The React Cheatsheet for 2021 (+ Real-World Examples)

React Fundamentals

JSX Elements

- React applications are structured using a syntax called **JSX**. This is the syntax of a basic **JSX element**

```
/*
JSX allows us to write in a syntax almost identical to plain HTML.
As a result, JSX is a powerful tool to structure our applications.
JSX uses all valid HTML tags (i.e. div/span, h1-h6, form/input, img, etc).
*/

<div>Hello React!</div>

/*
Note: this JSX would not be visible because it is needs to be rendered by our application using ReactDOM.render()
*/
```

- JSX is the most common way to structure React applications, but JSX is not required for React

```
/* JSX is a simpler way to use the function React.createElement()
In other words, the following two lines in React are the same: */

<div>Hello React!</div>  // JSX syntax

React.createElement('div', null, 'Hello React!'); // createElement syntax
```

- JSX is not understood by the browser. JSX needs to be compiled to plain JavaScript, which the browser can understand.
- The most commonly used compiler for JSX is called Babel

```
/*
When our project is built to run in the browser, our JSX will be converted by Babel into simple React.createElement() function calls
From this...
*/
const greeting = <div>Hello React!</div>;

/* ...into this: */
"use strict";

const greeting = /*#__PURE__*/React.createElement("div", null, "Hello React!");
```

- JSX differs from HTML in several important ways

```
/*
We can write JSX like plain HTML, but it's actually made using JavaScript functions.
Because JSX is JavaScript, not HTML, there are some differences:

1) Some JSX attributes are named differently than HTML attributes. Why? Because some attribute words are reserved words in JavaScrip

Also, because JSX is JavaScript, attributes that consist of multiple words are written in camelcase:
*/

<div id="header">
  <h1 className="title">Hello React!</h1>
</div>

/*
2) JSX elements that consist of only a single tag (i.e. input, img, br elements) must be closed with a trailing forward slash to be
*/
```

```
<input type="email" /> // <input type="email"> is a syntax error

/*
3) JSX elements that consists of an opening and closing tag (i.e. div, span, button element), must have both or be closed with a tra
*/

<button>Click me</button> // <button> or </button> is a syntax error
<button /> // empty, but also valid
```

- Inline styles can be added to JSX elements using the style attribute
- Styles are updated within an object, not a set of double quotes, as with HTML
- Note that style property names must be also written in camelcase

```
/*
Properties that accept pixel values (like width, height, padding, margin, etc), can use integers instead of strings.
For example: fontSize: 22. Instead of: fontSize: "22px"
*/
<h1 style={{ color: "blue", fontSize: 22, padding: "0.5em 1em" }}>
  Hello React!
</h1>;
```

- JSX elements are JavaScript expressions and can be used as such
- JSX gives us the full power of JavaScript directly within our user interface

```
/*
JSX elements are expressions (resolve to a value) and therefore can be assigned to plain JavaScript variables...
*/
const greeting = <div>Hello React!</div>;

const isNewToReact = true;

// ... or can be displayed conditionally
function sayGreeting() {
  if (isNewToReact) {
    // ... or returned from functions, etc.
    return greeting; // displays: Hello React!
  } else {
    return <div>Hi again, React</div>;
  }
}
```

- JSX allows us to insert (or embed) simple JavaScript expressions using the curly braces syntax

```
const year = 2021;

/* We can insert primitive JS values (i.e. strings, numbers, booleans) in curly braces: {} */
const greeting = <div>Hello React in {year}</div>;

/* We can also insert expressions that resolve to a primitive value: */
const goodbye = <div>Goodbye previous year: {year - 1}</div>;

/* Expressions can also be used for element attributes */
const className = "title";
const title = <h1 className={className}>My title</h1>;

/* Note: trying to insert object values (i.e. objects, arrays, maps) in curly braces will result in an error */
```

- JSX allows us to nest elements within one another, like we would HTML

```
/*
To write JSX on multiple lines, wrap in parentheses: ()
JSX expressions that span multiple lines are called multiline expressions
*/

const greeting = (
```

```
  // div is the parent element
  <div>
    {/* h1 and p are child elements */}
    <h1>Hello!</h1>
    <p>Welcome to React</p>
  </div>
);
/* 'parents' and 'children' are how we describe JSX elements in relation
  to one another, like we would talk about HTML elements */
```

- Comments in JSX are written as multiline JavaScript comments, written between curly braces

```
const greeting = (
  <div>
    {/* This is a single line comment */}
    <h1>Hello!</h1>
    <p>Welcome to React</p>
    {/* This is a
    multiline
    comment */}
  </div>
);
```

- All React apps require three things:

1. `ReactDOM.render()`: used to render (show) our app by mounting it onto an HTML element
2. A JSX element: called a "root node", because it is the root of our application. Meaning, rendering it will render all children within it
3. An HTML (DOM) element: Where the app is inserted within an HTML page. The element is usually a div with an id of "root", located in an index.html file

```
// Packages can be installed locally or brought in through a CDN link (added to head of HTML document)
import React from "react";
import ReactDOM from "react-dom";

// root node (usually a component) is most often called "App"
const App = <h1>Hello React!</h1>;

// ReactDOM.render(root node, HTML element)
ReactDOM.render(App, document.getElementById("root"));
```

Components and Props

- JSX can be grouped together within individual functions called **components**
- There are two types of components in React: **function components** and **class components**
- Component names, for function or class components, are capitalized to distinguish them from plain JavaScript functions that do not return JSX

```
import React from "react";

/*
Function component
Note the capitalized function name: 'Header', not 'header'
*/
function Header() {
  return <h1>Hello React</h1>;
}

// Function components which use an arrow function syntax are also valid
const Header = () => <h1>Hello React</h1>;

/*
Class component
Class components have more boilerplate (note the 'extends' keyword and 'render' method)
*/
class Header extends React.Component {
```

```
  render() {
    return <h1>Hello React</h1>;
  }
}
```

- Components, despite being functions, are not called like ordinary JavaScript functions
- Components are executed by rendering them like we would JSX in our app

```
// Do we call this function component like a normal function?

// No, to execute them and display the JSX they return...
const Header = () => <h1>Hello React</h1>;

// ...we use them as 'custom' JSX elements
ReactDOM.render(<Header />, document.getElementById("root"));
// renders: <h1>Hello React</h1>
```

- The huge benefit of components is their ability to be reused across our apps, wherever we need them
- Since components leverage the power of JavaScript functions, we can logically pass data to them, like we would by passing it one or more arguments

```
/*
The Header and Footer components can be reused in any page in our app.
Components remove the need to rewrite the same JSX multiple times.
*/

// IndexPage component, visible on '/' route of our app
function IndexPage() {
  return (
    <div>
      <Header />
      <Hero />
      <Footer />
    </div>
  );
}

// AboutPage component, visible on the '/about' route
function AboutPage() {
  return (
    <div>
      <Header />
      <About />
      <Testimonials />
      <Footer />
    </div>
  );
}
```

- Data passed to components in JavaScript are called **props**
- Props look identical to attributes on plain JSX/HTML elements, but you can access their values within the component itself
- Props are available in parameters of the component to which they are passed. Props are always included as properties of an object

```
/*
What if we want to pass custom data to our component from a parent component?
For example, to display the user's name in our app header.
*/

const username = "John";

/*
To do so, we add custom 'attributes' to our component called props
We can add many of them as we like and we give them names that suit the data we pass in.
To pass the user's name to the header, we use a prop we appropriately called 'username'
```

```
*/
ReactDOM.render(
  <Header username={username} />,
  document.getElementById("root")
);
// We called this prop 'username', but can use any valid identifier we would give, for example, a JavaScript variable

// props is the object that every component receives as an argument
function Header(props) {
  // the props we make on the component (username)
  // become properties on the props object
  return <h1>Hello {props.username}</h1>;
}
```

- Props must never be directly changed within the child component.
- Another way to say this is that props should never be **mutated**, since props are a plain JavaScript object

```
/*
Components should operate as 'pure' functions.
That is, for every input, we should be able to expect the same output.
This means we cannot mutate the props object, only read from it.
*/

// We cannot modify the props object :
function Header(props) {
  props.username = "Doug";

  return <h1>Hello {props.username}</h1>;
}
/*
But what if we want to modify a prop value that is passed to our component?
That's where we would use state (see the useState section).
*/
```

- The **children** prop is useful if we want to pass elements / components as props to other components

```
// Can we accept React elements (or components) as props?
// Yes, through a special property on the props object called 'children'

function Layout(props) {
  return <div className="container">{props.children}</div>;
}

// The children prop is very useful for when you want the same
// component (such as a Layout component) to wrap all other components:
function IndexPage() {
  return (
    <Layout>
      <Header />
      <Hero />
      <Footer />
    </Layout>
  );
}

// different page, but uses same Layout component (thanks to children prop)
function AboutPage() {
  return (
    <Layout>
      <About />
      <Footer />
    </Layout>
  );
}
```

- Again, since components are JavaScript expressions, we can use them in combination with if-else statements and switch statements to conditionally show content

```
function Header() {
  const isAuthenticated = checkAuth();
```

```
  /* if user is authenticated, show the authenticated app, otherwise, the unauthenticated app */
  if (isAuthenticated) {
    return <AuthenticatedApp />;
  } else {
    /* alternatively, we can drop the else section and provide a simple return, and the conditional will operate in the same way */
    return <UnAuthenticatedApp />;
  }
}
```

- To use conditions within a component's returned JSX, you can use the ternary operator or short-circuiting (&& and || operators)

```
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      {/* if isAuth is true, show nothing. If false, show Logo  */}
      {isAuthenticated || <Logo />}
      {/* if isAuth is true, show AuthenticatedApp. If false, show Login  */}
      {isAuthenticated ? <AuthenticatedApp /> : <LoginScreen />}
      {/* if isAuth is true, show Footer. If false, show nothing */}
      {isAuthenticated && <Footer />}
    </nav>
  );
}
```

- **Fragments** are special components for displaying multiple components without adding an extra element to the DOM
- Fragments are ideal for conditional logic that have multiple adjacent components or elements

```
/*
We can improve the logic in the previous example.
If isAuthenticated is true, how do we display both the AuthenticatedApp and Footer components?
*/
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      <Logo />
      {/*
      We can render both components with a fragment.
      Fragments are very concise: <> </>
    */}
      {isAuthenticated ? (
        <>
          <AuthenticatedApp />
          <Footer />
        </>
      ) : (
        <Login />
      )}
    </nav>
  );
}
/*
  Note: An alternate syntax for fragments is React.Fragment:

  <React.Fragment>
    <AuthenticatedApp />
    <Footer />
  </React.Fragment>
 */
```

Lists and Keys

- Use the **.map()** function to convert lists of data (arrays) into lists of elements

```
const people = ["John", "Bob", "Fred"];
const peopleList = people.map((person) => <p>{person}</p>);
```

- .map() can be used for components as well as plain JSX elements

```
function App() {
  const people = ["John", "Bob", "Fred"];
  // can interpolate returned list of elements in {}
  return (
    <ul>
      {/* we're passing each array element as props to Person */}
      {people.map((person) => (
        <Person name={person} />
      ))}
    </ul>
  );
}

function Person({ name }) {
  // we access the 'name' prop directly using object destructuring
  return <p>This person's name is: {name}</p>;
}
```

- Each React element within a list of elements needs a special **key prop**
- Keys are essential for React to be able to keep track of each element that is being iterated over with the .map() function
- React uses keys to performantly update individual elements when their data changes (instead of re-rendering the entire list)
- Keys need to have unique values to be able to identify each of them according to their key value

```
function App() {
  const people = [
    { id: "Ksy7py", name: "John" },
    { id: "6eAdl9", name: "Bob" },
    { id: "6eAdl9", name: "Fred" },
  ];

  return (
    <ul>
      {/* keys need to be primitive values, ideally a unique string, such as an id */}
      {people.map((person) => (
        <Person key={person.id} name={person.name} />
      ))}
    </ul>
  );
}

/* If you don't have some identifier with your set of data that is a unique
and primitive value, use the second parameter of .map() to get each elements index */

function App() {
  const people = ["John", "Bob", "Fred"];

  return (
    <ul>
      {/* use array element index for key */}
      {people.map((person, i) => (
        <Person key={i} name={person} />
      ))}
    </ul>
  );
}
```

Event Listeners and Handling Events

- Listening for events on JSX elements versus HTML elements differs in a few important ways
- You cannot listen for events on React components; only on JSX elements. Adding a prop called `onClick`, for example, to a React component would just be another property added to the props object

```
/*
The convention for most event handler functions is to prefix them with the word 'handle' and then the action they perform (i.e. hand
*/
function handleToggleTheme() {
  // code to toggle app theme
}

/* In HTML, onclick is all lowercase, plus the event handler includes a set of parentheses after being referenced */
<button onclick="handleToggleTheme()">
  Toggle Theme
</button>

/*
  In JSX, onClick is camelcase, like attributes / props.
  We also pass a reference to the function with curly braces.
  */
<button onClick={handleToggleTheme}>
  Toggle Theme
</button>;
```

- The most essential React events to know are `onClick`, `onChange`, and `onSubmit`
- `onClick` handles click events on JSX elements (namely on buttons)
- `onChange` handles keyboard events (namely a user typing into an input or textarea)
- `onSubmit` handles form submissions from the user

```
function App() {
  function handleInputChange(event) {
    /* When passing the function to an event handler, like onChange we get access to data about the event (an object) */
    const inputText = event.target.value; // text typed into the input
    const inputName = event.target.name; // 'email' from name attribute
  }

  function handleClick(event) {
    /* onClick doesn't usually need event data, but it receives event data as well that we can use */
    console.log("clicked!");
    const eventType = event.type; // "click"
    const eventTarget = event.target; // <button>Submit</button>
  }

  function handleSubmit(event) {
    /*
   When we hit the return button, the form will be submitted, as well as when a button with type="submit" is clicked.
   We call event.preventDefault() to prevent the default form behavior from taking place, which is to send an HTTP request and reloa
  */
    event.preventDefault();
    const formElements = event.target.elements; // access all element within form
    const inputValue = event.target.elements.emailAddress.value; // access the value of the input element with the id "emailAddress"
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        id="emailAddress"
        type="email"
        name="email"
        onChange={handleInputChange}
      />
      <button onClick={handleClick}>Submit</button>
    </form>
  );
}
```

Essential React Hooks

State and useState

- useState give us state in a function component
- **State** allows us to access and update certain values in our components over time

- Local component state is managed by the React hook `useState` which gives us both a state variable and a function that allows us to update it
- When we call `useState` we can give our state a default value by providing it as the first argument when we call `useState`

```
import React from "react";

/*
How do you create a state variable?
Syntax: const [stateVariable] = React.useState(defaultValue);
*/
function App() {
  const [language] = React.useState("JavaScript");
  /*
We use array destructuring to declare state variable.
Like any variable, we declare we can name it what we like (in this case, 'language').
*/

  return <div>I am learning {language}</div>;
}
```

- Note: Any hook in this section is from the React core library and can be imported individually

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

function App() {
  const [language] = useState("javascript");

  return <div>I am learning {language}</div>;
}
```

- useState also gives us a 'setter' function to update the state after it is created

```
function App() {
  /*
  The setter function is always the second destructured value.
  The naming convention for the setter function is to be prefixed with 'set'.
  */
  const [language, setLanguage] = React.useState("javascript");

  return (
    <div>
      <button onClick={() => setLanguage("python")}>Learn Python</button>
      {/*
      Why use an inline arrow function here instead of immediately calling it like so: onClick={setterFn()}?
      If so, setLanguage would be called immediately and not when the button was clicked by the user.
      */}
      <p>I am now learning {language}</p>
    </div>
  );
}

/*
  Note: whenever the setter function is called, the state updates,
  and the App component re-renders to display the new state.
  Whenever state is updated, the component will be re-rendered
  */
```

- useState can be used once or multiple times within a single component
- useState can accept primitive or object values to manage state

```
function App() {
  const [language, setLanguage] = React.useState("python");
  const [yearsExperience, setYearsExperience] = React.useState(0);

  return (
    <div>
      <button onClick={() => setLanguage("javascript")}>
        Change language to JS
```

```
      </button>
      <input
        type="number"
        value={yearsExperience}
        onChange={(event) => setYearsExperience(event.target.value)}
      />
      <p>I am now learning {language}</p>
      <p>I have {yearsExperience} years of experience</p>
    </div>
  );
}
```

- If the new state depends on the previous state, to guarantee the update is done reliably, we can use a function within the setter function that gives us the correct previous state

```
/* We have the option to organize state using whatever is the most appropriate data type, according to the data we're managing */
function App() {
  const [developer, setDeveloper] = React.useState({
    language: "",
    yearsExperience: 0,
  });

  function handleChangeYearsExperience(event) {
    const years = event.target.value;
    /* We must pass in the previous state object we had with the spread operator to spread it all of its properties */
    setDeveloper({ ...developer, yearsExperience: years });
  }

  return (
    <div>
      {/* No need to get previous state here; we are replacing the entire object */}
      <button
        onClick={() =>
          setDeveloper({
            language: "javascript",
            yearsExperience: 0,
          })
        }
      >
        Change language to JS
      </button>
      {/* We can also pass a reference to the function */}
      <input
        type="number"
        value={developer.yearsExperience}
        onChange={handleChangeYearsExperience}
      />
      <p>I am now learning {developer.language}</p>
      <p>I have {developer.yearsExperience} years of experience</p>
    </div>
  );
}
```

- If you are managing multiple primitive values, using useState multiple times is often better than using useState once with an object. You don't have to worry about forgetting to combine the old state with the new state

```
function App() {
  const [developer, setDeveloper] = React.useState({
    language: "",
    yearsExperience: 0,
    isEmployed: false,
  });

  function handleToggleEmployment(event) {
    /* We get the previous state variable's value in the parameters.
     We can name 'prevState' however we like.
  */
    setDeveloper((prevState) => {
      return { ...prevState, isEmployed: !prevState.isEmployed };
      // It is essential to return the new state from this function
    });
  }

  return (
```

```
    <button onClick={handleToggleEmployment}>Toggle Employment Status</button>
  );
}
```

Side effects and useEffect

- useEffect lets us perform side effects in function components. What are side effects?
- **Side effects** are where we need to reach into the outside world. For example, fetching data from an API or working with the DOM
- Side effects are actions that can change our component state in an unpredictable fashion (that have cause 'side effects')
- useEffect accepts a callback function (called the 'effect' function), which will by default run every time there is a re-render
- useEffect runs once our component mounts, which is the right time to perform a side effect in the component lifecycle

```
/* What does our code do? Picks a color from the colors array and makes it the background color */
import React, { useState, useEffect } from "react";

function App() {
  const [colorIndex, setColorIndex] = useState(0);
  const colors = ["blue", "green", "red", "orange"];

  /*
We are performing a 'side effect' since we are working with an API.
We are working with the DOM, a browser API outside of React.
*/
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  });
  /* Whenever state is updated, App re-renders and useEffect runs */

  function handleChangeColor() {
    /* This code may look complex, but all it does is go to the next color in the 'colors' array, and if it is on the last color, go
    const nextIndex = colorIndex + 1 === colors.length ? 0 : colorIndex + 1;
    setColorIndex(nextIndex);
  }

  return <button onClick={handleChangeColor}>Change background color</button>;
}
```

- To avoid executing the effect callback after each render, we provide a second argument, an empty array

```
function App() {
  /*
  With an empty array, our button doesn't work no matter how many times we click it...
  The background color is only set once, when the component first mounts.
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  }, []);

  /*
  How do we not have the effect function run for every state update  but still have it work whenever the button is clicked?
  */

  return <button onClick={handleChangeIndex}>Change background color</button>;
}
```

- useEffect lets us conditionally perform effects with the dependencies array
- The **dependencies array** is the second argument and if any one of the values in the array changes, the effect function runs again

```
function App() {
  const [colorIndex, setColorIndex] = React.useState(0);
```

```
  const colors = ["blue", "green", "red", "orange"];

  /*
  Let's add colorIndex to our dependencies array
  When colorIndex changes, useEffect will execute the effect function again
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
    /*
    When we use useEffect, we must think about what state values
    we want our side effect to sync with
  */
  }, [colorIndex]);

  function handleChangeIndex() {
    const next = colorIndex + 1 === colors.length ? 0 : colorIndex + 1;
    setColorIndex(next);
  }

  return <button onClick={handleChangeIndex}>Change background color</button>;
}
```

- useEffect lets us unsubscribe from certain effects by returning a function at the end

```
function MouseTracker() {
  const [mousePosition, setMousePosition] = useState({ x: 0, y: 0 });

  React.useEffect(() => {
    // .addEventListener() sets up an active listener...
    window.addEventListener("mousemove", handleMouseMove);

    /* ...So when we navigate away from this page, it needs to be
     removed to stop listening. Otherwise, it will try to set
     state in a component that doesn't exist (causing an error)

   We unsubscribe any subscriptions / listeners w/ this 'cleanup function')
   */
    return () => {
      window.removeEventListener("mousemove", handleMouseMove);
    };
  }, []);

  function handleMouseMove(event) {
    setMousePosition({
      x: event.pageX,
      y: event.pageY,
    });
  }

  return (
    <div>
      <h1>The current mouse position is:</h1>
      <p>
        X: {mousePosition.x}, Y: {mousePosition.y}
      </p>
    </div>
  );
}
```

- useEffect is the hook to use when you want to make an HTTP request (namely, a GET request when the component mounts)
- Note that handling promises with the more concise async/await syntax requires creating a separate function (Why? The effect callback function cannot be async)

```
const endpoint = "https://api.github.com/users/reedbarger";

// Using .then() callback functions to resolve promise
function App() {
  const [user, setUser] = React.useState(null);

  React.useEffect(() => {
    fetch(endpoint)
      .then((response) => response.json())
```

```
      .then((data) => setUser(data));
  }, []);
}

// Using async / await syntax to resolve promise:
function App() {
  const [user, setUser] = React.useState(null);
  // cannot make useEffect callback function async
  React.useEffect(() => {
    getUser();
  }, []);

  // We must apply async keyword to a separate function
  async function getUser() {
    const response = await fetch(endpoint);
    const data = await response.json();
    setUser(data);
  }
}
```

Refs and useRef

- Refs are a special attribute that are available on all React components. They allow us to create a reference to a given element / component when the component mounts
- useRef allows us to easily use React refs
- We call useRef (at top of component) and attach the returned value to the element's ref attribute to refer to it
- Once we create a reference, we use the current property to modify (mutate) the element's properties or can call any available methods on that element (like .focus() to focus an input)

```
function App() {
  const [query, setQuery] = React.useState("react hooks");
  /* We can pass useRef a default value.
   We don't need it here, so we pass in null to reference an empty object
  */
  const searchInput = useRef(null);

  function handleClearSearch() {
    /*
    .current references the input element upon mount
    useRef can store basically any value in its .current property
  */
    searchInput.current.value = "";
    searchInput.current.focus();
  }

  return (
    <form>
      <input
        type="text"
        onChange={(event) => setQuery(event.target.value)}
        ref={searchInput}
      />
      <button type="submit">Search</button>
      <button type="button" onClick={handleClearSearch}>
        Clear
      </button>
    </form>
  );
}
```

Hooks and Performance

Preventing Re-renders and React.memo

- React.memo is a function that allows us to optimize the way our components are rendered
- In particular, React.memo performs a process called **memoization** that helps us prevent our components from re-rendering when they do not need to be (see React.useMemo for more complete definition of memoization)

- React.memo helps most with preventing lists of components from being re-rendered when their parent components re-render

```
/*
In the following application, we are keeping track of our programming skills. We can create new skills using an input, they are adde
*/

function App() {
  const [skill, setSkill] = React.useState("");
  const [skills, setSkills] = React.useState(["HTML", "CSS", "JavaScript"]);

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill));
  }

  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} />
    </>
  );
}

/* But the problem, if you run this code yourself, is that when we type into the input, because the parent component of SkillList (A

/* However, once we wrap the SkillList component in React.memo (which is a higher-order function, meaning it accepts a function as a
const SkillList = React.memo(({ skills }) => {
  console.log("rerendering");
  return (
    <ul>
      {skills.map((skill, i) => (
        <li key={i}>{skill}</li>
      ))}
    </ul>
  );
});

export default App;
```

Callback functions and useCallback

- useCallback is a hook that is used for improving our component performance
- **Callback functions** are the name of functions that are "called back" within a parent component.
- The most common usage is to have a parent component with a state variable, but you want to update that state from a child component. What do you do? You pass down a callback function to the child from the parent. That allows us to update state in the parent component.
- useCallback functions in a similar way as React.memo. It memoizes callback functions, so it is not recreated on every re-render. Using useCallback correctly can improve the performance of our app

```
/* Let's keep the exact same App as above with React.memo, but add one small feature. Let's make it possible to delete a skill when

function App() {
  const [skill, setSkill] = React.useState("");
  const [skills, setSkills] = React.useState(["HTML", "CSS", "JavaScript"]);

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill));
  }

  function handleRemoveSkill(skill) {
    setSkills(skills.filter((s) => s !== skill));
```

```
  }

  /* Next, we pass handleRemoveSkill down as a prop, or since this is a function, as a callback function to be used within SkillList
  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} handleRemoveSkill={handleRemoveSkill} />
    </>
  );
}

/* When we try typing in the input again, we see rerendering in the console every time we type. Our memoization from React.memo is b

  What is happening is the handleRemoveSkill callback function is being recreated everytime App is rerendered, causing all children

  To fix our app, replace handleRemoveSkill with:

  const handleRemoveSkill = React.useCallback((skill) => {
  setSkills(skills.filter(s => s !== skill))
  }, [skills])

  Try it yourself!
  */
const SkillList = React.memo(({ skills, handleRemoveSkill }) => {
  console.log("rerendering");
  return (
    <ul>
      {skills.map((skill) => (
        <li key={skill} onClick={() => handleRemoveSkill(skill)}>
          {skill}
        </li>
      ))}
    </ul>
  );
});

export default App;
```

Memoization and useMemo

- useMemo is very similar to useCallback and is for improving performance, but instead of being for callbacks, it is for storing the results of expensive calculations
- useMemo allows us to **memoize**, or remember the result of expensive calculations when they have already been made for certain inputs.
- Memoization means that if a calculation has been done before with a given input, there's no need to do it again, because we already have the stored result of that operation.
- useMemo returns a value from the computation, which is then stored in a variable

```
/* Building upon our skills app, let's add a feature to search through our available skills through an additional search input. We c
 */

function App() {
  const [skill, setSkill] = React.useState("");
  const [skills, setSkills] = React.useState(["HTML", "CSS", "JavaScript"]);

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill));
  }

  const handleRemoveSkill = React.useCallback(
    (skill) => {
      setSkills(skills.filter((s) => s !== skill));
    },
    [skills]
  );

  return (
```

```
    <>
      <SearchSkills skills={skills} />
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} handleRemoveSkill={handleRemoveSkill} />
    </>
  );
}

// /* Let's imagine we have a list of thousands of skills that we want to search through. How do we performantly find and show the s
function SearchSkills() {
  const [searchTerm, setSearchTerm] = React.useState("");

  /* We use React.useMemo to memoize (remember) the returned value from our search operation and only run when it the searchTerm cha
  const searchResults = React.useMemo(() => {
    return skills.filter((s) => s.includes(searchTerm));
  }, [searchTerm]);

  function handleSearchInput(event) {
    setSearchTerm(event.target.value);
  }

  return (
    <>
      <input onChange={handleSearchInput} />
      <ul>
        {searchResults.map((result, i) => (
          <li key={i}>{result}</li>
        ))}
      </ul>
    </>
  );
}

export default App;
```

Advanced React Hooks

Context and useContext

- In React, we want to avoid the following problem of creating multiple props to pass data down two or more levels from a parent component

```
/*
React Context helps us avoid creating multiple duplicate props.
This pattern is also called props drilling.
*/

/* In this app, we want to pass the user data down to the Header component, but it first needs to go through a Main component which
function App() {
  const [user] = React.useState({ name: "Fred" });

  return (
    // First 'user' prop
    <Main user={user} />
  );
}

const Main = ({ user }) => (
  <>
    {/* Second 'user' prop */}
    <Header user={user} />
    <div>Main app content...</div>
  </>
);

const Header = ({ user }) => <header>Welcome, {user.name}!</header>;
```

- Context is helpful for passing props down multiple levels of child components from a parent component

```
/*
Here is the previous example rewritten with Context.
First we create context, where we can pass in default values
```

```
We call this 'UserContext' because we're passing down user data
*/
const UserContext = React.createContext();

function App() {
  const [user] = React.useState({ name: "Fred" });

  return (
    /*
  We wrap the parent component with the Provider property
  We pass data down the component tree on the value prop
 */
    <UserContext.Provider value={user}>
      <Main />
    </UserContext.Provider>
  );
}

const Main = () => (
  <>
    <Header />
    <div>Main app content</div>
  </>
);

/*
We can remove the two 'user' props. Instead, we can just use the Consumer property to consume the data where we need it
*/
const Header = () => (
  /* We use a pattern called render props to get access to the data */
  <UserContext.Consumer>
    {(user) => <header>Welcome, {user.name}!</header>}
  </UserContext.Consumer>
);
```

- The useContext hook can remove this unusual-looking render props pattern to consume context in any function component that is a child of the Provider

```
function Header() {
  /* We pass in the entire context object to consume it and we can remove the Consumer tags */
  const user = React.useContext(UserContext);

  return <header>Welcome, {user.name}!</header>;
}
```

Reducers and useReducer

- Reducers are simple, predictable (pure) functions that take a previous state object and an action object and return a new state object.

```
/* This reducer manages user state in our app: */

function userReducer(state, action) {
  /* Reducers often use a switch statement to update state in one way or another based on the action's type property */

  switch (action.type) {
    /* If action.type has the string 'LOGIN' on it, we get data from the payload object on action */
    case "LOGIN":
      return {
        username: action.payload.username,
        email: action.payload.email,
        isAuth: true,
      };
    case "SIGNOUT":
      return {
        username: "",
        isAuth: false,
      };
    default:
      /* If no case matches the action received, return the previous state */
      return state;
  }
}
```

- Reducers are a powerful pattern for managing state that is used in the popular state management library Redux (commonly used with React)
- Reducers can be used in React with the useReducer hook in order to manage state across our app, as compared to useState (which is for local component state)
- useReducer can be paired with useContext to manage data and pass it around components easily
- useReducer + useContext can be an entire state management system for our apps

```
const initialState = { username: "", isAuth: false };

function reducer(state, action) {
  switch (action.type) {
    case "LOGIN":
      return { username: action.payload.username, isAuth: true };
    case "SIGNOUT":
      // could also spread in initialState here
      return { username: "", isAuth: false };
    default:
      return state;
  }
}

function App() {
  // useReducer requires a reducer function to use and an initialState
  const [state, dispatch] = useReducer(reducer, initialState);
  // we get the current result of the reducer on 'state'

  // we use dispatch to 'dispatch' actions, to run our reducer
  // with the data it needs (the action object)
  function handleLogin() {
    dispatch({ type: "LOGIN", payload: { username: "Ted" } });
  }

  function handleSignout() {
    dispatch({ type: "SIGNOUT" });
  }

  return (
    <>
      Current user: {state.username}, isAuthenticated: {state.isAuth}
      <button onClick={handleLogin}>Login</button>
      <button onClick={handleSignout}>Signout</button>
    </>
  );
}
```

Writing custom hooks

- Hooks were created to easily reuse behavior between components, similar to how components were created to reuse structure across our application
- Hooks enable us to add custom functionality to our apps that suit our needs and can be combined with all the existing hooks that we've covered
- Hooks can also be included in third-party libraries for the sake of all React developers. There are many great React libraries that provide custom hooks such as `@apollo/client`, `react-query`, `swr` and more.

```
/* Here is a custom React hook called useWindowSize that I wrote in order to calculate the window size (width and height) of any com

import React from "react";

export default function useWindowSize() {
  const isSSR = typeof window !== "undefined";
  const [windowSize, setWindowSize] = React.useState({
    width: isSSR ? 1200 : window.innerWidth,
    height: isSSR ? 800 : window.innerHeight,
  });

  function changeWindowSize() {
    setWindowSize({ width: window.innerWidth, height: window.innerHeight });
```

```
  }

  React.useEffect(() => {
    window.addEventListener("resize", changeWindowSize);

    return () => {
      window.removeEventListener("resize", changeWindowSize);
    };
  }, []);

  return windowSize;
}

/* To use the hook, we just need to import it where we need, call it, and use the width wherever we want to hide or show certain ele

// components/Header.js

import React from "react";
import useWindowSize from "../utils/useWindowSize";

function Header() {
  const { width } = useWindowSize();

  return (
    <div>
      {/* visible only when window greater than 500px */}
      {width > 500 && <>Greater than 500px!</>}
      {/* visible at any window size */}
      <p>I'm always visible</p>
    </div>
  );
}
```

Rules of hooks

- There are two essential rules of using React hooks that we cannot violate for them to work properly:
- Hooks can only be used within function components (not plain JavaScript functions or class components)
- Hooks can only be called at the top of components (they cannot be in conditionals, loops, or nested functions)