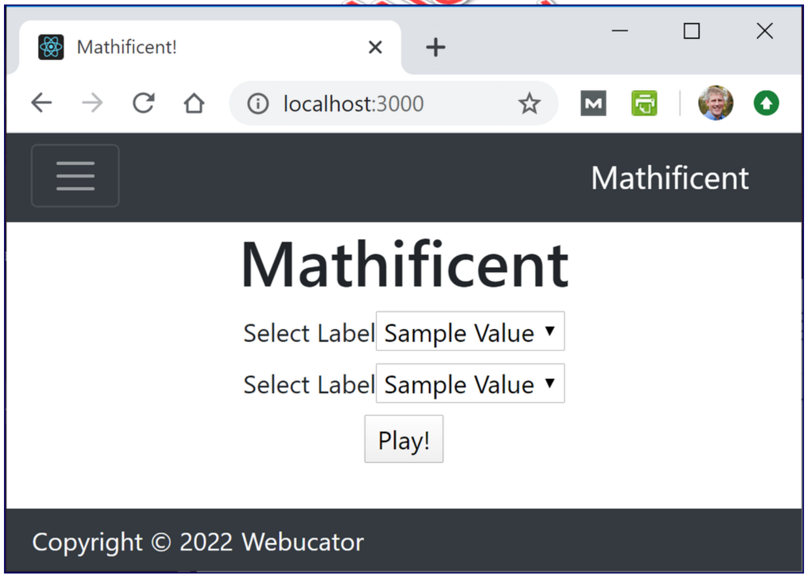
19. Replace the placeholder `div` in the `PlayButton` component with a `button` element:

```
<button>Play!</button>
```

20. Code the select dropdowns using static options and labels for now. We'll make them dynamic shortly:

```
<div>
  <label htmlFor="select">Select Label</label>
  <select id="select">
    <option value="sample value">Sample Value</option>
  </select>
</div>
```

21. Your application should now look like this:



Solution: React/Solutions/components/static/App.js

```
import React from 'react';
import './App.css';
import Header from './Header';
import Footer from './Footer';
import Main from './Main';

function App() {
  return (
  <div className="App">
    <Header />
    <h1>Mathificent</h1>
    <Main />
    <Footer />
  </div>
  );
}

export default App;
```

Solution: React/Solutions/components/static/Header.js

```
import React from 'react';

  function Header() {
  return (
  <header>
    <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
      <div className="container-fluid">
        <button className="navbar-toggler" type="button"
          data-bs-toggle="collapse" data-bs-target="#navbarText">
          <span className="navbar-toggler-icon"></span>
        </button>
        <div className="collapse navbar-collapse" id="navbarText">
          <ul className="navbar-nav mr-auto text-left">
            <li className="nav-item active">
              <a className="nav-link" href="/">Home</a>
            </li>
          </ul>
        </div>
        <a className="navbar-brand" href="/">Mathificent</a>
      </div>
    </nav>
  </header>
  )
}

export default Header;
```

Solution: React/Solutions/components/static/Footer.js

```
import React from 'react';

function Footer() {
  return (
      <footer className="navbar fixed-bottom bg-dark">
        <div className="container-fluid">
          <a href="https://www.webucator.com" className="nav-link text-light">
            Copyright &copy; {new Date().getFullYear()} Webucator
          </a>
        </div>
    </footer>
  )
}

export default Footer;
```

Solution: React/Solutions/components/static/Main.js

```
import React from 'react';
import SelectInput from './SelectInput';
import PlayButton from './PlayButton';

function Main() {
  return(
    <main>
      <SelectInput />
      <SelectInput />
      <PlayButton />
    </main>
  )
}

export default Main;
```

Solution: React/Solutions/components/static/SelectInput.js

```
import React from 'react';

function SelectInput() {
  return(
    <div>
      <label htmlFor="select">Select Label</label>
      <select id="select">
        <option value="sample value">Sample Value</option>
      </select>
    </div>
  )
}

export default SelectInput;
```

Solution: React/Solutions/components/static/PlayButton.js

```
import React from 'react';

function PlayButton() {
  return(
    <button>Play!</button>
  )
}

export default PlayButton;
```

Exercise 5: Passing Props Between Components

20 to 30 minutes

In this exercise, you will learn how to pass data from parent components to child components and then how to use that data in the child components.

1. Open `Main.js` in your editor.
2. Inside the function, but before the `return` statement, create an array to hold the possible operations.

```
const operations = ['+', '-', 'x', '/'];
```

3. Create an empty array inside the `Main` component that will hold the list of numbers that we'll use to populate the maximum number dropdown:

```
const numbers = [];
```

4. Use a `for` loop to create an array of the numbers from 2 to 100:

```
for (let number=2; number <= 100; number++) {
  numbers.push(number);
}
```

5. Modify your two `SelectInput` elements to pass the select dropdown values, labels, and ids to the child components:

```
<SelectInput label="Operation" id="operation"
  values={operations} />
<SelectInput label="Maximum Number" id="max-number"
  values={numbers} />
```

6. Open `SelectInput.js` in your editor.

7. Pass `props` into `SelectInput` as a parameter:

```
function SelectInput(props) {

}
```

8. Outside of the `return` statement in `SelectInput`, use the `Array.map()` method to generate a list of `option` elements. This same statement will be used by both of our dropdown boxes, as well as any future ones that we might add:

```
const values = props.values;
const selectOptions = values.map((value)=>
  <option value={value} key={value.toString()}>{value}</option>
);
```

9. Replace the hard-coded id and label with the prop values:

```
<label htmlFor={props.id}>{props.label}</label>
```

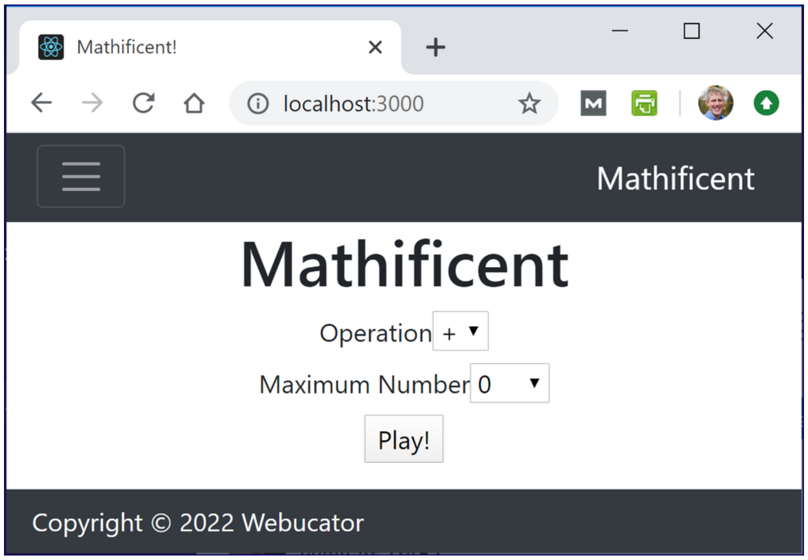
10. Change the `id` of the `select` to the `id` passed from the parent:

```
<select id={props.id}>
```

11. Replace the hard-coded `option` element with the array of `option` elements created by the `Array.map()` method:

```
<select id={props.id}>
  {selectOptions}
</select>
```

12. Run the app. It should look like this:



Solution: React/Solutions/components/static-with-props/Main.js

```
import React from 'react';
import SelectInput from './SelectInput';
import PlayButton from './PlayButton';

function Main() {
  const operations = ['+', '-', 'x', '/'];
  const numbers = [];
  for (let number = 2; number <= 100; number++) {
    numbers.push(number);
  }
  return(
    <main>
      <SelectInput label="Operation" id="operation"
        values={operations} />
      <SelectInput label="Maximum Number" id="max-number"
        values={numbers} />
      <PlayButton />
    </main>
  )
}

export default Main;
```

Solution: React/Solutions/components/static-with-props/SelectInput.js

```
import React from 'react';

function SelectInput(props) {
  const values = props.values;
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <div>
      <label htmlFor={props.id}>{props.label}</label>
        <select id={props.id}>
          {selectOptions}
        </select>
    </div>
  )
}

export default SelectInput;
```

Exercise 6: Organizing Your Components

15 to 25 minutes

As your app starts to get more complex, it can be helpful to organize it into subdirectories, rather than having everything in the same location as we have now.

In this exercise, you will learn how to create a more organized folder and file structure for your applications and put your components into subfolders.

1. Create a new directory inside `src` and name it `components`.
2. In your editor, drag the following files into the new `components` directory:
   A. `Footer.js`
   B. `Header.js`
   C. `Main.js`
   D. `PlayButton.js`
   E. `SelectInput.js`
3. Create another directory inside `src` and name it `containers`. Containers are components whose purpose is to hold other components, manage state, and pass data to presentational components.

4. In your editor, drag the following files into the new `containers` directory:
   - `App.js`
   - `App.css`
   - `App.test.js`
5. Open `App.js` in your editor and update the `import` statements to correctly import the subcomponents, like this:

```
import Header from '../components/Header';
import Footer from '../components/Footer';
import Main from '../components/Main';
```

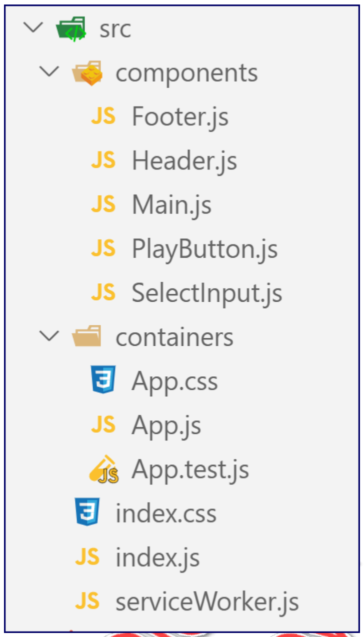
6. Open `index.js` in your editor and update the path to `App.js` in the `import`:

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

7. Delete the `logo.svg` file. You won't be using that anymore.
8. If it is not already running, start up your development server by running `npm start` in your terminal. It should look the same as it did before. If you get a "Module not found" error, then you need to check your `import` statements.

Solution

Your file structure should now look like this:


```
src
  components
    Footer.js
    Header.js
    Main.js
    PlayButton.js
    SelectInput.js
  containers
    App.css
    App.js
    App.test.js
  index.css
  index.js
  serviceWorker.js
```

3.2. Semantic HTML and the Fragment Element

Each React component can only return a single element. Sometimes that means you have to surround React elements with a container element to prevent an error. For example, the following `return` statement in a React component will produce an error because it's attempting to return more than one element:

```
return (
  <p>this is the first thing</p>
  <p>this is another thing</p>
);
```

One common way to fix this problem is by surrounding the required elements with a `div` element so that only one element, which may contain other elements, is returned:

```
return (
  <div>
    <p>this is the first thing</p>
    <p>this is another thing</p>
  </div>
);
```

Although this solution does make the React component work, it adds a layer of meaningless “non-semantic” markup to the document. React includes a `Fragment` element for creating valid React components without generating unnecessary layers of meaningless HTML.

`Fragment` creates a container around JSX elements without producing any HTML output. To use `Fragment` in a component, you first need to import it, like this:

```
import React, {Fragment} from 'react';
```

Once imported, you can use it in your `return` statement like you do any other element. For example:

```
return (
  <Fragment>
    <p>this is the first thing</p>
    <p>this is another thing</p>
  </Fragment>
)
```

The output of the this `return` statement will only include the two paragraphs.

React also includes a shorthand way of writing the `Fragment` element, which emphasizes that it doesn’t return anything and is also faster to type: `<></>`. Here’s the preceding example written using the shorthand `Fragment` element syntax:

```
return (
  <>
    <p>this is the first thing</p>
    <p>this is another thing</p>
  </>
);
```

One thing to be aware of with the shorthand `Fragment` syntax is that certain editors may incorrectly report it as an error.

In this course, we will write out the word "Fragment" as we find it clearer.

Exercise 7: Using Fragment

5 to 10 minutes

Examine all your components in the `components` folder. One of them is returning a `div` element that has no semantic value. Replace that `div` element with a `Fragment` element. Don't forget to import `Fragment`.

Solution: React/Solutions/components/fragment/SelectInput.js

```
import React, {Fragment} from 'react';

function SelectInput(props) {
  const values = props.values;
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <Fragment>
      <label htmlFor={props.id}>{props.label}</label>
      <select id={props.id}>
        {selectOptions}
      </select><br />
    </Fragment>
  )
}

export default SelectInput;
```

Code Explanation

Note that we did add a `<br />` tag after the `select` element, so that it continues to appear on its own line. We will remove that later in favor of CSS.

3.3. Destructuring props

Take another look at the solution to the Exercise 5:

Demo 3.4: React/Solutions/components/fragment/SelectInput.js

```
import React, {Fragment} from 'react';

function SelectInput(props) {
  const values = props.values;
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <Fragment>
      <label htmlFor={props.id}>{props.label}</label>
      <select id={props.id}>
        {selectOptions}
      </select><br />
    </Fragment>
  )
}

export default SelectInput;
```

Notice how we create a constant `values` and set it to `props.values`. We do that to make the next line a little easier to write. This is an especially common practice when you need to reuse a `props` property multiple times. Also, notice how we reference `props` properties several times in the file. It'd be nice to be able to refer to the property directly instead of having to type `props` over and over. We can do this by destructuring `props` in the function parameter, like this:

```
function SelectInput({label, id, values}) {...
```

See how that saves us from having to type `props` repeatedly:

Demo 3.5: React/Solutions/components/props-broken-out/SelectInput.js

```
import React, {Fragment} from 'react';

function SelectInput({label, id, values}) {
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <Fragment>
      <label htmlFor={id}>{label}</label>
      <select id={id}>
        {selectOptions}
      </select><br />
    </Fragment>
  )
}

export default SelectInput;
```

Conclusion

In this lesson, you have learned how to make more robust and flexible components through the rule of F.I.R.S.T. You also learned how to pass data between components using `props`, to organize your components, and to use the `Fragment` element.

LESSON 4
React State

Topics Covered

☑ "State" in React.

☑ Stateful variables.

☑ Updating state.

Introduction

The **state** of a React component is the internal data of that component that determines how it should change. In this lesson, you'll learn how to use state to add interactivity to your components.

4.1. Understanding State

State is the internal data of a component that keeps track of how it changes over time. At its most basic level, state is just an object within a component. To understand state, it's helpful to look at a few examples, starting first with a component that does not modify state:

Demo 4.1: React/demo-viewer/src/Demos/ChangeCount1.js

```
import React from 'react';

function ChangeCount1() {

  let count = 0;

  function increment(e) {
    count++;
    console.log(`Count is ${count}.`);
    // e.target.innerHTML = count;
  }

  return (
    <div className="container">
      <button className="btn btn-primary"
        onClick={increment}>{count}</button>
    </div>
  );
}

export default ChangeCount1;
```

Code Explanation

Things to notice:

1. The `count` variable is initialized to `0`:

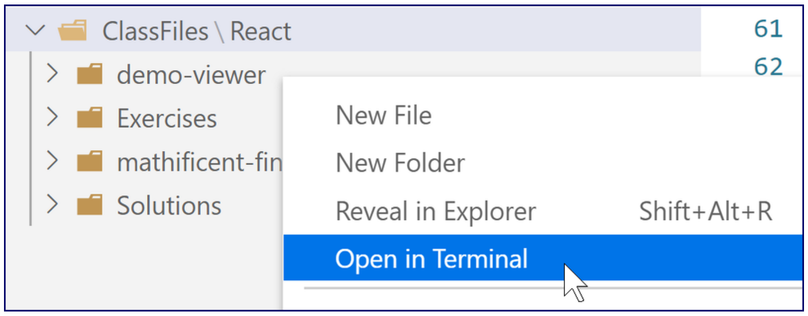
   ```
   let count = 0;
   ```

2. The button shows the value of count:

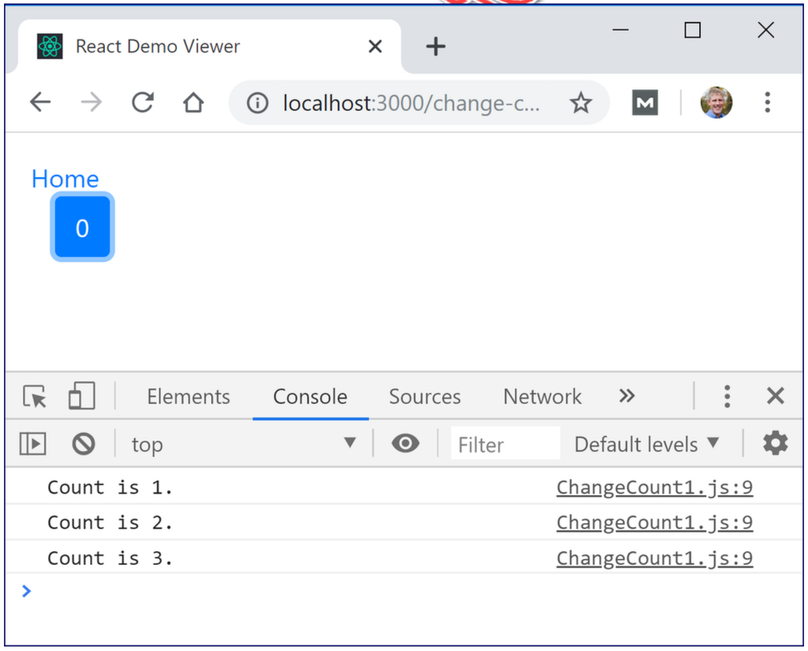
   ```
   <button className="btn btn-primary"
     onClick={increment}>{count}</button>
   ```

3. When the user clicks the button, the callback function `increment()` is called. Remember that, in JavaScript, an event's callback function is passed the event itself.
4. The `increment()` function currently does two things:
   A. Increments the value of `count` by `1`.
   B. Logs the value of `count` to the console.

5. To run this demo, open `React/demo-viewer` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



6. Run `npm start` to launch the demo-viewer React application.
7. Click the **ChangeCount1** link under **React State**. You should see a page with a button with the number `0` on it. Remember, this button is showing the value of `count`.
8. With Google Chrome's console open, click the button several times. You should see something like this:



```
Count is 1.                                ChangeCount1.js:9
Count is 2.                                ChangeCount1.js:9
Count is 3.                                ChangeCount1.js:9
```

You can see from the output in the console that the value of `count` is being incremented, but the value on the button does not change. In other words, the page is not **react**ing to the change in `count`.

9. Open `demo-viewer/src/Demos/ChangeCount1.js` in your editor and uncomment the following line:

   ```
   e.target.innerHTML = count;
   ```

   Here we explicitly update the DOM to reflect the change in the value of `count`. Return to the browser and click the button. The button now updates as the value of `count` changes.

4.2. Getting React to React

In the demo we just saw, we had to explicitly update the DOM to get it in sync with the value of the `count` variable. That's because React will only re-render when there is a change in state. Updating a local variable does not cause a change in state. Instead, we have to tell our component to create the variable *using* state. You do that like this:

```
const [count, setCount] = useState(0);
```

The `useState()` function will return two things:

1. An initial value.
2. A function for changing that value.

Using array destructuring, we assign these two values to constants. Although you can name those constants whatever you like, traditionally, the function name begins with `set` followed by the name of the the first constant. For example, if the first constant is named `foo`, the second constant would be named `setFoo`:

```
const [foo, setFoo] = useState('bar');
```

Let's see how it works:

Demo 4.2: React/demo-viewer/src/Demos/ChangeCount2.js

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

function ChangeCount2() {

  const [count, setCount] = useState(0);

  function increment() {
    setCount(count + 1);
    console.log(`Button clicked. Count is ${count}.`);
  }

  return (
    <div className="container">
      <button className="btn btn-primary"
        onClick={increment}>{count}</button>
    </div>
  );
}

export default ChangeCount2;
```

Code Explanation

Things to notice:

1. We have to import `useState` into our component. When you import a single function from a file containing multiple functions, you use curly braces around the name of the function. Here's what the new React import should look like:

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

2. We then create the `count` variable using:

   ```
   const [count, setCount] = useState(0);
   ```

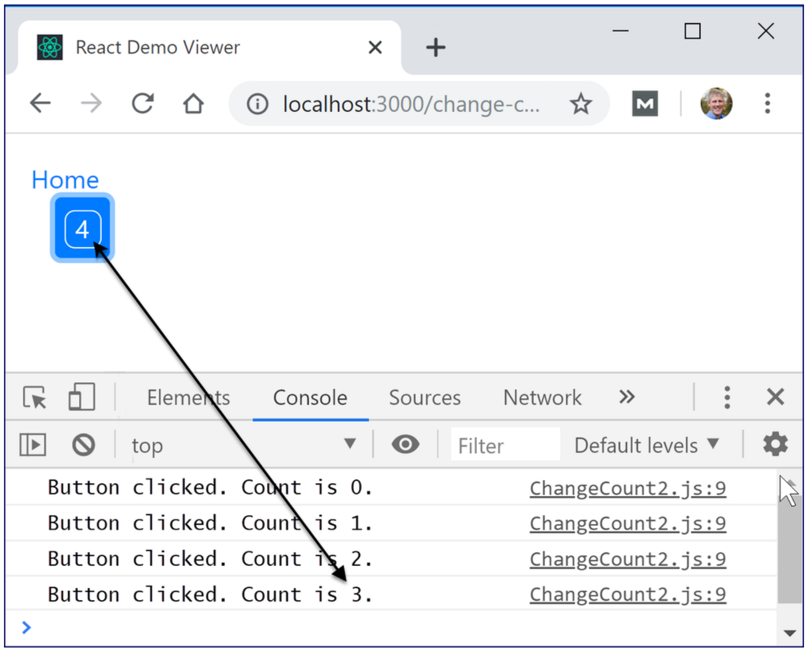
   And we update it using:

   ```
   setCount(count + 1);
   ```

3. Note that we do not explicitly change the `innerHTML` of the button.

4. In your browser, clear the console and then return to the demo-viewer home page and click the **ChangeCount2** link under **React State**. With Google Chrome's console open, click the button several times. The button value should be updating on each click. The page now *reacts* to the change in the value of `count`.

5. Notice the logged value of `count` vs. the value that shows up on the button. The logged value is always the value before the click:



```
Button clicked. Count is 0.          ChangeCount2.js:9
Button clicked. Count is 1.          ChangeCount2.js:9
Button clicked. Count is 2.          ChangeCount2.js:9
Button clicked. Count is 3.          ChangeCount2.js:9
```

To understand why that is, understand that your state variables live in their own space. When any state value changes, the component re-renders, but only after the current process is complete. So in this case, we have the following sequence of events:

A. The component is rendered and the value of `count` is set.

B. The user clicks the button resulting in a call to `increment()`.

C. The value of the state variable `count` is changed using `setCount()`.

D. The value of `count` at the time of the last rendering is logged to the console. That last rendering took place **before** the user clicked the button.

E. The component is re-rendered because of a change in state caused by the change in the value of the state variable `count`. This re-rendering causes the DOM to be updated based on any state changes.

The important takeaway here is that the callback function finishes executing before the re-rendering occurs.

4.3. Why is count a Constant?

It seems strange that `count` is a constant when its value is changing. Understanding why is helpful in understanding state. The value of the `count` is unchanged on each rendering of the component. Each time the component is rendered, the value of the constant `count` is set to the corresponding value of the state variable `count`. That state variable count is indeed *variable*. It can be changed using `setCount()`. Doing so will trigger a re-rendering of the component, meaning the component code is executed again, and a new constant `count` is created using the updated value of the state variable `count`.

To see this flow of events:

1. With the **ChangeCount2** example still open in your browser, clear the console.
2. Open `demo-viewer/src/Demos/ChangeCount2.js` in your editor and add the following line immediate after first setting count:

```
const [count, setCount] = useState(0);
console.log(`Rendering. Count is : ${count}.`);
```

3. Return to the browser. Notice that the following was logged to the console:

```
Rendering. Count is : 0.
```

4. Click the button twice. The following will be logged to the console:

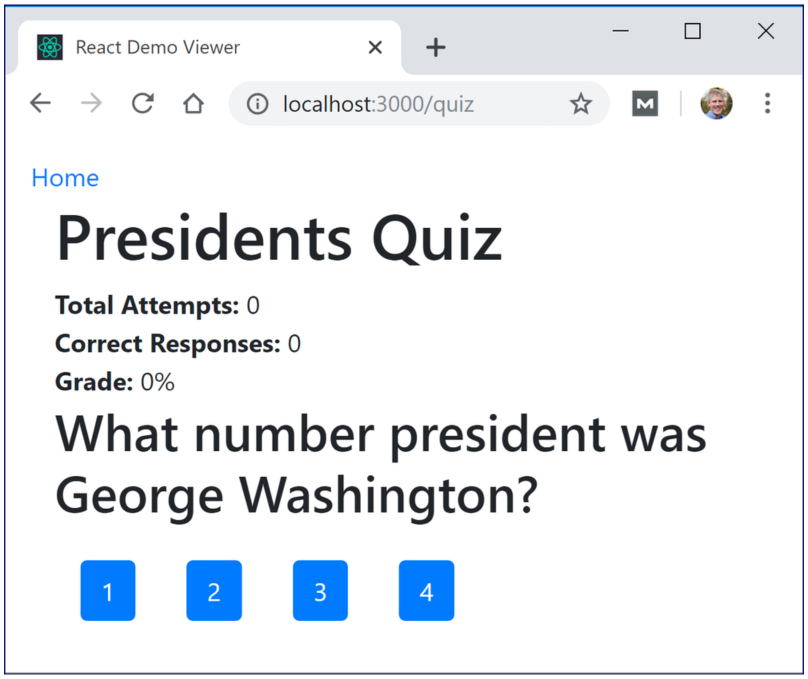
```
Button clicked. Count is 0.
Rendering. Count is : 1.
Button clicked. Count is 1.
Rendering. Count is : 2.
```

Notice that the output resulting from the button click occurs *before* the output from the re-rendering. The new value of the constant `count` is only set *after* the component is re-rendered.

4.4. Child Components and State

When a component's state is changed, that component is re-rendered, which causes all of its child components to also be re-rendered. This gives the child components the opportunity to update the DOM as well. Let's take a look at a more complete application. First, we'll look at the application in the browser:

Go back to the demo-viewer home page and click the **Quiz** link under under **React State**. You should see a page like this one:



Presidents Quiz

Total Attempts: 0
Correct Responses: 0
Grade: 0%

What number president was George Washington?

Play around a bit with the quiz and then, before reading on, spend some time exploring the code used to create this application. See if you can figure out how everything works.

We'll start by looking at the data:

Demo 4.3: React/demo-viewer/src/Demos/quiz/presidents-quiz.js

```
const quiz = {
  "title": "Presidents Quiz",
  "questions" : [
    {
      "question": "What number president was George Washington?",
      "answers": [1, 2, 3, 4],
      "correct": 1
    },
    {
      "question": "What number president was Thomas Jefferson?",
      "answers": [2, 3, 4, 5],
      "correct": 3
    },
    {
      "question": "What number president was Abraham Lincoln?",
      "answers": [14, 15, 16, 17],
      "correct": 16
    },
    {
      "question": "What number president was John F. Kennedy?",
      "answers": [33, 34, 35, 36],
      "correct": 35
    },
-------Lines 24 through 28 Omitted-------
  ]
};

export default quiz;
```

Code Explanation

This is simply a JavaScript file that exports a JavaScript object with two properties:

1. `title` - The title of the quiz.
2. `questions` - A list of `question` objects, each of which contains:
    - A. `question` - A question.
    - B. `answers` - A list of possible answers.
    - C. `correct` - The correct answer.

Now let's look at the main application file:

Demo 4.4: React/demo-viewer/src/Demos/quiz/Quiz.js

```
import React, {useState} from 'react';
import Question from './Question';
import Answer from './Answer';
import Message from './Message';
import Scoreboard from './Scoreboard';
import quiz from './presidents-quiz';

function Quiz() {

  const [qNum, setQNum] = useState(0);
  const [totalAttempts, setTotalAttempts] = useState(0);
  const [correctResponses, setCorrectResponses] = useState(0);
  const [message, setMessage] = useState('');

  const quizTitle = quiz.title;

  // All of these will be updated when qNum changes
  const question = quiz.questions[qNum];
  const questionText = question['question'];
  const correctAnswer = question['correct'];
  const answers = question['answers'].map((answer) =>
    <Answer answer={answer} key={answer} checkAnswer={checkAnswer} />
  );

  function checkAnswer(answer) {
    setTotalAttempts(totalAttempts + 1);
    if (answer === correctAnswer) {
      setCorrectResponses(correctResponses + 1);
      nextQuestion();
    } else {
      setMessage('Wrong!');
    }
  }

  function nextQuestion() {
    const nextQNum = qNum + 1;
    setMessage('');
    if (nextQNum === quiz.questions.length) {
      setQNum(0); // Go back to first question
    } else {
      setQNum(nextQNum);
    }
  }

  return (
    <div className="container">
      <h1>{quizTitle}</h1>
      <Scoreboard totalAttempts={totalAttempts}
        correctResponses={correctResponses} />
      <Question question={questionText} />
      {answers}
      <Message msg={message} />
    </div>
  );
}

export default Quiz;
```

Code Explanation

1. This file imports the quiz data and four child components:
   A. `Question`
   B. `Answer`
   C. `Message`
   D. `Scoreboard`

2. At the start of the `Quiz()` function, we set state variables and the functions for changing them:
   A. `qNum` - the current question number.
   B. `totalAttempts` - the number of attempts.
   C. `correctResponses` - the number of correct responses.
   D. `message` - the message to output when the user attempts to answer a question.
3. We then set `quizTitle` to the `title` property of the imported `quiz` object.
4. Next, we set `question` based on the current value of `qNum`. That will give us an object like this:

```
{
  "question": "What number president was George Washingon?",
  "answers": [1, 2, 3, 4],
  "correct": 1
}
```

And we use the properties of that object to set `questionText`, `correctAnswer`, and `answers`, which will be an array of `<Answer>` tags to include in our `return` statement. Let's take a look at the `Answer` component:

Demo 4.5: React/demo-viewer/src/Demos/quiz/Answer.js

```
import React from 'react';

function Answer(props) {

  return (
    <button className="btn btn-primary m-3"
      onClick={() => {
        props.checkAnswer(props.answer);
      }}>{props.answer}</button>
  );
}

export default Answer;
```

Code Explanation

This returns a button that shows the value of the passed-in `answer`. Notice that the `onClick` handler's callback function is:

```
() => {
  props.checkAnswer(props.answer);
}
```

`props.checkAnswer` holds the function passed in via the `checkAnswer` attribute in the component tag:

```
<Answer answer={answer} key={answer} checkAnswer={checkAnswer} />
```

The `checkAnswer()` function looks like this:

```
function checkAnswer(answer) {
  setTotalAttempts(totalAttempts + 1);
  if (answer === correctAnswer) {
    setCorrectResponses(correctResponses + 1);
    nextQuestion();
  } else {
    setMessage('Wrong!');
  }
}
```

Whether or not the answer is correct, it will increment `totalAttempts` by one, which will result in a re-rendering of the child `Scoreboard` component. It then checks the user's answer and responds accordingly:

1. If the answer is correct, it will increment `correctResponses` by one and call `nextQuestion()`, which will clear the message and update `qNum`, resulting in a re-rendering of the child `Question` component, all `Answer` components, and the `Message` component.
2. If the answer is incorrect, it will set the message to "Wrong!", resulting in a re-rendering of the child `Message` component.

Review the `Question`, `Message`, and `Scoreboard` components as well. You should be able to understand how they work:

Demo 4.6: React/demo-viewer/src/Demos/quiz/Question.js

```
import React from 'react';

function Question(props) {

  return (
    <h2>{props.question}</h2>
  );
}

export default Question;
```

Demo 4.7: React/demo-viewer/src/Demos/quiz/Message.js

```
import React from 'react';

function Message(props) {

  return (
    <strong className="text-danger">{props.msg}</strong>
  );
}

export default Message;
```

Demo 4.8: React/demo-viewer/src/Demos/quiz/Scoreboard.js

```
import React from 'react';

function Scoreboard(props) {

  function getGrade() {
    if (props.totalAttempts === 0) {
      return 0;
    }
    const grade = (props.correctResponses / props.totalAttempts) * 100;
    return Math.round(grade);
  }

  return (
    <div>
      <strong>Total Attempts: </strong>
      {props.totalAttempts} <br />

      <strong>Correct Responses: </strong>
      {props.correctResponses} <br />

      <strong>Grade: </strong>
      {getGrade()}%
    </div>
  );
}

export default Scoreboard;
```

Now that you have an understanding of how state works to keep React apps reactive, let's use state in our Mathificent game.

Exercise 8: Adding State

15 to 25 minutes

In this exercise we'll add state to the Mathificent application. On the initial screen, the user selects the operation and the maximum number using dropdowns. The application needs to keep track of those selections to generate the questions used in the game. We'll use state variables to set and update those values.

1. Inside `App.js`, modify the first `import` statement to also import `useState` from 'react':

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

2. Inside the function definition for the `App` component, create a new state variable called `operation` and a `setOperation` function for modifying that variable. Set the default value to `'x'`:

```
const [operation, setOperation] = useState('x');
```

3. Next, use the same technique to create a state variable named `maxNumber` and a function called `setMaxNumber`. Set the default value to the number `10`:

```
const [maxNumber, setMaxNumber] = useState(10);
```

4. Inside the `return` statement in `App.js`, pass all of the state variables and functions to the `Main` component:

```
<Main operation={operation}
  setOperation={setOperation}
  maxNumber={maxNumber}
  setMaxNumber={setMaxNumber} />
```

5. Create, import, and display a new `Game` component:

A. In the `components` directory, create a new `Game` component that for now just outputs the `operation` and `maxNumber` state variables, which we will pass in via `props`:

```
import React from 'react';

function Game(props) {
  return (
    <div>
      Operation is <strong>{props.operation}</strong>.
      Maximum number is <strong>{props.maxNumber}</strong>.
    </div>
  )
}

export default Game;
```

B. Back in `App.js`, import the new `Game` component:

```
import Game from './components/Game';
```

And below the `Main` element, add a `Game` element, passing in `operation` and `maxNumber`:

```
<Game operation={operation}
  maxNumber={maxNumber} />
```

C. Note that eventually, the `App` component will use routing to either display the `Main` component or the `Game` component, but as we haven't learned how to do routing yet, it is currently displaying both together.

6. Update the `Main` function component to receive `props` as a parameter.

```
function Main(props) {
```

7. Pass the correct `props` variables and functions to the `SelectInput` components:

```
<SelectInput label="Operation"
  id="operation"
  currentValue={props.operation}
  setCurrentValue={props.setOperation}
  values={operations} />

<SelectInput label="Maximum Number"
  id="max-number"
  currentValue={props.maxNumber}
  setCurrentValue={props.setMaxNumber}
  values={numbers} />
```

8. Inside the `SelectInput` component, add `currentValue` and `setCurrentValue` to the deconstructed props parameter:

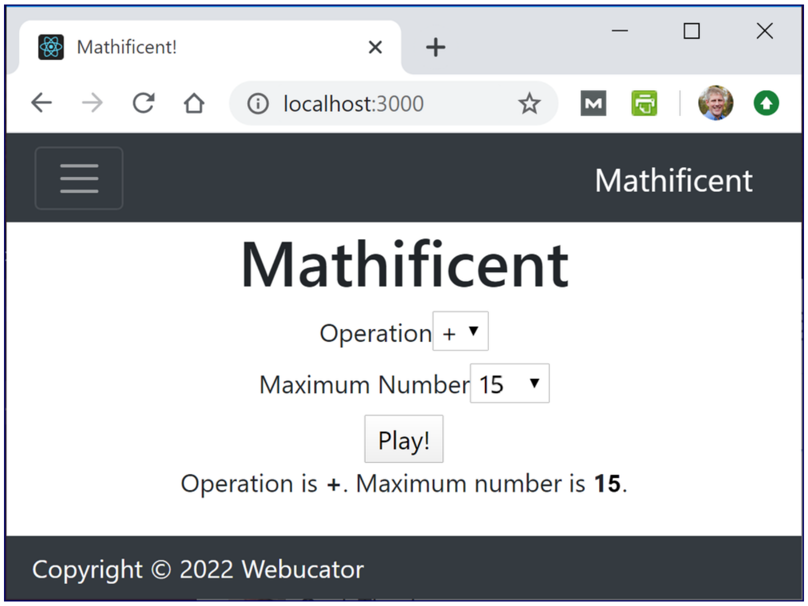
```
function SelectInput({label, id, values,
  currentValue, setCurrentValue}) {…
```

Then set `defaultValue` of the `select` control and use an `onChange` handler to update the state when a the user selects a different dropdown item.

```
<select id={id}
  defaultValue={currentValue}
  onChange={(e) => setCurrentValue(e.target.value)}>
  {selectOptions}
</select>
```

The `defaultValue` attribute of form controls is a special React attribute that is used to set the value when the form control loads.

9. If it is not already running, start up your development server by running `npm start` in your terminal. Change the operation to "+" and the maximum number to `15`. You should see the following:



Mathificent

Operation +

Maximum Number 15

Play!

Operation is **+**. Maximum number is **15**.

Copyright © 2022 Webucator

Solution: React/Solutions/state/App.js

```
import React, {useState} from 'react';
import './App.css';

import Header from '../components/Header';
import Footer from '../components/Footer';
import Main from '../components/Main';
import Game from '../components/Game';

function App() {
  const [operation, setOperation] = useState('x');
  const [maxNumber, setMaxNumber] = useState(10);
  return (
  <div className="App">
    <Header />
    <h1>Mathificent</h1>
    <Main operation={operation}
          setOperation={setOperation}
          maxNumber={maxNumber}
          setMaxNumber={setMaxNumber} />
    <Game operation={operation}
          maxNumber={maxNumber} />
    <Footer />
  </div>
  );
}

export default App;
```

Solution: React/Solutions/state/Game.js

```
import React from 'react';

function Game(props) {
  return (
    <div>
      Operation is <strong>{props.operation}</strong>.
      Maximum number is <strong>{props.maxNumber}</strong>.
    </div>
  )
}

export default Game;
```

Solution: React/Solutions/state/Main.js

```
import React from 'react';
import SelectInput from './SelectInput';
import PlayButton from './PlayButton';

function Main(props) {
  const operations = ['+', '-', 'x', '/'];
  const numbers = [];
  for (let number = 2; number <= 100; number++) {
    numbers.push(number);
  }
  return(
    <main>
      <SelectInput label="Operation"
        id="operation"
        currentValue={props.operation}
        setCurrentValue={props.setOperation}
        values={operations} />
      <SelectInput label="Maximum Number"
        id="max-number"
        currentValue={props.maxNumber}
        setCurrentValue={props.setMaxNumber}
        values={numbers} />
      <PlayButton />
    </main>
  )
}

export default Main;
```

Solution: React/Solutions/state/SelectInput.js

```
import React, {Fragment} from 'react';

function SelectInput({label, id, values,
                      currentValue, setCurrentValue}) {
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <Fragment>
      <label htmlFor={id}>{label}</label>
      <select id={id}
        defaultValue={currentValue}
        onChange={(e) => setCurrentValue(e.target.value)}>
        {selectOptions}
      </select><br />
    </Fragment>
  )
}

export default SelectInput;
```

Conclusion

In this lesson, you have learned how to manage state using `useState` and to pass state variables between components.

LESSON 5
React Routing

Topics Covered

☑ Routing in a React application.

Introduction

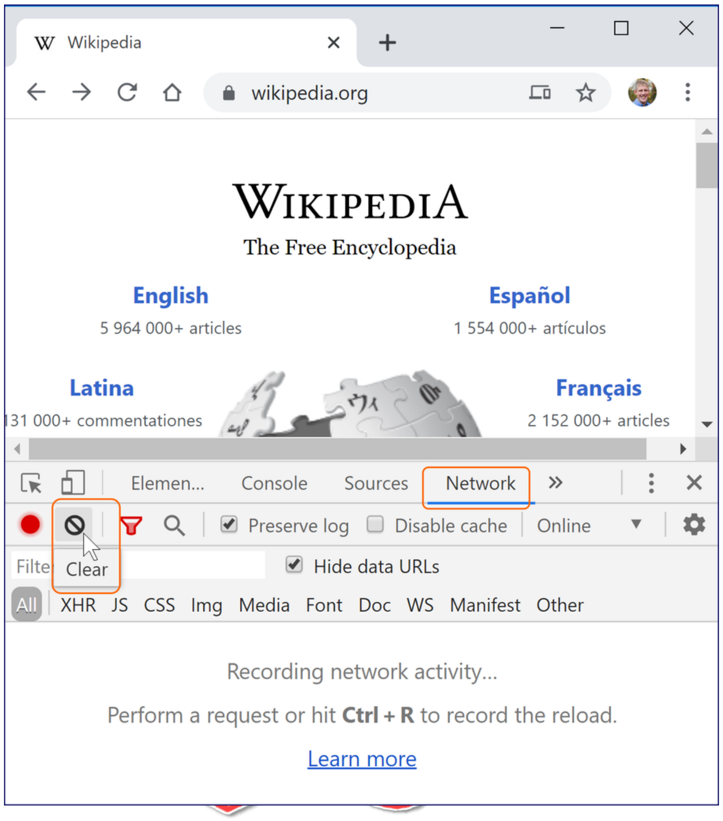
Routing refers to the ability to change what displays in the browser based on the current value of the browser `location` property.

5.1. Routing

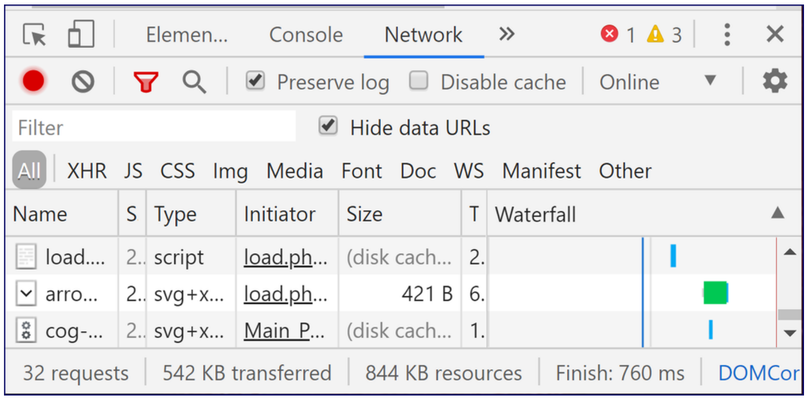
The `location` property of the browser is how the browser tracks the current web page being viewed. However, with JavaScript "single page" applications, the actual web page downloaded to the browser is always the same (`index.html` in our case), so we can use the `location` property to determine which components are mounted at any one time, without needing to download another file from the server.

First, let's take a look at how standard web pages work:

1. Visit `https://www.wikipedia.org/` in Google Chrome.
2. Open Google Chrome's **Network** tab and then clear it by pressing the Clear icon:



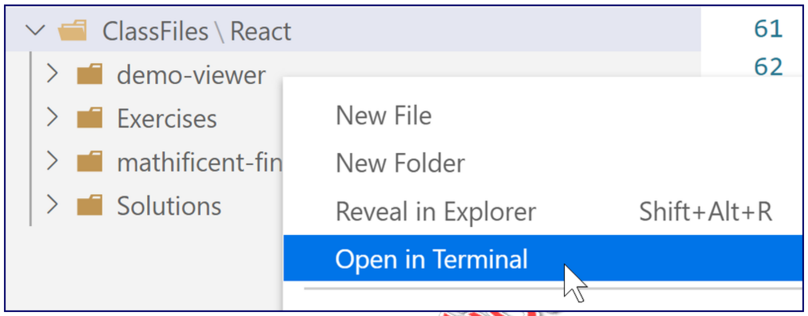
3. With the **Network** tab open, click any link on the page. You will see the Network tab fill up with downloaded assets:



Name	S	Type	Initiator	Size	T	Waterfall
load....	2..	script	load.ph...	(disk cach...	2.	
arro...	2..	svg+x...	load.ph...	421 B	6.	
cog-...	2..	svg+x...	Main P...	(disk cach...	1.	

32 requests | 542 KB transferred | 844 KB resources | Finish: 760 ms | DOMCor

4. Now open `React/demo-viewer` in VS Code's terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



5. Run `npm start` to launch the demo-viewer React application.
6. Open and clear Google Chrome's **Network** tab.
7. Click around. Notice that the URL changes and the page display changes, but nothing gets downloaded. This is because each link click does not result in a fresh fetch of a web page from the web server. Instead, it results in a new router to a different component, which then updates the DOM.

In the next exercise, we will see how this is done.

Exercise 9: Implementing Routes

15 to 25 minutes

In this exercise, we'll add routing to the Mathificent app.

1. Open `Exercises/mathficent` in the terminal.
2. Install **React Router** by running the following command:

```
npm install react-router-dom@latest
```

3. Use the following `import` statement to import the `Routes` and `Route` components from React Router into `App.js`:

```

import {Routes,Route} from 'react-router-dom';
```

4. Open `index.js` and import `BrowserRouter` into it:

```

import {BrowserRouter} from 'react-router-dom';
```

5. Inside the `root.render` method in `index.js`, wrap the `App` component with `BrowserRouter` tags:

```
root.render(<BrowserRouter><App /></BrowserRouter>);
```

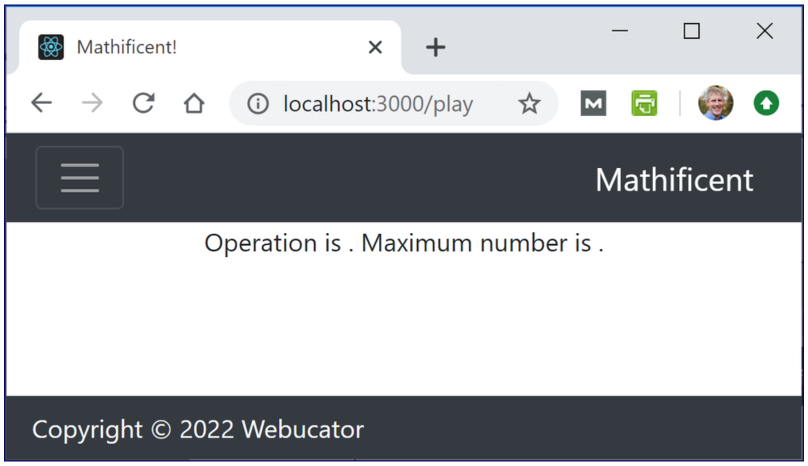
6. In `App.js`, we currently display both `Main` and `Game`:

```
<Main operation={operation} setOperation={setOperation} maxNumber={maxNumber}
setMaxNumber={setMaxNumber} />
<Game operation={operation} maxNumber={maxNumber} />
```

Replace that code with a `Routes` component containing two `Route` components:

```
<Routes> <Route exact path="/" element={<Main />} /> <Route path="/play" ele ↵
ment={<Game>}
/> </Routes>
```

7. Start up your development server in `Exercises/mathificent` if it is not running already. You should see the same view as before, with the Mathificent header and the two dropdowns, but without the Game component.
8. Manually change the URL in the browser address bar to `http://localhost:3000/play`. The `Game` component will render where the `Main` component was, as shown here:

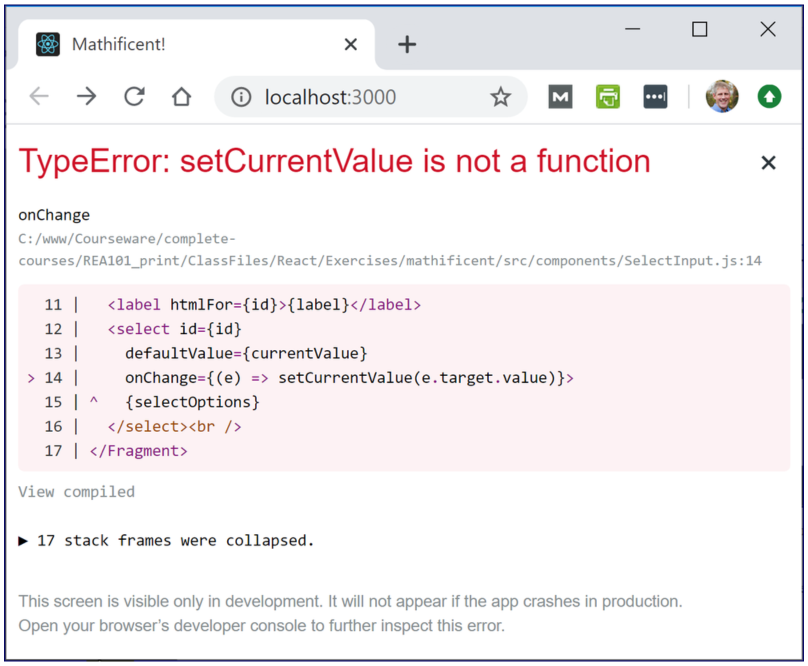


Operation is . Maximum number is .

Copyright © 2022 Webucator

Notice that the values for `props.operation` and `props.maxNumber` do not show up. We'll fix that soon.

9. Use your browser's **Back** button to return to the home page route.
10. Try to change the selected option in one of the dropdown menus. You'll get an error as shown here:


Mathificent!

localhost:3000

TypeError: setCurrentValue is not a function

```
onChange
C:/www/Courseware/complete-
courses/REA101_print/ClassFiles/React/Exercises/mathificent/src/components/SelectInput.js:14
  11 |     <label htmlFor={id}>{label}</label>
  12 |     <select id={id}
  13 |       defaultValue={currentValue}
> 14 |       onChange={(e) => setCurrentValue(e.target.value)}>
  15 |  ^    {selectOptions}
  16 |     </select><br />
  17 |   </Fragment>
View compiled

▶ 17 stack frames were collapsed.
```

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.

The reason for this error is that we're no longer passing the state data into the `Main` component. Let's fix these issues.

11. Add the props for the main element to the `element` attribute in the `Main` route:

```
<Route exact path="/" element={<Main operation={operation}
        setOperation={setOperation} maxNumber={maxNumber} setMaxNumber={setMaxNumber} />}
 />
```

The `exact` keyword ensures that the component will not show up on subpaths (e.g., `http://localhost:3000/foo`).

12. Return to your browser and you should be able to change the currently selected option and see the changes working correctly.

13. The next thing we need to do is to make the **Play** button change the route to the game. Open `PlayButton.js` and import `Link` from `react-router-dom`:

```
import {Link} from 'react-router-dom';
```

14. Change the `Button` element to a `Link` element and add a `to` attribute. The `to` attribute indicates what the `location` property should be set to when the `Link` element is clicked. So,

set the value of the `to` attribute to `/play`. We'll also add some Bootstrap classes to make it look like a button:

```
<Link className="btn btn-primary" to="/play">Play!</Link>
```

15. In your browser, click the button. The currently-displayed component should change from `Main` to `Game`.
16. Update the route for `Game` in the `App` component so that it passes `operation` and `maxNumber` as `props`:

```
<Route exact path="/play" element={<Game operation={operation}
          maxNumber={maxNumber} />} />
```

17. Finally, move the `h1` element from `App.js` to `Main.js`. Place it right below the open `<main>` tag:

```
<main> <h1>Mathificent</h1>…
```

Congratulations! You now have routing set up, and we're ready to implement the actual logic of the game.

Solution: React/Solutions/routing/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import {BrowserRouter} from 'react-router-dom';
import './index.css';
import App from './containers/App';
import * as serviceWorker from './serviceWorker';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<BrowserRouter><App /></BrowserRouter>);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Solution: React/Solutions/routing/App.js

```
import React, {useState} from 'react';
import {Routes,Route} from 'react-router-dom';
import './App.css';
import Header from '../components/Header';
import Footer from '../components/Footer';
import Main from '../components/Main';
import Game from '../components/Game';

function App() {
  const [operation, setOperation] = useState('+');
  const [maxNumber, setMaxNumber] = useState(10);
  return (
  <div className="App">
    <Header />
    <Routes>
      <Route exact path="/"
      element={
          <Main operation={operation}
          setOperation={setOperation}
          maxNumber={maxNumber}
          setMaxNumber={setMaxNumber} />} />
    <Route path="/play"
      element={
          <Game operation={operation}
          maxNumber={maxNumber} />} />
    </Routes>
    <Footer />
  </div>
  );
}

export default App;
```

Solution: React/Solutions/routing/Main.js

```
-------Lines 1 through 10 Omitted-------
  return(
    <main>
      <h1>Mathificent</h1>
      <SelectInput label="Operation"
        id="operation"
        currentValue={props.operation}
        setCurrentValue={props.setOperation}
        values={operations} />
      <SelectInput label="Maximum Number"
        id="max-number"
        currentValue={props.maxNumber}
        setCurrentValue={props.setMaxNumber}
        values={numbers} />
      <PlayButton />
    </main>
  )
-------Lines 27 through 29 Omitted-------
```

Solution: React/Solutions/routing/PlayButton.js

```
import React from 'react';
import {Link} from 'react-router-dom';

function PlayButton() {
  return(
    <Link className="btn btn-primary" to="/play">Play!</Link>
  )
}

export default PlayButton;
```

Conclusion

In this lesson, you have learned about routing and how React router makes it possible to change the component being displayed based on the browser's `location` property.

LESSON 6
Styling React Apps

Topics Covered

☑ Styling React components and applications.

Introduction

Our current application isn't all that pretty. Let's start working on that.

6.1. Plain-old CSS

The first thing to know is that, at the end of the day, we're styling HTML with CSS, just like you would with any other website. Early on in the course, we added Bootstrap's CSS to our `public/index.html` file, and we used some Bootstrap classes to style our `header` and `footer`:

Demo 6.1: Header.js

```
import React from 'react';

function Header() {
  return (
    <header>
      <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
          <div className="container-fluid">
            <button className="navbar-toggler" type="button"
            data-bs-toggle="collapse" data-bs-target="#navbarText">
              <span className="navbar-toggler-icon"></span>
          </button>
          <div className="collapse navbar-collapse" id="navbarText">
              <ul className="navbar-nav mr-auto text-left">
                  <li className="nav-item active">
                      <a className="nav-link" href="/">Home</a>
                  </li>
              </ul>
          </div>
  <a className="navbar-brand" href="/">Mathificent</a>
          </div>
      </nav>
    </header>
  )
}

export default Header;
```

Demo 6.2: Footer.js

```
import React from 'react';

function Footer() {
  return (
    <footer className="navbar fixed-bottom bg-dark">
      <div className="container-fluid">
        <a href="https://www.webucator.com" className="nav-link text-light">
          Copyright &copy; {new Date().getFullYear()} Webucator
        </a>
      </div>
    </footer>
  )
}

export default Footer;
```

className Replaces class

Remember that we must use the attribute `className` in place of `class`, because `class` is a reserved keyword in JavaScript.

You could replace the Bootstrap CSS with any CSS framework or file(s) that you like. Likewise, you could add additional CSS files to the `index.html` page. These styles will be available to all components in your React application.

6.1. Importing CSS Modules to Components

When we first created our React app with **Create React App**, an `App.css` file was included and the original `App.js` file imported it:

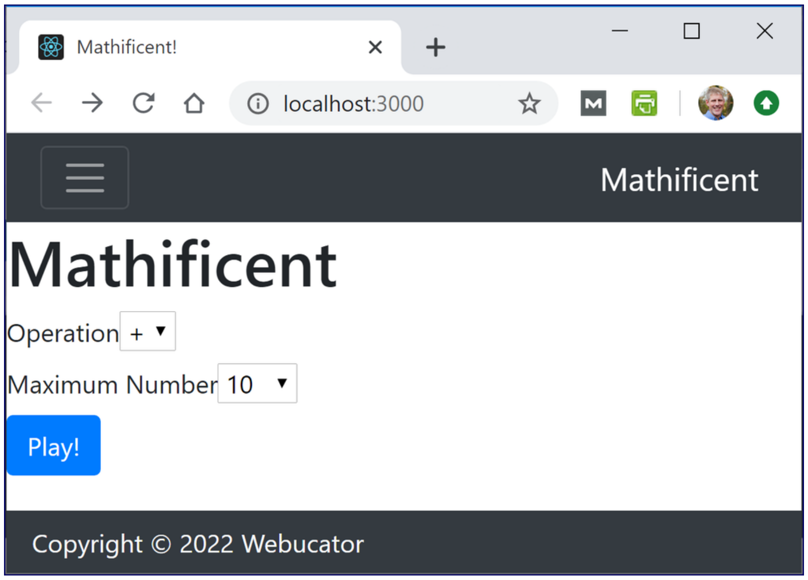
```
import './App.css';
```

We have left that import in place, which is fine, but we should clean up the styles as they no longer relate to our application. Let's do that now.

Exercise 10: Cleaning Up App.css

10 to 15 minutes

1. Start up your development server in `Exercises/mathificent` if it is not running already.
2. With the web app open in your browser, open `containers/App.css` in your editor.
3. Delete everything in the file and save. Your app should automatically update to look like this:



4. Now, add the following rule:

```
footer, header {
  background-color: #3f7cad;
}
```

The background color of the `header` and `footer` will not change, because this rule is being overridden by a Bootstrap class. Let's fix that.

5. Open the `Header` and `Footer` components and delete the `bg-dark` class from the `<nav>` in the header and the `<footer>` tag, so they look like this:

```
<nav className="navbar navbar-expand-lg navbar-dark">
```

```
<footer className="navbar fixed-bottom">
```

The background color should now be a steel blue.

6. We chose to style the `header` and `footer` in `App.css` because we want them to look the same throughout the application.

Solution: React/Solutions/styling/App.css

```
footer, header {
  background-color: #3f7cad;
}
```

Solution: React/Solutions/styling/Header.js

```
-------Lines 1 through 5 Omitted-------
      <nav className="navbar navbar-expand-lg navbar-dark">
-------Lines 7 through 26 Omitted-------
```

Solution: React/Solutions/styling/Footer.js

```
-------Lines 1 through 4 Omitted-------
    <footer className="navbar fixed-bottom">
-------Lines 6 through 15 Omitted-------
```

Exercise 11: Styling the Main Component

10 to 15 minutes

When we deleted the old styles from `App.css`, the main content of the page became left aligned. Let's move it back to the center and add some additional styling.

1. Start up your development server in `Exercises/mathificent` if it is not running already.
2. Create a new `Main.css` file in the `components` directory and add the following rule to it:

```
main {
  width: 380px;
  margin: auto;
}
```

3. Open `Main.js` in your editor and import the new `Main.css` file:

```
import './Main.css';
```

4. Now, still in `Main.js`, place each component within the `main` element in its own row using `div` elements with the "row", "mx-1", and "my-3" classes from Bootstrap:

```
<div className="row mx-1 my-3">
  <SelectInput… />
</div>
<div className="row mx-1 my-3">
  <SelectInput… />
</div>
<div className="row mx-1 my-3">
  <PlayButton… />
</div>
```

The "row" classes breaks the lines into rows. The "mx-1" and "my-3" classes add horizontal and vertical margin, respectively. The page will look a little better now.

5. Now, let's add some Bootstrap classes to the `SelectInput` and `PlayButton` components:

   A. Open `SelectInput.js` in your editor. Remember that this component is being placed inside a Bootstrap "row".

   B. Add the "col" and "fw-bold" classes to the `label` element.

   C. Add the "col" and "form-control" classes to the `select` element.

Those elements should now look like this:

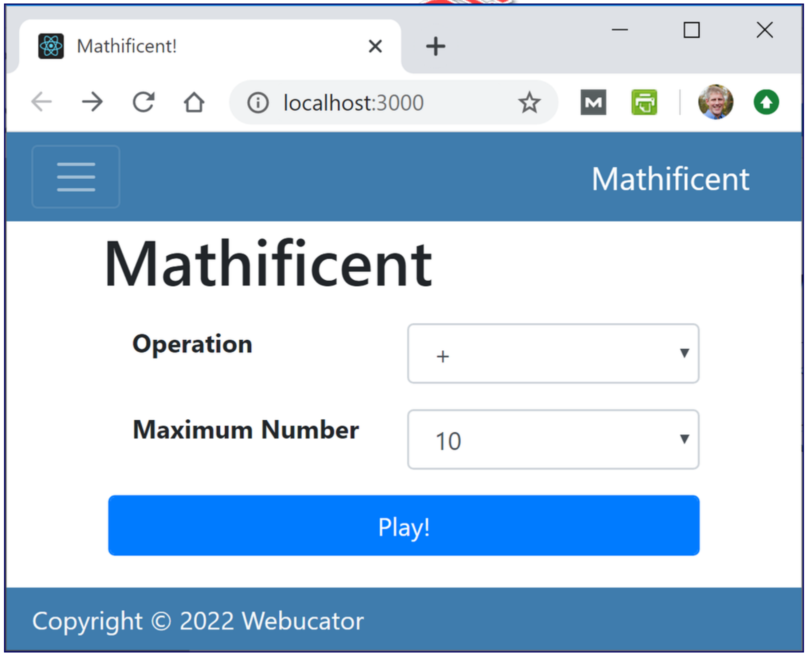
```
<label htmlFor={id} className="col fw-bold">{label}</label>
<select id={id}
  defaultValue={currentValue}
  onChange={(e) => setCurrentValue(e.target.value)}
  className="col form-control">
    {selectOptions}
</select>
```

A. Open `PlayButton.js` in your editor.

B. Add the "form-control" class to the `Link` element:

```
<Link className="btn btn-primary form-control" to="/play">Play!</Link>
```

The page should now appear like this:



Solution: React/Solutions/styling/Main.js

```
import React from 'react';
import SelectInput from './SelectInput';
import PlayButton from './PlayButton';
import './Main.css';

function Main(props) {
  const operations = ['+', '-', 'x', '/'];
  const numbers = [];
  for (let number = 2; number <= 100; number++) {
    numbers.push(number);
  }
  return(
    <main>
      <h1>Mathificent</h1>
      <div className="row mx-1 my-3">
        <SelectInput label="Operation"
          id="operation"
          currentValue={props.operation}
          setCurrentValue={props.setOperation}
          values={operations} />
      </div>
      <div className="row mx-1 my-3">
        <SelectInput label="Maximum Number"
          id="max-number"
          currentValue={props.maxNumber}
          setCurrentValue={props.setMaxNumber}
          values={numbers} />
      </div>
      <div className="row mx-1 my-3">
        <PlayButton />
      </div>
    </main>
  )
}

export default Main;
```

Solution: React/Solutions/styling/Main.css

```
main {
  width: 380px;
  margin: auto;
}
```

Solution: React/Solutions/styling/SelectInput.js

```
import React, {Fragment} from 'react';

function SelectInput({label, id, values,
                      currentValue, setCurrentValue}) {
  const selectOptions = values.map((value)=>
    <option value={value} key={value.toString()}>{value}</option>
  );

  return(
    <Fragment>
      <label htmlFor={id} className="col fw-bold">{label}</label>
      <select id={id}
        defaultValue = {currentValue}
        onChange = {(e) => setCurrentValue(e.target.value)}
        className="col form-control">
        {selectOptions}
      </select>
    </Fragment>
  )
}

export default SelectInput;
```

Solution: React/Solutions/styling/PlayButton.js

```
import React from 'react';
import {Link} from 'react-router-dom';

function PlayButton() {
  return(
    <Link className="btn btn-primary form-control" to="/play">
      Play!
    </Link>
  )
}

export default PlayButton;
```

Exercise 12: Improving the Operation Dropdown

10 to 15 minutes

Before we add more styles, let's make a quick improvement to the **Operation** dropdown. Rather than show the operation symbols, let's show the full words (e.g., "Multiplication" in place of "x").

1. In `Main.js`, change the definition of the `operations` constant to and array of arrays:

```
const operations = [
  ['Addition', '+'],
  ['Subtraction', '-'],
  ['Multiplication', 'x'],
  ['Division', '/']
];
```

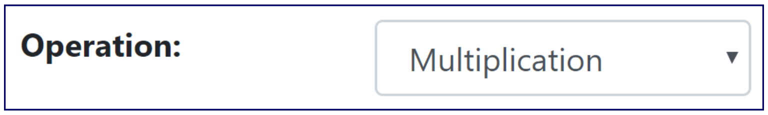
2. In `SelectInput.js`, change the `option` element to treat each value in the `values` array as a 2-element array containing a value and a text string:

```
<option value={value[1]} key={value[0].toString()}>{value[0]}</option>
```

3. This change will break our **Maximum Number** dropdown. To fix it, go back to `Main.js` and change the code to push an array onto `numbers` for each number:

```
const numbers = [];
for (let number = 2; number <= 100; number++) {
  numbers.push([number, number]);
}
```

The **Operation** dropdown should now look like this:



Solution: React/Solutions/styling/Main-2.js

```
-------Lines 1 through 5 Omitted-------
function Main(props) {
  const operations = [
    ['Addition', '+'],
    ['Subtraction', '-'],
    ['Multiplication', 'x'],
    ['Division', '/']
  ];
  const numbers = [];
  for (let number = 2; number <= 100; number++) {
    numbers.push([number, number]);
  }
-------Lines 17 through 41 Omitted-------
```

Solution: React/Solutions/styling/SelectInput-2.js

```
-------Lines 1 through 4 Omitted-------
  const selectOptions = values.map((value)=>
    <option value={value[1]} key={value[0].toString()}>{value[0]}</option>
  );
-------Lines 8 through 22 Omitted-------
```

6.2. Inline Styles

In HTML, it is possible to apply CSS properties to elements by passing them directly into the `style` attribute of the element, like this:

```
<p style="padding: 1em; color: blue; font-family: sans-serif;">
  Paragraph content.
</p>
```

But this is considered bad practice largely because it makes managing styles difficult.

In React, however, you're writing reusable components. Styles that you apply in a single React element define the base styles for that element, which may be reused many times in your user interface. React uses its own `style` attribute (written using JSX, not HTML and CSS) to apply styles inside of a component. Here's an example of using a `style` attribute in JSX:

```
<p style={
  {
      padding: "1em",
      color: "blue",
      fontFamily: "sans-serif"
  }
}>Paragraph content.</p>
```

The JSX `style` attribute looks somewhat similar to the HTML `style` attribute, but it has a few important differences because it is JavaScript:

1. The value of the `style` attribute is surrounded by curly braces, to indicate to the JSX processor that it is JavaScript code embedded in JSX.
2. Inside the first curly braces are another set of curly braces which create a JavaScript object.
3. The property names are JavaScript `style` properties, rather than CSS properties. The differences is that in JavaScript `style` properties, lowerCamelCase is used in place of dashes for multi-word property names: for example, CSS's `font-family` becomes `fontFamily`, `background-color` becomes `backgroundColor`, etc.
4. JavaScript `style` properties are separated by commas, rather than by semicolons.
5. The values of JavaScript `style` properties must adhere to the rules of JavaScript variables (for example, strings must be enclosed in quotes).

You can also create a `style` object like this:

```
const blueParagraph = {
  padding: "1em",
  color: "blue",
  fontFamily: "sans-serif"
}
```

Then, you can use the name of this object inside single curly braces as the value of one or multiple `style` attributes. If you use this technique, you can also define the `style` objects in a separate file and then import them into your components, which allows you to reuse the same styles for multiple components.

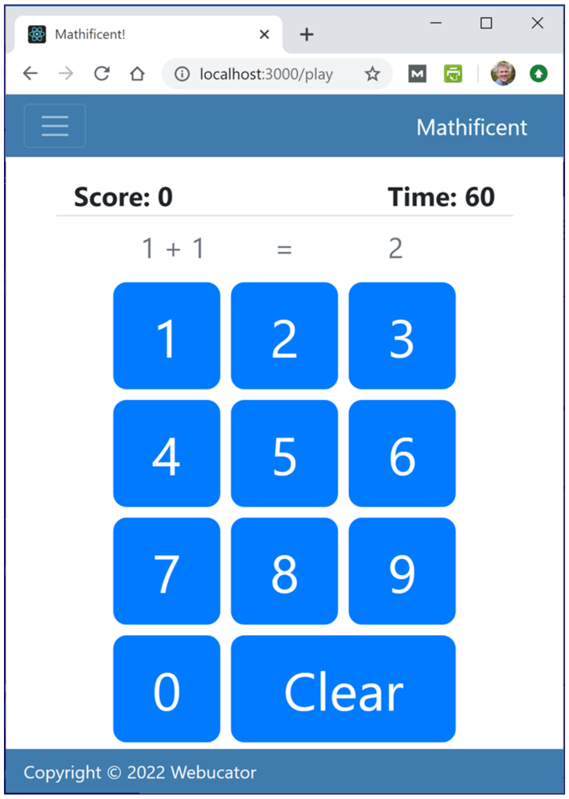
```
return (
  <p style={blueParagraph}></p>
);
```

Exercise 13: Creating the Game Component

25 to 40 minutes

In this exercise, you will create the `Game` component for Mathificent. The `Game` is where the user will see random math problems and will be able to enter the solutions to the problems. To see what the final game looks like, run `npm install` and then `npm start` from `mathificent-final` and play around. Be sure to stop the server before moving on to the exercise.

1. Here is what the game will look like when it is complete:



   Using this screenshot, try to identify the individual elements that make up the user interface of the game.

2. Think about how many different components you need to make to build this user interface. It has:

   A. A score.

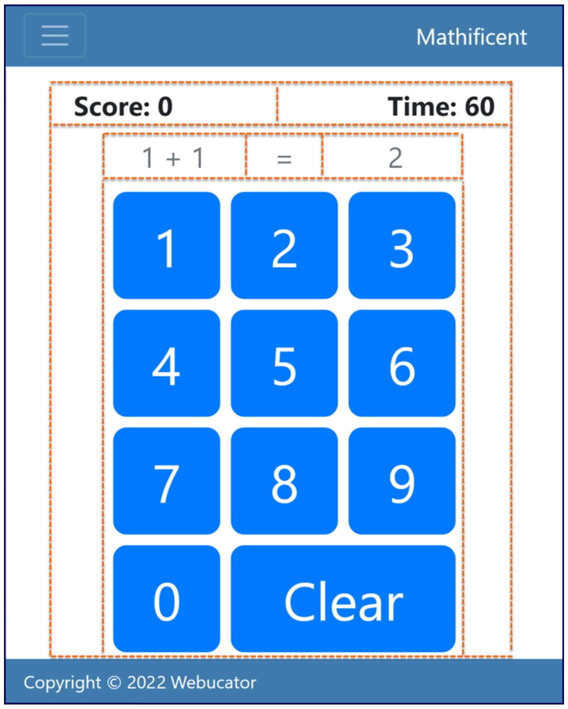
   B. A timer.

C. An equation.

D. Ten number buttons.

E. A clear button.

You could lay that out in rows and columns like this:



Note that the buttons are all in a single row, but are constrained by the width of the container.

3. Start up your development server in `Exercises/mathificent` if it is not running already. **Press the Play button to show the game. As you proceed through the exercise, keep an eye on the web browser to see how the interface changes.**

4. Create the following new components with the return values shown:

   A. `Score`

   ```
   <strong>Score: {props.score}</strong>
   ```

   B. `Timer`

   ```
   <strong>Time: {props.timeLeft}</strong>
   ```

C. `Equation`

```
<Fragment>
  <div className="col-5">{props.question}</div>
  <div className="col-2">=</div>
  <div className="col-5">{props.answer}</div>
</Fragment>
```

Be sure to import `Fragment` from 'react'.

D. `NumberButton`

```
<button className="btn btn-primary">{props.value}</button>
```

E. `ClearButton`

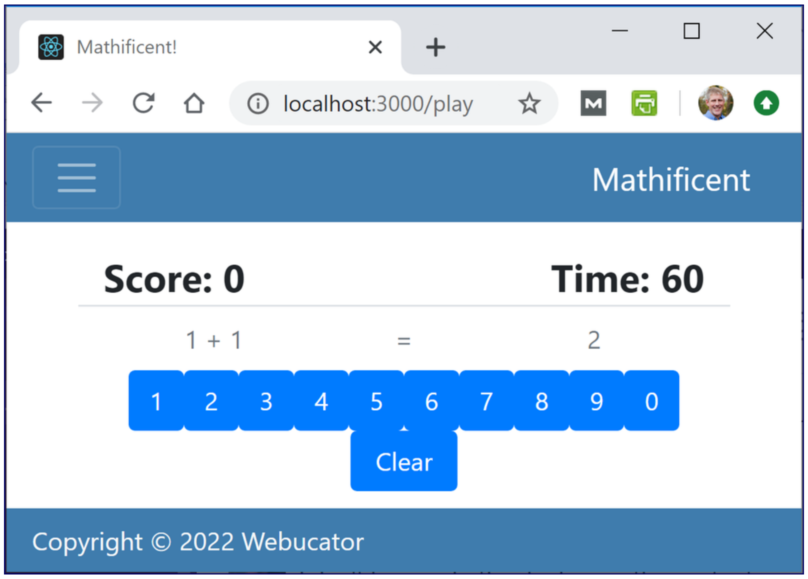
```
<button className="btn btn-primary">Clear</button>
```

5. Using `import` statements, include each of these new components into `Game`.
6. In the `return` statement of `Game`, place:
   A. One `Score` component.
   B. One `Timer` component.
   C. One `Equation` component.
   D. Ten `NumberButton` components.
   E. One `ClearButton` component.

Use the following code, which lays out the game using Bootstrap classes. Notice that we use attributes to pass in values to the child components:

```
<main className="text-center" id="game-container">
  <div className="row border-bottom">
    <div className="col px-3 text-left">
      <Score score="0" />
    </div>
    <div className="col px-3 text-right">
      <Timer timeLeft="60" />
    </div>
  </div>
  <div className="row text-secondary my-2" id="equation">
    <Equation question="1 + 1" answer="2" />
  </div>
  <div className="row" id="buttons">
    <div className="col">
      <NumberButton value="1" />
      <NumberButton value="2" />
      <NumberButton value="3" />
      <NumberButton value="4" />
      <NumberButton value="5" />
      <NumberButton value="6" />
      <NumberButton value="7" />
      <NumberButton value="8" />
      <NumberButton value="9" />
      <NumberButton value="0" />
      <ClearButton />
    </div>
  </div>
</main>
```

7. Your app should now look something like this:



Don't worry. We will make it better.

8. Create a new file in the `components` directory named `Game.css` and open it in your editor. Add the following code to `Game.css`:

```css
#game-container {
  margin: 1em auto;
  width: 380px;
}

#equation {
  font-size: 1.6em;
  margin: auto;
  width: 90%;
}

#buttons button {
  border-radius: .25em;
  font-size: 3em;
  height: 2em;
  margin: .1em;
  text-align: center;
  width: 2em;
}
```

9. Import `Game.css` into `Game.js`:

```
import './Game.css';
```

10. All the buttons, including the **Clear** button have the same width. We want the **Clear** button to be wider. We could create a `ClearButton.css` file, but that seems like a little overkill. Instead, add the following inline style in the `ClearButton` component:

```
<button className="btn btn-primary" style={{width: "4.2em"}}>Clear</button>
```

11. The only remaining thing we need to do is beef up the font size of the first row a little. In the `Game` component, add the following `style` to the first row:

```
<div className="row border-bottom" style={{fontSize: "1.5em"}}>
```

The app should now look like the screenshot at the beginning of this exercise.

❖ E13.1. Using map() with NumberButton

Although this doesn't pertain to styling, we can output the ten instances of the `NumberButton` component in a more efficient way by using an array and the `Array.map()` method:

1. In the `Game` component, use the following code to create an array of numbers and then loop through that array to create an array of `NumberButton` components. This should go inside the function, but before the `return` statement:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
const numberButtons = numbers.map((number) =>
  <NumberButton value={number} key={number} />
);
```

   Notice the `key` attribute. Any time you create a list of elements in React, it is important to supply a unique `key` attribute to each element in the list to help React perform updates in the most efficient way.

2. In the `return` statement, replace the list of `NumberButton` elements with `{numberButtons}`. This will cause the items in the array to be rendered individually, resulting in ten buttons being output.

The page should still look the same.

Solution: React/Solutions/styling/Game.css

```css
#game-container {
  margin: 1em auto;
  width: 380px;
}

#equation {
  font-size: 1.6em;
  margin: auto;
  width: 90%;
}

#buttons button {
  border-radius: .25em;
  font-size: 3em;
  height: 2em;
  margin: .1em;
  text-align: center;
  width: 2em;
}
```

Solution: React/Solutions/styling/Game.js

```
import React from 'react';
import Score from './Score';
import Timer from './Timer';
import Equation from './Equation';
import NumberButton from './NumberButton';
import ClearButton from './ClearButton';
import './Game.css';

function Game(props) {

  const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
  const numberButtons = numbers.map((number) =>
    <NumberButton value={number} key={number} />
  );

  return (
    <main className="text-center" id="game-container">
      <div className="row border-bottom" style={{fontSize: "1.5em"}}>
        <div className="col px-3 text-left">
          <Score score="0" />
        </div>
        <div className="col px-3 text-right">
          <Timer timeLeft="60" />
        </div>
      </div>
      <div className="row text-secondary my-2" id="equation">
        <Equation question="1 + 1" answer="2" />
      </div>
      <div className="row" id="buttons">
        <div className="col">
          {numberButtons}
          <ClearButton />
        </div>
      </div>
    </main>
  )
}

export default Game;
```

Solution: React/Solutions/styling/ClearButton.js

```
import React from 'react';

function ClearButton(props) {
  return (
    <button className="btn btn-primary"
      style={{width: "4.2em"}}>Clear</button>
  )
}

export default ClearButton;
```

Solution: React/Solutions/styling/NumberButton.js

```
import React from 'react';

function NumberButton(props) {
  return (
    <button className="btn btn-primary">{props.value}</button>
  )
}

export default NumberButton;
```

Solution: React/Solutions/styling/Equation.js

```
import React, {Fragment} from 'react';

function Equation(props) {
  return (
    <Fragment>
      <div className="col-5">{props.question}</div>
      <div className="col-2">=</div>
      <div className="col-5">{props.answer}</div>
    </Fragment>
  )
}

export default Equation;
```

Solution: React/Solutions/styling/Timer.js

```
import React from 'react';

function Timer(props) {
  return (
    <strong>Time: {props.timeLeft}</strong>
  )
}

export default Timer;
```

Solution: React/Solutions/styling/Score.js

```
import React from 'react';

function Score(props) {
  return (
    <strong>Score: {props.score}</strong>
  )
}

export default Score;
```

6.3. A Word of Caution

It is tempting to think that style modules imported into a component will only affect that component. However, that is not the case. To illustrate, we'll look at the following files:

Demo 6.3: React/demo-viewer/src/Demos/styling/Apple.js

```
import React from 'react';
import './Apple.css';

function Apple() {
    return (
      <div className="fruit">
        Apple
      </div>
    );
}

export default Apple;
```

Demo 6.4: React/demo-viewer/src/Demos/styling/Apple.css

```
.fruit {
  background-color: black;
  color: red;
  text-decoration: underline;
}
```

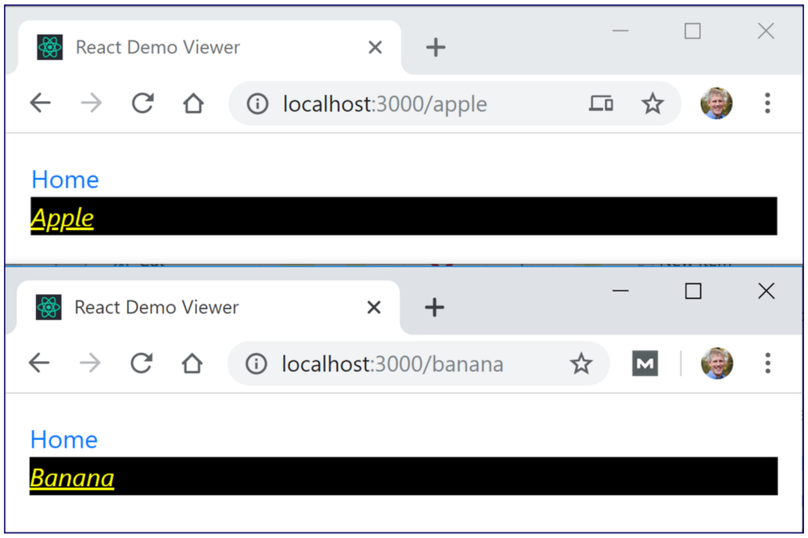
Demo 6.5: React/demo-viewer/src/Demos/styling/Banana.js

```
import React from 'react';
import './Banana.css';

function Banana() {
    return (
      <div className="fruit">
        Banana
      </div>
    );
}

export default Banana;
```

Demo 6.6: React/demo-viewer/src/Demos/styling/Banana.css

```
.fruit {
  color: yellow;
  font-style: italic;
}
```

Notice that the `Apple` and `Banana` components each import their own stylesheets. Both stylesheets create a `fruit` class. The classes are different though: the `fruit` class in `Apple.css` is red and underlined, while the `fruit` class in `Banana.css` is yellow and italic. But the end result, as shown in the following screenshot, is that all styles get applied to both components.



Because `color` was set in both CSS files, one value (in this case, yellow) overrides the other (red).

To see this in your browser, start up the demo-viewer app and click the **Apple** and **Banana** links under **Component Style**.

The takeaway is that you must be careful when using imported stylesheets. One solution is to use ids in combination with classes. For example, you could add "apple" and "banana" ids to the main `divs` in the two components and then change your CSS selectors to `#apple.fruit` and `#banana.fruit`.

Conclusion

In this lesson, you have learned how to use style in React components and applications.

LESSON 7
Implementing Game Logic

Topics Covered

☑ Working with your new React skills.

Introduction

In this lesson, you will build out most of the Mathificent game in a series of exercises.

Exercise 14: Setting the Equation

45 to 60 minutes

In this exercise, you will write the code to create the equations displayed in Mathificent.

1. Start up your development server in `Exercises/mathificent` if it is not running already. You should keep it running throughout this lesson.
2. We will need to generate random integers for the equation. The function for generating random integers is not specific to Mathificent, so we will put it in a separate `helpers.js` file and import it:
   A. Create a new folder within the `src` folder called `helpers`.
   B. Within the `helpers` folder, create a file called `helpers.js`.
   C. In the `Exercises/starter-code.txt` file, you will find a JavaScript function called `randInt()` that looks like this:

   ```
   export function randInt(low, high) {
     const rndDec = Math.random();
     return Math.floor(rndDec * (high - low + 1) + low);
   }
   ```

   Copy and paste that code into `helpers.js` and save.
   D. Open `Game.js` in your editor.
   E. Beneath the other imports, import the `randInt()` function from `helpers.js` using this code:

   ```
   import {randInt} from '../helpers/helpers';
   ```

3. Destructure `props` in the `Game()` function so that we can refer to `operation` and `maxNumber` directly without having to prefix them with `props`:

   ```
   function Game({operation, maxNumber}) {…
   ```

   Remember that `operation` and `maxNumber` are passed in from `App.js`:

   ```
   <Game operation={operation} maxNumber={maxNumber} />
   ```

4. Add the `getRandNumbers()` function at the beginning of the `Game()` function right after:

```
function Game({operation, maxNumber}) {
```

If you don't want to type it out, you can copy and paste the function from the `Exercises/starter-code.txt` file:

```
function getRandNumbers(operator, low, high) {
  let num1 = randInt(low, high);
  let num2 = randInt(low, high);
  const numHigh = Math.max(num1, num2);
  const numLow = Math.min(num1, num2);

  if(operator === '-') { // Make sure higher num comes first
    num1 = numHigh;
    num2 = numLow;
  }

  if(operator === '/') {
    if (num2 === 0) { // No division by zero
      num2 = randInt(1, high);
    }
    num1 = (num1 * num2); // product
  }
  return {num1, num2};
}
```

Review this function carefully. See how it returns an object with two numbers that will make up the equation: `num1` will be the operand before the operator and `num2` will be the operand after the operator.[1]

5. We are going to use some state variables in this component, so import `useState` from 'react':

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

6. Above the `getRandNumbers()` function that you just pasted in, type the following to create the state variables:

```
let randNums = getRandNumbers(operation, 0, maxNumber);
const [operands, setOperands] = useState(randNums);
```

1. The *operands* are the numbers being operated on by the *operator*. In `5 + 3`, `5` and `3` are the operands.

- `randNums` is a variable (rather than a constant) because we will get new random numbers each time the user answers a question correctly.
- We will assign new random numbers to the state variable `operands` using `setOperands()` every time we get a new question (equation).

7. Create a constant called `question` as shown here:

```
const question = operands.num1 + ' ' + operation +
  ' ' + operands.num2;
```

Remember that React will react to changes. So when the state variable `operands` changes, `question` will change as well.

8. Finally, pass `question` into the `<Equation>` tag in place of the placeholder text that is there now:

```
<Equation question={question} answer="2" />
```

9. Let's see how it works in your browser:
   A. On the Mathificent home page, click **Play!**. Notice that the equation is randomly generated.
   B. Refresh the page. Notice that the equation changes.
   C. Return to the home page and change the operation to division and maximum number to 100. Then, press **Play!** Notice that you get a division problem, possibly a difficult one.

10. But the user doesn't yet have a way to answer the question. We'll tackle that next.

Solution: React/Solutions/implementing/Game1.js

```
import React, {useState} from 'react';
-------Lines 2 through 7 Omitted-------
import {randInt} from '../helpers/helpers';

function Game({operation, maxNumber}) {

  let randNums = getRandNumbers(operation, 0, maxNumber);
  const [operands, setOperands] = useState(randNums);
  const question = operands.num1 + ' ' + operation +
                   ' ' + operands.num2;

  function getRandNumbers(operator, low, high) {
    let num1 = randInt(low, high);
    let num2 = randInt(low, high);
    const numHigh = Math.max(num1, num2);
    const numLow = Math.min(num1, num2);

    if(operator === '-') { // Make sure higher num comes first
      num1 = numHigh;
      num2 = numLow;
    }

    if(operator === '/') {
        if (num2 === 0) { // No division by zero
          num2 = randInt(1, high);
        }
        num1 = (num1 * num2); // product
      }
    return {num1, num2};
  }
-------Lines 36 through 41 Omitted-------
  return (
    <main className="text-center" id="game-container">
      <div className="row border-bottom" style={{fontSize: "1.5em"}}>
        <div className="col px-3 text-left">
          <Score score="0" />
        </div>
        <div className="col px-3 text-right">
          <Timer timeLeft="60" />
        </div>
      </div>
      <div className="row text-secondary my-2" id="equation">
        <Equation question={question} answer="2" />
      </div>
      <div className="row" id="buttons">
        <div className="col">
          {numberButtons}
          <ClearButton />
        </div>
      </div>
    </main>
  );
-------Lines 63 through 65 Omitted-------
```

Exercise 15: Getting the User's Answer

30 to 45 minutes

In this exercise, you will write the code to let the user answer the question and then follow that up with another random question. You will need to capture the user's clicks on the buttons and modify the user's answer accordingly.

1. In the `Game` component, add a state variable for the user's answer:

   ```
   const [userAnswer, setUserAnswer] = useState('');
   ```

   This must be a *state* variable because we need to share it between components and re-render when it changes.

2. Create a function for updating the user's answer. Call the function `appendToAnswer()`:

   ```
   function appendToAnswer(num) {
     setUserAnswer(userAnswer + num);
   }
   ```

   In the code that generates the `numberButtons` array, pass this function to the `NumberButton` components:

   ```
   const numberButtons = numbers.map((number) =>
     <NumberButton value={number} key={number}
     handleClick={appendToAnswer} />
   );
   ```

3. Open the `NumberButton` component and add an `onClick` event handler to update `userAnswer` when a button is clicked:

   ```
   <button className="btn btn-primary"
     onClick={() => {props.handleClick(props.value)}}>…
   ```

   This calls the function passed into `handleClick` via `props` (`appendToAnswer()`) and passes it the button's `value`. Remember `appendToAnswer()` just adds the passed-in value to `userAnswer`.

4. Finally, pass `userAnswer` into the `<Equation>` tag in place of the placeholder text that is there now:

```
<Equation question={question} answer={userAnswer} />
```

5. Test the solution in your browser:

   A. On the Mathificent home page, click **Play!**.

   B. Click some number buttons. The answer should update with your clicks.

   C. Refresh the page. Click the "0" button. Now click another button. Notice that the "0" stays at the beginning of the user's answer, which makes sense, because the `appendToAnswer()` function just appends the value of the clicked button to `userAnswer`. But it's not ideal. We would like answers like "09" to be converted to just "9". An easy way to do that in JavaScript is to convert the string to a number and then back to a string. For example:

   ```
   let num = '09';
   num = Number(num);
   num = String(num);
   console.log(num); // '9'
   ```

   In a single step, it would look like this:

   ```
   num = String(Number(num));
   ```

   Fix the `appendToAnswer()` function to get rid of leading "0"s:

   ```
   function appendToAnswer(num) {
     setUserAnswer(String(Number(userAnswer + num)));
   }
   ```

6. Let's get the `Clear` button working as well:

   A. We just have to make the `Clear` button set `userAnswer` back to an empty string. We can do that by passing it the `setUserAnswer` function:

   ```
   <ClearButton handleClick={setUserAnswer} />
   ```

B. Now, in the `ClearButton` component, we'll handle the click just as we did in the `NumberButton` component:

```
<button className="btn btn-primary" style={{width: "4.2em"}}
  onClick={() => {props.handleClick('');}}>Clear</button>
```

Remember that `props.handleClick` holds `setUserAnswer`, so a click ultimately results in a call to `setUserAnswer('')`, which clears `userAnswer`.

C. Try it out in your browser: just enter a wrong answer and press **Clear**. The answer should go away.

7. Now we need to check if the user answered the question correctly. We'll handle that next.

Solution: React/Solutions/implementing/Game2.js

```
-------Lines 1 through 9 Omitted-------
function Game({operation, maxNumber}) {

  let randNums = getRandNumbers(operation, 0, maxNumber);
  const [operands, setOperands] = useState(randNums);
  const question = operands.num1 + ' ' + operation +
                   ' ' + operands.num2;

  const [userAnswer, setUserAnswer] = useState('');

  function appendToAnswer(num) {
    setUserAnswer(String(Number(userAnswer + num)));
  }
-------Lines 22 through 42 Omitted-------
  const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
  const numberButtons = numbers.map((number) =>
    <NumberButton value={number} key={number}
      handleClick={appendToAnswer} />
  );

  return (
    <div className="text-center" id="game-container">
      <div className="row border-bottom" style={{fontSize: "1.5em"}}>
        <div className="col px-3 text-left">
          <Score score="0" />
        </div>
        <div className="col px-3 text-right">
          <Timer timeLeft="60" />
        </div>
      </div>
      <div className="row text-secondary my-2" id="equation">
        <Equation question={question} answer={userAnswer} />
      </div>
      <div className="row" id="buttons">
        <div className="col">
          {numberButtons}
          <ClearButton handleClick={setUserAnswer} />
        </div>
      </div>
    </div>
  );
-------Lines 70 through 72 Omitted-------
```

Solution: React/Solutions/implementing/NumberButton.js

```
import React from 'react';

function NumberButton(props) {
  return (
    <button className="btn btn-primary"
      onClick={() => {props.handleClick(props.value)}}>
        {props.value}
    </button>
  )
}

export default NumberButton;
```

Solution: React/Solutions/implementing/ClearButton.js

```
import React from 'react';

function ClearButton(props) {
  return (
    <button className="btn btn-primary" style={{width: "4.2em"}}
      onClick={() => {props.handleClick('');}}>Clear</button>
  );
}

export default ClearButton;
```

Exercise 16: Checking the User’s Answer

45 to 60 minutes

In this exercise, you will write the code to handle correct answers. You will need to:

1. Compare the user’s answer with the correct answer.
2. When the user’s answer matches the correct answer, increment the score by 1 and show a new question.

Let’s give it a try.

1. In the `Game` component, add a state variable for the score:

   ```
   const [score, setScore] = useState(0);
   ```

   And pass `score` into the `<Score>` tag in place of the placeholder text that is there now:

   ```
   <Score score={score} />
   ```

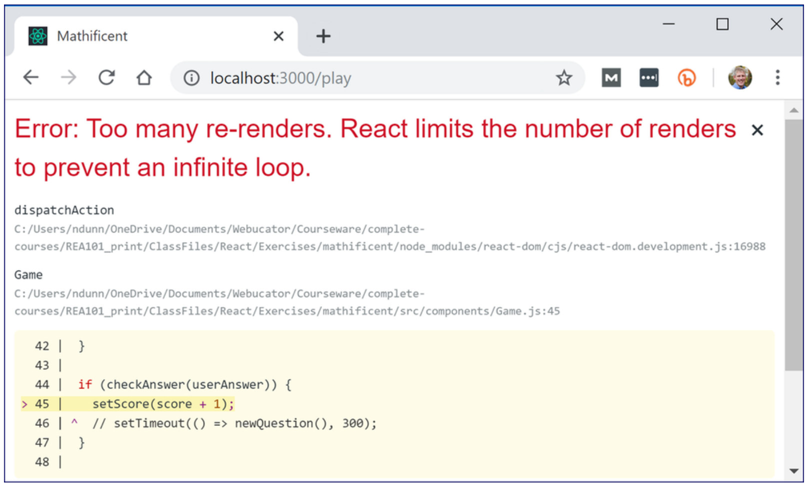
2. Add a `checkAnswer()` function to compare the user's answer with the correct answer. If you don't want to type it out, you can copy and paste the function from the `Exercises/starter-code.txt` file:

```
function checkAnswer(userAnswer) {
  if (isNaN(userAnswer)) return false; // User hasn't answered

  let correctAnswer;
  switch(operation) {
    case '+':
      correctAnswer = operands.num1 + operands.num2;
      break;
    case '-':
      correctAnswer = operands.num1 - operands.num2;
      break;
    case 'x':
      correctAnswer = operands.num1 * operands.num2;
      break;
    default: // division
      correctAnswer = operands.num1 / operands.num2;
  }
  return (parseInt(userAnswer) === correctAnswer);
}
```

3. Add the following code below the `checkAnswer()` function to call `checkAnswer()` on each re-rendering:

```
if (checkAnswer(userAnswer)) {
  setScore(score + 1);
}
```

4. Try this out. When you get a correct answer, you will either get a blank page or get an error similar to this one:



```
Error: Too many re-renders. React limits the number of renders
to prevent an infinite loop.

dispatchAction
C:/Users/ndunn/OneDrive/Documents/Webucator/Courseware/complete-
courses/REA101_print/ClassFiles/React/Exercises/mathificent/node_modules/react-dom/cjs/react-dom.development.js:16988

Game
C:/Users/ndunn/OneDrive/Documents/Webucator/Courseware/complete-
courses/REA101_print/ClassFiles/React/Exercises/mathificent/src/components/Game.js:45

  42 |   }
  43 |
  44 |   if (checkAnswer(userAnswer)) {
> 45 |     setScore(score + 1);
  46 | ^   // setTimeout(() => newQuestion(), 300);
  47 |   }
  48 |
```

If you get a blank page, you can see the error in the JavaScript console. The reason for this error is that `userAnswer` has been set to a value equal to the correct answer, so we get stuck in this infinite loop:

A. `if (checkAnswer(userAnswer))` - The condition is `true`, so we execute the body of the condition…

B. `setScore(score + 1);` - This changes a state variable, which results in a re-rendering…

C. When the page is re-rendered, it will check the `if` condition again: `if (checkAnswer(userAnswer))`. As neither `userAnswer` nor the correct answer have been modified, the condition returns `true` again, taking us back to step B in which we change the value of `score`. And this will repeat infinitely.

5. We will handle this problem by creating another state variable named `answered`, which we will set to `true` when the user answers a question. Then we will modify the `if` condition to only call `checkAnswer()` if `answered` is `false`. Below the `userAnswer` state variable, create the `answered` state variable:

```
const [answered, setAnswered] = useState(false);
```

Modify the `if` condition as follows:

```
if (!answered && checkAnswer(userAnswer)) {
  setAnswered(true);
  setScore(score + 1);
}
```

Now, the first time that `(!answered && checkAnswer(userAnswer))` returns `true`, `answered` will be set to `true` as well, so on the next re-rendering, the `(!answered && checkAnswer(userAnswer))` will return `false`.

6. Try it again in the browser. This time, when you answer the question correctly, the score should update to `1`.

7. Next, we need to change the question after the user answers correctly. Add the following function to the `Game` component:

```
function newQuestion() {
  setUserAnswer('');
  setAnswered(false);
  randNums = getRandNumbers(operation, 0, maxNumber);
  setOperands(randNums);
}
```

This sets `userAnswer` back to an empty string, sets `answered` back to `false`, assigns new random numbers to `randNums` and assigns them to `operands`. As this function changes several state variables, it will result in a re-rendering, showing the new equation. Now we just need to call the function after the user gets a correct answer. Do that at the end of the `if` condition checking for a correct answer:

```
if (!answered && checkAnswer(userAnswer)) {
  setAnswered(true);
  setScore(score + 1);
  newQuestion();
}
```

8. Try it again in the browser. You should now be able to answer question after question for forever and ever.

9. The question changes are a little abrupt. It'd be nice to fade the old question out. We will use the Bootstrap "fade" class for this. In the `Game` component, add a constant called `equationClass` whose value depends on the value of `answered`. You can add it right before the `return` statement:

```
const equationClass = answered
  ? 'row my-2 text-primary fade'
  : 'row my-2 text-secondary';
```

Now, change the class of the `div` containing the `<Equation>` tag to use `equationClass`:

```
<div className={equationClass} id="equation">
  <Equation question={question} answer={userAnswer} />
</div>
```

Finally, we need to add a little delay before getting the next question so that the user has time to see the fade effect. In the `if` condition where you call `newQuestion()`, use a `setTimeout` to delay that call by 300 milliseconds:

```
if (!answered && checkAnswer(userAnswer)) {
  setAnswered(true);
  setScore(score + 1);
  setTimeout(newQuestion, 300);
}
```

Things are working pretty well! Time to implement our timer.

Solution: React/Solutions/implementing/Game3.js

```
-------Lines 1 through 9 Omitted-------
function Game({operation, maxNumber}) {

  let randNums = getRandNumbers(operation, 0, maxNumber);
  const [operands, setOperands] = useState(randNums);
  const question = operands.num1 + ' ' + operation +
                  ' ' + operands.num2;

  const [userAnswer, setUserAnswer] = useState('');
  const [answered, setAnswered] = useState(false);
  const [score, setScore] = useState(0);

  function appendToAnswer(num) {
    setUserAnswer(String(Number(userAnswer + num)));
  }

  function checkAnswer(userAnswer) {
    if (isNaN(userAnswer)) return false; // User hasn't answered

    let correctAnswer;
    switch(operation) {
      case '+':
        correctAnswer = operands.num1 + operands.num2;
        break;
      case '-':
        correctAnswer = operands.num1 - operands.num2;
        break;
      case 'x':
        correctAnswer = operands.num1 * operands.num2;
        break;
      default: // division
        correctAnswer = operands.num1 / operands.num2;
    };
    return (parseInt(userAnswer) === correctAnswer);
  }

  if (!answered && checkAnswer(userAnswer)) {
    setAnswered(true);
    setScore(score + 1);
    setTimeout(newQuestion, 300);
  }

  function newQuestion() {
    setUserAnswer('');
53.       setAnswered(false);
54.       randNums = getRandNumbers(operation, 0, maxNumber);
55.       setOperands(randNums);
56.     }
      -------Lines 57 through 83 Omitted-------
84.     const equationClass = answered
85.       ? 'row my-2 text-primary fade'
86.       : 'row my-2 text-secondary';
87.
88.     return (
89.       <div className="text-center" id="game-container">
90.         <div className="row" style={{fontSize: "1.5em"}}>
91.           <div className="col px-3 text-left">
92.             <Score score={score} />
93.           </div>
94.           <div className="col px-3 text-right">
95.             <Timer timeLeft="60" />
96.           </div>
97.         </div>
98.         <div className={equationClass} id="equation">
99.           <Equation question={question} answer={userAnswer} />
100.        </div>
101.        <div className="row" id="buttons">
102.          <div className="col">
103.            {numberButtons}
104.            <ClearButton handleClick={setUserAnswer} />
105.          </div>
106.        </div>
107.      </div>
108.    );
109.  }
110.
111.  export default Game;
```

Exercise 17: Creating the Timer

20 to 30 minutes

In this exercise, you will add the countdown timer.

1. Open the `Timer` component. Add the following code above the `return` statement:

   ```
   if (props.timeLeft > 0) {
     setTimeout(() => {
       props.setTimeLeft(props.timeLeft - 1);
     }, 1000)
   };
   ```

   This uses `setTimeout()` to decrement `timeLeft` by 1 every 1000 milliseconds (1 second) until `timeLeft` is `0`.

2. Open the `Game` component. Below the `score` state variable, create a `gameLength` constant to hold the number of seconds a game should last:

   ```
   const gameLength = 60;
   ```

   Below that create a `timeLeft` state variable and set it to `gameLength`:

   ```
   const [timeLeft, setTimeLeft] = useState(gameLength);
   ```

3. Change the `<Timer>` tag to pass `timeLeft` and `setTimeLeft` to the `Timer` component:

   ```
   <Timer timeLeft={timeLeft} setTimeLeft={setTimeLeft} />
   ```

4. Your timer should be working now. The next step is to change the screen when `timeLeft` hits `0`. Add the following `restart()` function. You can place it below the `newQuestion()` function:

   ```
   function restart() {
     setTimeLeft(gameLength);
     setScore(0);
     newQuestion();
   }
   ```

   This sets `timeLeft` back to `gameLength` and `score` back to `0`. It also generates a new question.

5. Add the following code above the `return` statement:

```
if (timeLeft === 0) {
  return (
    <div className="text-center" id="game-container">
      <h2>Time's Up!</h2>
      <strong style={{fontSize: "1.5em"}}>You Answered</strong>
      <div style={{fontSize: "5em"}}>{score}</div>
      <strong style={{fontSize: "1.5em"}}>
        Questions Correctly
      </strong>
      <button className="btn btn-primary form-control m-1"
        onClick={restart}>
          Play Again with Same Settings
      </button>
      <Link className="btn btn-secondary form-control m-1" to="/">
        Change Settings
      </Link>
    </div>
  )
}
```

Remember that once a function returns something, it is finished executing. In this case, we are checking if `timeLeft` is equal to `0`. If it is, we are returning a **Time's Up!** screen. If it isn't, we move past this code and return the **Game** screen. Two important things to note:

- Clicking the **Play Again with Same Settings** button will call `restart()`, which will set `timeLeft` back to `gameLength` resulting in a re-rendering. On that re-rendering, since `timeLeft` will no longer be `0`, the **Game** screen will be displayed.
- The `Link` element points to "/", which leads to the home page. For the `Link` tag to work, we need to import `Link`. Add this code below the first `import` at the top of the page:

  ```
  import {Link} from 'react-router-dom';
  ```

Everything should work now, including the timer, but it's a little glitchy. If you haven't noticed any glitches, try this:

1. Change the value of `gameLength` to `5`, so that games end quickly.

2. Start playing. Set yourself up to answer a question correctly right before the timer runs out. For example, if the question is **5 + 7**, click the **1** button and then wait until the timer reads "1". As soon as it does, click the **2** button to answer "12".
3. As soon as the **Time's Up!** screen shows up, click **Play Again with Same Settings**.
4. The result will be that the screen switches between the **Time's Up!** screen and the **Game** screen once or twice before settling on the **Game** screen. If you don't see this the first time, try it a couple of times. We will learn how to fix this in the next lesson.

Solution: React/Solutions/implementing/Game4.js

```
1.   import React, {useState} from 'react';
2.   import {Link} from 'react-router-dom';
     -------Lines 3 through 21 Omitted-------
22.    const gameLength = 60; // Seconds
23.    const [timeLeft, setTimeLeft] = useState(gameLength);
     -------Lines 24 through 61 Omitted-------
62.    function restart() {
63.      setTimeLeft(gameLength);
64.      setScore(0);
65.      newQuestion();
66.    }
67.
     -------Lines 68 through 97 Omitted-------
98.    if (timeLeft === 0) {
99.      return (
100.       <div className="text-center" id="game-container">
101.         <h2>Time's Up!</h2>
102.         <strong style={{fontSize: "1.5em"}}>You Answered</strong>
103.         <div style={{fontSize: "5em"}}>{score}</div>
104.         <strong style={{fontSize: "1.5em"}}>Questions Correctly</strong>
105.         <button className="btn btn-primary form-control m-1"
106.           onClick={restart}>
107.             Play Again with Same Settings
108.         </button>
109.         <Link className="btn btn-secondary form-control m-1" to="/">
110.           Change Settings
111.         </Link>
112.       </div>
113.     );
114.   }
115.
116.   return (
117.     <div className="text-center" id="game-container">
118.       <div className="row" style={{fontSize: "1.5em"}}>
119.         <div className="col px-3 text-left">
120.           <Score score={score} />
121.         </div>
122.         <div className="col px-3 text-right">
123.           <Timer timeLeft={timeLeft} setTimeLeft={setTimeLeft} />
124.         </div>
125.       </div>
126.       <div className={equationClass} id="equation">
127.         <Equation question={question} answer={userAnswer} />
```

```
      </div>
      <div className="row" id="buttons">
        <div className="col">
          {numberButtons}
          <ClearButton handleClick={setUserAnswer} />
        </div>
      </div>
    </div>
  );
-------Lines 137 through 139 Omitted-------
```

Solution: React/Solutions/implementing/Timer.js

```
import React from 'react';

function Timer(props) {

  if (props.timeLeft > 0) {
    setTimeout(() => {
      props.setTimeLeft(props.timeLeft - 1);
    }, 1000)
  };

  return (
    <strong>Time: {props.timeLeft}</strong>
  )
}

export default Timer;
```

Conclusion

In this lesson, you have used your React and JavaScript skills to build out the Mathificent game. We have a couple of improvements to make, but the game is usable at this point.

LESSON 8
React Effects

Topics Covered

☑ The purpose of hooks.

☑ The `useEffect` hook.

Introduction

Hooks make it possible to add functionality to a React application without adding new components. In this lesson, you'll learn how to make use of React's built-in `useEffect` hook.

8.1. React Hooks

Hooks are functions that let you hook into React state and lifecycle features from function components. To use built-in React Hooks, you need to import the hook from the React library. For example, we have already been using the `useState` hook, which is imported like this:

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

Once you've imported the Hook, you can call it like you would call any function. It's important to be aware of what values to pass as arguments to the Hook and what values it will return. You can find out about all the details in the React Hooks API documentation[2].

The `useState` hook accepts one argument, which is the initial value of the stateful variable you're creating, and returns a stateful variable and a function for updating that variable.

2. https://reactjs.org/docs/hooks-reference.html

8.2. The useEffect Hook

The `useEffect` hook gives you access to the lifecycle of function components. For example, it's quite common to need to do some sort of setup or data retrieval as soon as a component first loads (or "mounts"). With the `useEffect` hook, you can run a function when a function component mounts, when it's updated, when it's about to be unmounted, or when all of these events happen. You can also specify specific state variables that, when changed, will trigger the effect.

❖ 8.2.1. The Need for useEffect

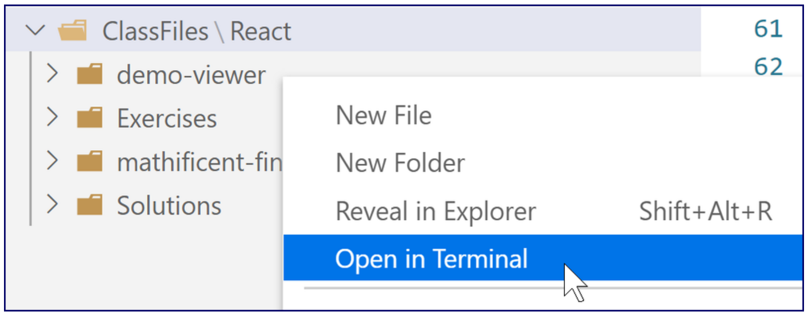
To understand the need for the `useEffect` hook, we'll first look at a component that doesn't use it:

Demo 8.1: React/demo-viewer/src/Demos/effect-hook/LightBulb1.js

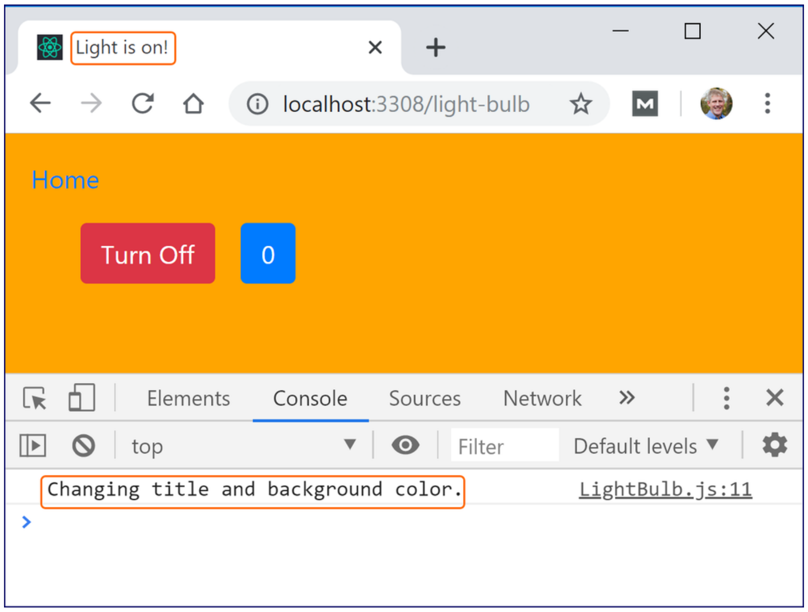
```
import React, {useState} from 'react';
import ToggleButton from './ToggleButton';

function LightBulb1() {

  const [on, setOn] = useState(true);
  const [count, setCount] = useState(0);

  document.title = on ? 'Light is on!' : 'Light is off!';
  document.body.style.backgroundColor = on ? 'orange' : 'white';
  console.log('Changing title and background color.');

  return (
    <div className="container">
      <ToggleButton on={on} setOn={setOn} />
      <button className="btn btn-primary" onClick={() => {
        setCount(count + 1);
      }}>{count}</button>
    </div>
  )
}
export default LightBulb1;
```

Code Explanation

1. To run this demo, open `React/demo-viewer` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



2. Run `npm start` to launch the demo-viewer React application.
3. Open the Chrome developer console.
4. Click the **LightBulb1** link under **useEffect**. You should see a page like the one shown below:



```
Changing title and background color.                LightBulb.js:11
```

Notice that the document title on the tab reads “Light is on!” That is a result of this code, which uses JavaScript’s ternary operator to set a value for `document.title` based on the value of `on`, which is initially set to `true`:

```
document.title = on ? 'Light is on!' : 'Light is off!';
```

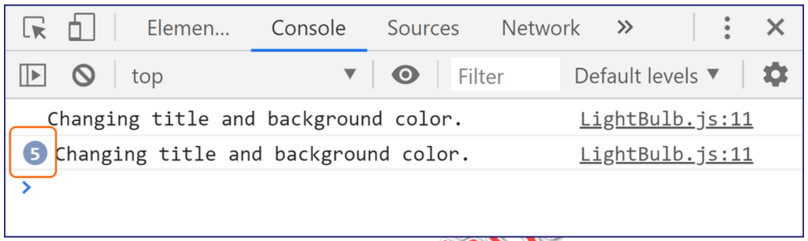
Also, notice that the background color of the page has been set to orange. That is a result of this code:

```
document.body.style.backgroundColor = on ? 'orange' : 'white';
```

Finally, notice that "Changing title and background color." has been logged to the console, which is a result of this line:

```
console.log('Changing title and background color.');
```

5. The important thing to realize is that those lines of code will run **every time** this component is rendered, which will happen with every change of state. To see this, click the first button, which just toggles the value of the state variable `on` between `true` and `false` and changes the text of the button accordingly. Notice that each time you press the button, the title on the tab changes and "Changing title and background color." gets logged to the console. That may be represented by a number in a circle indicating the number of times the page has logged the same thing:



```
Changing title and background color.    LightBulb.js:11
5 Changing title and background color.  LightBulb.js:11
```

When you click the **Turn On / Turn Off** button, the logging is accurate: the title and background color are indeed changing. But now click the second button, which just keeps track of the number of times it has been clicked by changing the state variable `count`. Notice that the logging to the console continues. Again, this is because all the code on this page runs **every time** this component is rendered. By putting that code in a `useEffect` hook, we can limit it to run only when the `on` variable changes.

6. There is another issue with this component. To see it, turn the light on so that the background color changes to orange. Then click the **Home** link. Notice that the background color stays orange and the title still says "Light is on!". We would like to set the background color back to white when the component "unmounts."

8.3. useEffect to the Rescue

In our last demo, we saw two problems:

1. Code that we only wanted to run on *some* changes of state was running on *every* change of state.

2. Changes that affected the entire app (the change in background color) didn't get cleaned up when the component unmounted.

The following demo will use a `useEffect` hook to address both those issues:

Demo 8.2: React/demo-viewer/src/Demos/effect-hook/LightBulb2.js

```
import React, {useState, useEffect} from 'react';
import ToggleButton from './ToggleButton';

function LightBulb2() {

  const [on, setOn] = useState(true);
  const [count, setCount] = useState(0);

  useEffect(() => {
    const originalTitle = document.title;
    document.title = on ? 'Light is on!' : 'Light is off!';
    document.body.style.backgroundColor = on ? 'orange' : 'white';
    console.log('Changing title and background color.');
    return () => {
      document.body.style.backgroundColor = 'white';
      document.title = originalTitle;
    }
  }, [on]);

  return (
    <div className="container">
      <ToggleButton on={on} setOn={setOn} />
      <button className="btn btn-primary" onClick={() => {
        setCount(count + 1);
      }}>{count}</button>
    </div>
  )
}
export default LightBulb2;
```

Code Explanation

1. First, clear the console. Then, on the demo-viewer home page, click the **LightBulb2** link under **useEffect**. This looks exactly like the last demo, but it's not!
2. Click the **Turn On / Turn Off** button a few times. Just like before, it changes the title in the tab and the background color, and it logs to the console.

3. Click the `count` button several times. Notice that it does not log to the console. That's good. That button doesn't change the title or background color, so it shouldn't log that it does.
4. Turn the light on so that the background color is orange, and then click the **Home** link. Notice that the background color reverts to white and the title reverts to "React Demo Viewer". This is good too.

We have fixed the issues by importing useEffect:

```
import React, {useState, useEffect} from 'react';
```

And then moving the code that should respond to changes in the value of `on` into a useEffect hook:

```
useEffect(() => {
  const originalTitle = document.title;
  document.title = on ? 'Light is on!' : 'Light is off!';
  document.body.style.backgroundColor = on ? 'orange' : 'white';
  console.log('Changing title and background color.');
  return () => {
    document.body.style.backgroundColor = 'white';
    document.title = originalTitle;
  }
}, [on]);
```

Notice that the `useEffect()` function takes two arguments:

1. A callback function:

```
() => {
  const originalTitle = document.title;
  document.title = on ? 'Light is on!' : 'Light is off!';
  document.body.style.backgroundColor = on ? 'orange' : 'white';
  console.log('Changing title and background color.');
  return () => {
    document.body.style.backgroundColor = 'white';
    document.title = originalTitle;
  }
}
```

2. An array of dependencies:

```
[on]
```

The callback function will only run when one of the variables in the dependencies array changes.

Notice that the callback function returns another function:

```
return () => {
  document.body.style.backgroundColor = 'white';
  document.title = originalTitle;
}
```

This returned function is often called the “cleanup” function, because it runs when the component unmounts. In our case, we are using it to set the background color back to white and the title back to the original title.

8.4. Mount and Unmount

When you have code that should only execute when the component is mounted (i.e., first loaded), put it in a `useEffect` hook with an empty dependencies array, like this:

```
useEffect(() => {
  // only run on mounting
}, []);
```

To add code that should run when the component unmounts, add a `return` statement to the callback function in the `useEffect`:

```
useEffect(() => {
  // only run on mounting
  return () => {
    // only run on unmounting
  }
}, []);
```

In the next demo, we have added a `useEffect` function to our `LightBulb` component to set the title when the component mounts and set it back to the original title when the component unmounts. This is a common thing to do in a React application, so the title reflects where the user is in the app.

Demo 8.3: React/demo-viewer/src/Demos/effect-hook/LightBulb3.js

```
import React, {useState, useEffect} from 'react';
import ToggleButton from './ToggleButton';

function LightBulb3() {

  const [on, setOn] = useState(true);
  const [count, setCount] = useState(0);

  useEffect(() => {
    const originalTitle = document.title;
    return () => {
      document.title = originalTitle;
    }
  }, []);

  useEffect(() => {
    document.title = on ? 'Light is on!' : 'Light is off!';
    document.body.style.backgroundColor = on ? 'orange' : 'white';
    console.log('Changing title and background color.');
    return () => {
      document.body.style.backgroundColor = 'white';
    }
  }, [on]);

  return (
    <div className="container">
      <ToggleButton on={on} setOn={setOn} />
      <button className="btn btn-primary" onClick={() => {
        setCount(count + 1);
      }}>{count}</button>
    </div>
  )
}
export default LightBulb3;
```

You might also use a `useEffect` function with an empty dependencies array to fetch data from a data feed when the component mounts and then clean up that data when the component unmounts.

8.5. Passing Functions to State Variable Setters

The usual way to change a state variable is to pass its setter a new value. For example, imagine you have a state variable named `on`:

```
const [on, setOn] = useState(true);
```

You can change its value like this:

```
setOn(false);
```

You could also reference the variable itself when making the change:

```
setOn(!on);
```

This will result in toggling the value of `on` between `true` and `false`.

Take a look at the following demo:

Demo 8.4: React/demo-viewer/src/Demos/effect-hook/LightBulb4.js

```
import React, {useState, useEffect} from 'react';

function LightBulb4() {

  const [on, setOn] = useState(true);

  document.body.style.backgroundColor = on ? 'orange' : 'white';

  useEffect(() => {
    window.addEventListener('click', () => {
      setOn(!on);
    })
  }, []);

  return (
    <div className="container">
      {on ? 'Light is on!' : 'Light is off!'}
    </div>
  )
}
export default LightBulb4;
```

Code Explanation

1. Open the demo-viewer app in the browser and click the **LightBulb4** link below **useEffect**.
2. Click anywhere on the page. Notice the light turns off.
3. Click anywhere on the page again. Nothing happens. The reason nothing happens is that the code that creates the event listener only runs when the `LightBulb4` component mounts. At that point the value of `on` is `true`. Because the `useEffect()` function doesn't run on each re-rendering, it doesn't realize the value of `on` is changing.
4. You will also note that you get a warning about a missing dependency: 'on'. You might be tempted to add `on` to the dependency array like this:

```
useEffect(() => {
  window.addEventListener('click', () => {
    setOn(!on);
  })
}, [on]);
```

But if you do that, a new event listener will be added with every mouse click. It might work, but after a lot of mouse clicks, you might find your app slowing down a bit.

5. The correct solution is to pass a function to the `setOn()` setter, like this:

```
useEffect(() => {
  window.addEventListener('click', () => {
    setOn((prevOn) => {
      return !prevOn;
    })
  })
}, []);
```

   When you pass a function to a state variable setter, the current value of the state variable is automatically passed to that function as an argument. The common practice is to call that parameter `prevVarName`. For example, if the state variable name is `on`, the name of the function parameter would be `prevOn`. But this is just by convention. Technically, you can call it whatever you like.

6. JavaScript provides a shortcut way of returning a value from a function when the function body only has a `return` statement:

```
setOn((prevOn) => !prevOn);
```

   And because this function only takes one argument, JavaScript allows you to remove the parentheses around the argument:

```
setOn(prevOn => !prevOn);
```

   This is a common syntax in React. Another example is incrementing a state variable by `1`:

```
setCount(prevCount => prevCount + 1);
```

We will use this technique in the next exercise.

Exercise 18: Fixing the Timer

20 to 30 minutes

You'll remember that moving from the Mathificent **Game** screen to the **Time's Up!** screen was sometimes a little glitchy. That's because the `Timeout` set in the `Timer` component is still active when the component unmounts. We need to make sure that gets cleaned up when the `Timer` component unmounts.

Let's take a look at the `Timer` component as it now stands:

Exercise Code 18.1: React/Solutions/implementing/Timer.js

```
import React from 'react';

function Timer(props) {

  if (props.timeLeft > 0) {
    setTimeout(() => {
      props.setTimeLeft(props.timeLeft - 1);
    }, 1000)
  };

  return (
    <strong>Time: {props.timeLeft}</strong>
  )
}

export default Timer;
```

1. Start up the Mathificent app from your `Exercises/mathificent` folder.
2. Change the `gameLength` in the `Game` component to 5, so that games finish quickly.
3. Looking back at the `Timer` component, let's review how it works:
   A. When the component mounts, it checks if `props.timeLeft` is greater than `0`. If it is, it sets a *Timeout* to decrement `props.timeLeft` by `1` in `1000` milliseconds.
   B. When `1000` milliseconds passes, the `props.timeLeft` will change, which will cause the component to re-render. This will result in a new *Timeout* being set to decrement `props.timeLeft` by `1` again.

   C. This cycle will continue until `props.timeLeft` is equal to `0`, at which point the `Game` component will show the **Times Up!** screen causing the `Timer` component to unmount.

4. Now, lets consider how we would like this to work:
   A. When the component mounts, instead of setting a *Timeout*, set an *Interval* using `setInterval()` to decrement `props.timeLeft` every `1000` milliseconds. Make sure this Interval only gets set when the Timer component originally mounts, so that we're not creating new *Intervals* every time the component re-renders.
   B. When the component unmounts, clear the *Interval* using `clearInterval()`.

5. Open the `Timer` component in your editor.

6. Import the `useEffect` hook:

```
import React, {useEffect} from 'react';
```

7. Destructure the `props` parameter so that we can reference the variables without the `props` prefix:

```
function Timer({timeLeft, setTimeLeft}) {
```

8. Replace the `if` condition and the contained `setTimeout()` code with this:

```
useEffect(() => {
  const timer = setInterval(() => {
      setTimeLeft(prevTimeLeft => prevTimeLeft - 1);
    }, 1000);
  return () => {
    clearInterval(timer); //cleanup
  }
}, []);
```

Because the dependencies array is empty, this code will only run when the `Timer` mounts, which will cause `timeLeft` to be decremented by `1` every `1000` milliseconds. The returned function, which clears the interval, will run immediately before the `Timer` unmounts, which will remove the glitchiness we saw before when switching between the **Times Up!** screen and the **Game** screen.

9. Note that you may get a warning saying that there is a missing dependency. This is because the linter (i.e., the tool that analyzes the code to see if there are any problems) *thinks* that the

value of `setTimeLeft` could change. But `setTimeLeft` is a function and will not change, so you don't need to worry about this. You can make the warning go away by adding this comment right after the `return` statement in the `useEffect` function:

```
// eslint-disable-next-line react-hooks/exhaustive-deps
```

Solution: React/Solutions/effect-hook/Timer.js

```
import React, {useEffect} from 'react';

function Timer({timeLeft, setTimeLeft}) {

  useEffect(() => {
    const timer = setInterval(() => {
        setTimeLeft(prevTimeLeft => prevTimeLeft - 1);
      }, 1000);
      console.log('Starting timer.');
    return () => {
      clearInterval(timer); //cleanup
      console.log('Cleaning up.');
    }
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  return (
    <strong>Time: {timeLeft}</strong>
  )
}
export default Timer;
```

Code Explanation

Note that we've added some logging to the console just for demonstration purposes. You do not need to include that.

Exercise 19: Catching Keyboard Events

10 to 20 minutes

To finish up our Mathificent app, we will add a component that allows the user to answer using the keyboard. This component is already created for you and shown below:

Exercise Code 19.1: React/Solutions/effect-hook/Keyboard.js

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

function Keyboard({setUserAnswer}) {

  useEffect(() => {
    const handleKeyUp = (e) => {
      e.preventDefault(); // prevent the normal behavior of the key
      if (e.keyCode === 32 || e.keyCode === 13) { // space/Enter
        setUserAnswer('');
      } else if (e.keyCode === 8) { // backspace
        setUserAnswer(prevUserAnswer =>
          prevUserAnswer.substring(0, prevUserAnswer.length - 1));
      } else if (!isNaN(e.key)) {
        // Number() will remove leading zeroes
        setUserAnswer(prevUserAnswer =>
          String(Number(prevUserAnswer + e.key)));
      }
    }
    window.addEventListener('keyup', handleKeyUp);
    return () => {
      window.removeEventListener('keyup', handleKeyUp); //cleanup
    }
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []); // No dependencies. Will only run on mounting

  return null; // This component doesn't output anything
}
export default Keyboard;
```

Code Explanation

You should be able to understand this code. The only new React-related bit is that the component doesn't output anything. We use `return null` to indicate that.

1. If it's not already running, start up the Mathificent app from your `Exercises/mathificent` folder.
2. Copy `Keyboard.js` from `React/Solutions/effect-hook/Keyboard.js` and paste it in the `Exercises/mathificent/src/components/` folder.
3. Open the `Game` component in your editor and import the `Keyboard` component:

```
import Keyboard from './Keyboard';
```

4. Add the `Keyboard` tag to your main `return` statement in the `Game` component. Be sure to pass in `setUserAnswer` as an attribute:

```
<main className="text-center" id="game-container">
  <div className="row border-bottom" style={{fontSize: "1.5em"}}>
    <div className="col px-3 text-left">
      <Score score={score} />
    </div>
    <div className="col px-3 text-right">
      <Timer timeLeft={timeLeft} setTimeLeft={setTimeLeft} />
    </div>
  </div>
  <div className={equationClass} id="equation">
    <Equation question={question} answer={userAnswer} />
  </div>
  <div className="row" id="buttons">
    <div className="col">
      {numberButtons}
      <ClearButton handleClick={setUserAnswer} />
    </div>
  </div>
  <Keyboard setUserAnswer={setUserAnswer} />
</main>
```

5. Test this out by starting a game. You should be able to use your keyboard to answer questions. The **spacebar** and **Enter** keys should work like the **Clear** button, and the **Backspace** key should work to delete the last character added.

8.6. Building and Deploying Your React App

The final phase in making any kind of software is called “deployment.” During this last phase, the following things typically happen:

1. Source code files are compiled from separate modules into one file or bundle.
2. Code and formatting that’s necessary during the development phase but not needed by the end user (such as testing and debugging code and code comments) is removed.
3. Whitespace (spaces, tabs, and line breaks) are removed from the files (this is called “minification”).
4. A distribution file or directory is created. This is the end product of software development.

A compiled React application (also known as a “production build”) uses a special, smaller version of the React library. This version doesn’t contain the debugging and testing code that the full version of React has. Instead, it’s optimized just for making your React application as fast as posssible when someone visits.

Building your React application is very easy if you started with Create React App. All you need to do is to run `npm run build` in your terminal window. Create React App will then automatically go through a process of testing and compiling your application. At the end of the process, you’ll have a new directory named `build` in your Create React App project directory. This is your production build. Feel free to zip it up and send it to a friend. They should be able to unzip and run the application using `npm start` as long as they have Node on their computer.

Conclusion

In this lesson, you have learned to use `useEffect` hooks to control when code runs in a component.