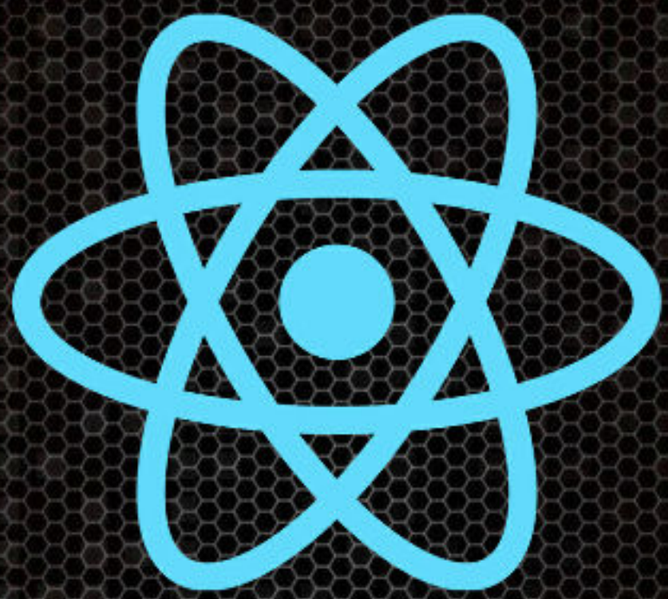
# LEARN REACT



## BUILDING DYNAMIC WEB APPS WITH REACT

BY JORDAN SICKLER

# Table of Contents

FETCHING DATA

RULES OF HOOKS

SEPARATING HOOKS FOR SEPARATE EFFECTS

REVIEW

GETTING STARTED WITH STYLING IN REACT

DIFFERENT WAYS TO STYLE REACT COMPONENTS

MULTIPLE STYLESHEETS

REVIEW

REACT FORMS MADE SIMPLE

## What's the Deal with React?

Developed by engineers at Facebook, React.js stands out as a dynamic JavaScript library. Let's delve into the key motives driving developers to embrace React:

React means speed. Applications crafted using React exhibit remarkable proficiency in handling intricate updates while maintaining a swift and responsive user experience.

Modularity defines React. Instead of grappling with extensive, unwieldy code files, you can create numerous smaller, reusable files. React's modular approach serves as an elegant antidote to JavaScript's challenges in terms of maintainability.

Scalability is React's forte. It excels in handling large-scale programs that present copious amounts of fluctuating data.

Flexibility reigns with React. Its utility extends beyond web app development, accommodating diverse projects. The uncharted potential of React continues to be a space for exploration.

React boasts popularity. While this factor may not be intrinsically tied to React's intrinsic quality, the reality is that proficiency in React enhances your employability.

If you're a newcomer to the realm of React, this course is tailored for you—prior knowledge of React is not a prerequisite. We embark on this journey from the very basics, progressing gradually. By the course's culmination, you'll be poised to code proficiently in React, armed with a genuine comprehension of your actions.

---

## Hello World

Let's Break Down the "Hello World" Mystery

Imagine this piece of secret code:

```
const h1 = <h1>Hello world</h1>;
```

It might seem like a mix of strange stuff, right? Is it talking in JavaScript? Or is it chatting in HTML? Or is it something completely different?

Okay, here's the deal: even though it starts with `const` and ends with a `;`, which are like JavaScript things, it's not totally JavaScript. So, if you tried to use it in a regular webpage, it wouldn't play nice.

But wait, there's more! The part that goes `<h1>Hello world</h1>` looks exactly like HTML – you know, the stuff that makes websites look pretty. But if you tried to use just that part in a usual JavaScript program, it would be a no-go.

So, what's the dealio here?

**Exercise: "Hello World" with React**

In this interactive coding exercise, you'll get hands-on experience with React and JSX by creating a simple "Hello World" application. You'll learn how to set up a basic React app and render a "Hello world" message on a webpage. Let's get started!

**Instructions: **

1. Create a new directory for your React project. You can name it "hello-world-app" or anything you like.

2. Open your terminal and navigate to the project directory you just created using the `cd` command. For example:

```
cd hello-world-app
```

3. Inside your project directory, initialize a new React app by running the following command:

```
npx create-react-app hello-world
```

4. This command sets up a new React project named "hello-world."

5. Once the setup is complete, navigate to the "hello-world" directory:

```
cd hello-world
```

6. Open the "src" folder and locate the "App.js" file. This is where we'll write our "Hello world" code.

7. In the "App.js" file, you'll see some existing code. Remove everything inside the `return` statement of the

`App` component's `render` method. Your code should look like this:

```
return (
  <div className="App">
    {/* Your "Hello world" message will go here */}
  </div>
);
```

7. Now, let's add a "Hello world" message using JSX. Replace the comment inside the `div` with the following JSX code:

```
<h1>Hello world</h1>
```

And here is your updated return statement:

```
return (
  <div className="App">
    <h1>Hello world</h1>
  </div>
);
```

8. Save the "App.js" file.

9. Open your terminal and make sure you're still in the "hello-world" project directory.

10. Start the development server by running the following command:

```
npm start
```

11. React will start the development server, and your default web browser should open, displaying your "Hello world" message.

12. Congratulations! You've just created a simple "Hello World" application using React and JSX.

**Bonus Challenge (Optional):** Try customizing your "Hello world" message or adding additional elements to your React app. Explore React's capabilities and have fun!

**Hint:** If you want to stop the development server at any time, you can press `Ctrl + C` in your terminal.

**Note:** This exercise assumes that you have Node.js and npm (Node Package Manager) installed on your computer. If you haven't already installed them, please do so before starting this exercise.

## The Secret's Out!

Look at that line of code you made again. Does it fit better in a file for JavaScript, a file for HTML, or some other secret spot?

Guess what? It's a match for a JavaScript file! Even though it might seem like it, the code you wrote doesn't have any real HTML in it.

And that part that seems like HTML, <h1>Hello world</h1>, is actually a thing called JSX.

## What's JSX, Anyway?

JSX is like a special add-on language for JavaScript. It was designed to work with React. When you see JSX code, it seems a lot like HTML, the stuff that makes websites look nice.

But what's the deal with "syntax extension"?

Here's the scoop: JSX isn't the same as regular JavaScript. The web browsers don't understand it!

If a JavaScript file has JSX in it, it needs a makeover. Before the file goes to a web browser, a special JSX translator will change all that JSX stuff into regular JavaScript.

Meet JSX Building Blocks

So, in the world of JSX, we have these things called JSX elements. Think of them as the building blocks.

Here's an example of one:

```
<h1>Hello world</h1>
```

This JSX element might trick you because it looks exactly like HTML – the stuff that makes websites pretty. But here's the twist: you'd find it hanging out in a JavaScript file, not an HTML one.

## Getting Friendly with JSX Details

Hey there, let's talk about something called "attributes" in JSX. It's kinda like when you add special stuff to HTML things.

Here's how it works: an attribute in JSX looks like this in a sort of HTML way – a name, then an equals sign, and then a value in quotes:

```
my-attribute-name="my-attribute-value"
```

Now, check out these examples of JSX elements with attributes:

```
<a href='http://www.example.com'>Welcome to the Web</a>;
const title = <h1 id='title'>Introduction to React.js: Part I</h1>;
const panda = <img src='img/panda.jpg' alt='' width='50px' height='50px'>;
```

And guess what? You can have lots of attributes on one JSX thing, just like in HTML:

```
const panda = <img src='img/panda.jpg' alt='' width='50px' height='50px'>;
```

So, now you're on your way to being a JSX attributes expert!

## Let's Talk About Putting JSX Inside Other JSX

Alright, imagine you can put one thing inside another thing – just like how you do it in HTML. Well, you can do that with JSX too.

For example, you can stick a JSX <h1> thing inside a JSX <a> thing like this:

```
const panda = <img src='img/panda.jpg' alt='' width='50px' height='50px'>;
```

But let's keep it neat. You can use line breaks and spaces to make it look better:

```
<a href="https://www.example.com">
  <h1>
    Click me!
  </h1>
</a>
```

Now, here's a little trick: if your JSX thing takes up more than one line, wrap it in these brackets: ( ). It might seem odd at first, but it's just a thing you get used to:

```jsx
<a href="https://www.example.com">
  <h1>
    Click me!
  </h1>
</a>
```

Oh, and guess what? You can save these nested JSX things as names, and do all sorts of cool stuff with them, just like you do with normal JSX things. Here's an example:

```jsx
(
  <a href="https://www.example.com">
    <h1>
      Click me!
    </h1>
  </a>
)
```

And there you go! You're becoming a pro at putting JSX inside other JSX stuff!

## The Big Rule About JSX Main Parts

Here's something important: when you're playing with JSX, it likes to have just one main thing going on. For example, this is fine:

```
const paragraphs = (
  <div id="i-am-the-outermost-element">
    <p>I am a paragraph.</p>
    <p>I, too, am a paragraph.</p>
  </div>
);
```

But this won't work:

```
const paragraphs = (
  <p>I am a paragraph.</p>
  <p>I, too, am a paragraph.</p>
);
```

You see, the first start thingy and the last end thingy in a JSX expression need to belong to the same big thing!

It's pretty easy to forget this rule and end up with confusing errors.

So, if you ever see that your JSX expression has lots of starting and ending things, just wrap it all in a  thing. That usually fixes the problem!

Let's Make Your JSX Show Up!

Now that you've got the hang of creating JSX elements, let's put them on the screen. This is called "rendering". Here's how it goes:

```
const paragraphs = (
  <p>I am a paragraph.</p>
  <p>I, too, am a paragraph.</p>
);
```

See, we've got this "container" thing that's like a special spot on the webpage. It's where our stuff will go. Then, we make a "root" using the "createRoot()" tool. This root knows where to put our JSX.

Last step, we tell the root to "render" this cool thing: <h1>Hello world</h1>. And bam! "Hello world" will show up on the screen.

## Rendering JSX Explained

## You're doing it! Your JSX is coming to life!

Let's Break Down How Stuff Appears on the Screen

So, check this out, we're gonna look at some code you just did:

```
const container = document.getElementById('app');
const root = createRoot(container);
root.render(<h1>Hello world</h1>);
```

But let's start with the basics: React needs two things to show stuff on the screen – what to show and where to put it.

So, first up:

```
const container = document.getElementById('app');
```

This line uses the "document" thing that's all about your web page. Grabs the part of your webpage that has an "id" called "app".

```
const root = createRoot(container);
```

Keeps that part handy in a thing called "container". Now, moving on:

In this line, we're using a special tool called "createRoot()" from React's magic toolkit. This tool makes a special place (called a root) in the container. It's where we'll put our stuff.

Lastly, we've got:

```
root.render(<h1>Hello world</h1>);
```

Here's where the magic happens! We're telling the root, "Hey, show this stuff: <h1>Hello world</h1>". And guess what? It shows "Hello world" in a big bold way on the screen.

So, remember, it's like telling React what to show ("Hello world") and where to put it (in the root). And there you go, you're a JSX master!

## Passing a Variable to render()

So, remember when we learned about making a React root with "createRoot()" and showing JSX with "render()"? Well, here's the deal: the thing we put in "render()" doesn't have to be fancy JSX.

Take a look:

```jsx
const toDoList = (
  <ol>
    <li>Learn React</li>
    <li>Become a Developer</li>
  </ol>
);

const container = document.getElementById('app');
const root = createRoot(container);
root.render(toDoList);
```

In this example, we have a list saved in a thing called "toDoList". It's JSX, but it's just a list, nothing too crazy.

Now, we take that "toDoList" and put it in the "render()" part. You see, "render()" is pretty chill – it can handle JSX or things that turn into JSX.

So, the bottom line is, you can show your JSX or even stuff that turns into JSX using "render()". Cool, right? You're on your way to being a JSX superstar!

## Virtual Dom

So, here's a really neat thing about how React works: when you use the "render()" thing from React's root, it's super smart. It only changes the parts of the webpage that really need to change.

```
const hello = <h1>Hello world</h1>;
// This makes "Hello world" show up:
root.render(hello, document.getElementById('app'));
// But this won't really do anything:
root.render(hello, document.getElementById('app'));
```

Imagine this:

Here's why it's cool: React doesn't waste time redoing everything when it doesn't need to. If you show the same thing again, React's like, "Nah, no need to do anything extra."

This is actually a big deal! It's a big part of why React is so popular. It's like magic, and it's all thanks to something called the "virtual DOM". It's like a clone of your webpage that React uses to figure out what needs to change. And that's why React is amazing!

## Let's Sum It Up!

Awesome job! You've taken your first steps into the world of React, and that's a big deal. Here's a quick recap of what we covered:

React is a front-end framework that's modular, scalable, flexible, and super popular.

JSX is like a fancy way to mix JavaScript with HTML. You can store JSX in all sorts of places like variables, objects, and arrays.

JSX can have attributes, and you can stack JSX inside other JSX, just like you do with HTML.

But here's the rule: JSX needs to have one big outer element, and you can stuff more elements inside it.

You can use "createRoot()" from "react-dom/client" to make a React starting point at a certain spot on your webpage.

Then, you can use "render()" to make your JSX show up on the screen. And the cool part is, it only changes what really needs changing, thanks to the virtual DOM.

So, keep going! In your React journey, you'll learn even cooler things about JSX, tackle some common problems, and become a JSX superstar!

## React's Secret: The Virtual DOM

Okay, let's dive into something really cool about React: the virtual DOM. It's like a super-smart way of making sure we don't do unnecessary work when we're changing stuff on a webpage.

Here's the deal:

The Problem:

So, when we work with webpages, we often need to change things, right? Like making buttons work or checking off items in a list. But here's the issue: changing stuff on a webpage can be slow, especially if we do it a lot.

The Solution - The Virtual DOM:

This is where the virtual DOM comes in. In React, for every thing on the webpage (like a button or a list item), there's a "virtual" copy of it. This virtual thing is just a lightweight version, like a blueprint.

Now, here's the magic: when we want to change something on the webpage, instead of directly changing it (which is slow), we first change the virtual copy. Think of it like editing a blueprint instead of rearranging furniture in a real house.

Why It Helps:

When we make changes, React updates all these virtual copies. It might sound like a lot of work, but it's super fast because it's not actually changing what you see on the screen yet.

Then, React is like a detective. It compares these new virtual copies with how they looked before we made changes. It figures out exactly what's different. This detective work is called "diffing."

Once React knows what's changed, it only updates those things on the actual webpage. So, if you checked off one item in a list, React's smart enough to only update that one item, not the whole list.

Why It Matters:

This is a game-changer! React only updates the parts of the webpage that need changing. It's like doing just the necessary work, not redoing everything.

And that's why React is known for being super fast and efficient. It's all thanks to this virtual DOM trick. In short, here's what happens when you make changes in React:

The virtual copy of everything gets updated.

React compares the new virtual stuff with how it was before. React figures out what's different.

Only the things that changed get updated on the actual webpage. The changes on the webpage make your screen change.

That's React's secret sauce: the virtual DOM!

## Let's Talk About class vs. className

Alright, we're diving into some advanced JSX stuff here. Don't worry; I'll break it down for you.

So, when we're writing in JSX, it's kind of like writing in HTML. But there's a little quirk to keep an eye on, and it involves the word "class."

In regular HTML, it's totally normal to use "class" as an attribute, like this:

```
<h1 className="big">Title</h1>
```

But in JSX, you can't use "class" like that! Nope, you need to use "className" instead:

```
<h1 className="big">Title</h1>
```

Why the change? Well, JSX gets turned into JavaScript when it's used. And guess what? "class" is a special word in JavaScript, so we can't use it the same way we do in HTML.

But don't worry too much because when JSX is finally shown on the webpage, those "className" things you wrote turn into regular "class" attributes. It's like a little translation magic.

So, remember: in JSX, it's "className" when you're writing, but it becomes "class" when you see it on the screen. Easy peasy!

## Self-Closing Tags

Hey there! Let's talk about self-closing tags in JSX, and I promise to keep it simple.

First off, what's a self-closing tag? Well, in regular HTML, most things have two parts, like an opening and a closing tag. For example,  opens, and  closes. Easy, right?

But, some things in HTML, like <img> or <input>, are like the cool loners; they only need one tag. This one tag isn't really an opener or closer; it's called a self-closing tag.

In regular HTML, you can write a self-closing tag with or without a slash before the final angle bracket, like this:

```
// Totally cool in HTML with a slash:
<br />
// Also cool in HTML, no slash needed:
<br>
```

Now, here's the catch: in JSX, you must use the slash. If you forget it, you'll get an error. So, in JSX, always do this:

```
// Perfectly fine in JSX:
<br />
// A big no-no in JSX:
<br>
```

See? JSX likes things tidy and needs that slash. Keep that in mind, and you're good to go!

## Let's Decode Curly Braces in JSX

Alright, let's clear up something that might seem a bit confusing.

In the last thing we tried, instead of doing math and giving us 2 + 3 = 5, it just showed "2 + 3" as if it were a bunch of letters. Weird, right?

Well, here's the deal: when you have numbers or stuff between JSX tags like <h1> and </h1>, JSX doesn't treat them like regular math. Nope, it reads them as plain text, kinda like how HTML does.

But sometimes, you want JSX to treat things like real JavaScript, even when they're inside those tags. That's when you bring in the curly braces { }.

So, if you want to say, "Hey, even though I'm in the middle of JSX, treat me like normal JavaScript," you wrap your code with these curly braces, like this:

```
<h1>5</h1>
```

Now, it gets it! So, always remember, when you're doing something special inside JSX, use those magical curly braces to make it work.

## Variables in JSX

Alright, let's talk about something cool. When you mix JavaScript with JSX, it's like they're all in the same club.

```
// First, we make a variable:
const name = 'Gerdo';
```

Here's the deal: you can use variables inside a JSX thing, even if you made those variables outside of the JSX part.

Look at this:

```
// Then, we invite it into our JSX world:
const greeting = <p>Hello, {name}!</p>;
```

See what's happening? We declare a variable called "name" and then use it right inside our JSX thing. It's like they're best pals.

So, remember, when you're in JSX land, your variables can join the party too!

## Using Variables to Set Attributes in JSX

Hey there! Let's dive into something really useful in JSX: using variables to set attributes. Here's a simple example to help you get it:

```
// We've got a variable for the image size:
const sideLength = "200px";

// And here's our image with those attributes set:
const panda = (
  <img
    src="images/panda.jpg"
    alt="panda"
    height={sideLength}
    width={sideLength}
  />
);
```

See how we used the variable "sideLength" to set the "height" and "width" attributes of our image? It's like telling JSX, "Hey, use this variable to set these attributes."

And when you have lots of attributes, like for an object, you can keep it clean and readable by putting each attribute on its own line, like this:

```
// We've got an object with image links:
const pics = {
  panda: "http://bit.ly/1Tqltv5",
  owl: "http://bit.ly/1XGtkM3",
  owlCat: "http://bit.ly/1Upbczi"
};

// And here, we use those links for different images:
const panda = (
  <img
    src={pics.panda}
    alt="Lazy Panda"
  />
);

const owl = (
  <img
    src={pics.owl}
    alt="Unimpressed Owl"
  />
);

const owlCat = (
  <img
    src={pics.owlCat}
    alt="Ghastly Abomination"
  />
);
```

So, remember, in JSX, you can use variables and object properties to set attributes and keep your code looking clean and organized.

## Event Listeners in JSX

Alright, time to chat about something cool: event listeners in JSX. These are like the secret sauce that makes your web pages interactive!

Here's how it works:

Creating an Event Listener:

```
<img onClick={clickAlert} />
```

You make an event listener by giving a JSX element a special superpower. You do this by adding a special attribute to the element. For example:

This says, "Hey, when someone clicks on this image, do something special!" Naming Your Event Listener:

Your event listener's name should be like "onClick" or "onMouseOver." It's the word "on" plus the type of event you're waiting for. You can find a list of supported event names in the React documentation.

Defining the Action:

Your event listener's value should be a function, like this:

```
function clickAlert() {
  alert('You clicked this image!');
}
```

See how it's a function? It's like a set of instructions for what to do when the event happens. Putting It Together:

Finally, you attach the event listener to your element:

```
<img onClick={clickAlert} />
```

Now, when someone clicks that image, it will run the clickAlert function and show an alert message.

Oh, one more thing: in regular HTML, event listener names are in all lowercase, like "onclick" or "on-mouseover." But in JSX, they use camelCase, like "onClick" or "onMouseOver."

So, there you go! Event listeners are how you make your web page come to life.

## JSX Conditionals: When 'if' Doesn't Play Nice

You're doing great! You've got the hang of using curly braces to mix in some JavaScript with your JSX.

Now, there's one important thing to remember: you can't just stick an "if" statement right into a JSX expression. It doesn't work.

```
(
  <h1>
    {
      if (purchase.complete) {
        'Thank you for placing an order!'
      }
    }
  </h1>
)
```

Take a look at this example:

### Uh-oh, that won't fly in JSX!

But hold on, what if you want something to show up in your JSX only under certain conditions? Well, you've got a bunch of tricks up your sleeve. In the upcoming lessons, we'll explore some clever ways to write conditionals in JSX so that your code knows when to do its thing.

### JSX Conditionals: Making If Statements Work

Okay, let's figure out how to use if statements in JSX, even though you can't just shove them right in there. Here's the trick: you write your if statement outside of JSX. It's like doing your homework before you go play.

Check out the "if.js" file. See how the if statement isn't squeezed in between those JSX tags? Instead, it's sitting outside, from line 8 to 20.

Why does this work? Because the "if" and "else" stuff isn't injected into the JSX itself. The if statement does its thing outside, and JSX is none the wiser.

This is a pretty common way to deal with conditionals in JSX. You get the best of both worlds!

## Meet the Ternary Operator

Hey there, it's time to learn about a nifty way to do conditionals in JSX called the "ternary operator." Don't worry, it's not as complicated as it sounds.

Here's how it works: you write something like x ? y : z, where x, y, and z are just regular pieces of JavaScript. When your code runs, it checks if x is either "truthy" or "falsy." If it's truthy, it gives you y. If it's falsy, you get z.

Now, let's see it in action in JSX:

```
const headline = (
  <h1>
    {age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff'}
  </h1>
);
```

In this example, if age is greater than or equal to drinkingAge, then headline becomes <h1>Buy Drink</h1>. If not, it becomes <h1>Do Teen Stuff</h1>.

So, it's like a quick way to make decisions in JSX. Simple, right? The Sneaky && Operator

Alright, we've got one more trick up our sleeves for conditionals in React, and it's called the "&& operator." Don't worry, it's not as tricky as it sounds.

Here's how it rolls: && isn't just for React; you'll see it all over the place in React code.

```
const tasty = (
  <ul>
    <li>Applesauce</li>
    { !baby && <li>Pizza</li> }
    { age > 15 && <li>Brussels Sprouts</li> }
    { age > 20 && <li>Oysters</li> }
    { age > 25 && <li>Grappa</li> }
  </ul>
);
```

The cool thing about && is that it's great when you want to do something sometimes, but other times, you just want to do nothing at all.

In this example, if the thing on the left of the && is true, it shows the JSX on the right. But if it's false, it just skips it, like it's not even there.

So, it's like having a superpower that lets you decide when to show stuff and when to keep it hidden. Pretty cool, right?

.map( and JSX: A Dynamic Duo

Let's dive into something super useful: the .map( method in React. It's like a magic wand for creating lists of stuff!

Here's the deal: if you want to make a list of JSX elements, .map( is your go-to buddy. It might look a bit weird at first, but don't worry, it's pretty simple.

Check this out:

```
const strings = ['Home', 'Shop', 'About Me'];
const listItems = strings.map(string => <li>{string}</li>);
<ul>{listItems}</ul>
```

In this example, we start with an array of strings. Then, we use .map() on that array, and it gives us a brand new array of <li> elements.

Now, here's the fun part: {listItems} on the last line turns into an array because it's the result of .map()! JSX

<li>s don't have to be in an array like this, but they totally can be. Look, you can also do it like this:

```
// This is perfectly fine in JSX:
<ul>
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
</ul>

// And guess what? This is cool too!
const liArray = [
  <li>item 1</li>,
  <li>item 2</li>,
  <li>item 3</li>
];
<ul>{liArray}</ul>
```

<ul>{liArray}</ul>

See? .map() is like your creative tool for making lists of JSX elements. It's your shortcut to awesome!

## Keys in JSX

Alright, let's uncover the secrets of keys in JSX. Think of keys as little tags that help React keep track of lists. They're like nametags for list items, but you don't see them.

Here's how it works:

```
<ul>
  <li key="li-01">Example1</li>
  <li key="li-02">Example2</li>
  <li key="li-03">Example3</li>
</ul>
```

See those "key" things? They're just attributes in JSX. Like an ID, but unique to each list item.

Now, here's the important part: keys don't make anything visible on your page. React uses them behind the scenes to keep lists in order. If you skip using keys when you should, React might mix up your list items, and that's no fun!

So, when do you need keys? Here's the deal:

If your list items need to remember stuff between renders, like whether they're checked off in a to-do list, you need keys. They help items keep their memory.

If your list might change order from one render to the next, you also need keys. Like a list of search results that could shuffle around.

If none of these apply, you're good to go without keys. But when in doubt, just use them to be safe!

## React.createElement: JSX's Silent Partner

Okay, let's get this straight: you can do React without JSX. Most folks use JSX, but it's good to know you have options!

Check out this JSX expression:

```
const h1 = <h1>Hello world</h1>;
```

Now, watch this: you can rewrite it without JSX, like so:

```
const h1 = React.createElement(
  "h1",
  null,
  "Hello world"
);
```

When JSX gets turned into code, it magically becomes this React.createElement() thing. Basically, every JSX tag secretly becomes a call to React.createElement().

Now, we won't dive deep into how React.createElement() works here, but if you're curious, you can find all the details in the React docs. It's like JSX's quiet partner in crime!

React Components: Building Blocks of Awesomeness

Alright, let's break it down. In the world of React, everything revolves around components. But what's a component, you ask?

Think of it as a little helper that does one specific job really well. Most of the time, its job is to show some stuff on the screen and update it when needed.

Take a peek at this code. It's like the recipe for creating and using a React component:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MyComponent() {
  return <h1>Hello world</h1>;
}

ReactDOM.createRoot(
  document.getElementById('app')
).render(<MyComponent />);
```

Don't let it scare you! We'll unravel it step by step. By the end of this lesson, you'll be a pro at building your own React components.

## Importing React: Your First Step in the React World

Remember that cool React component we built earlier? Well, it all began with this magical line:

```
import React from 'react';
```

What this does is create something like a toolbox called "React." Inside this toolbox, you'll find all the tools you need to work with React. You get this toolbox by importing it from the 'react' package, which should already be sitting in your project as a friend.

So, what can you do with this toolbox? Well, tons of stuff! You can use it for things like Hooks (don't worry, we'll get into those later).

For the next few exercises, we'll be bouncing between two files: App.js and index.js. In the world of React, App.js is usually where all the action starts, and index.js is like the front door to your React app.

## Import ReactDOM

Another import we need in addition to React is ReactDOM:

```
import ReactDOM from 'react-dom/client';
```

The methods imported from 'react-dom' interact with the DOM.

The methods imported from 'react' do not deal with the DOM at all. They don't engage directly with anything that isn't part of React.

To clarify: the DOM is used in React applications, but it isn't part of React. After all, the DOM is also used in countless non-React applications. Methods imported from 'react' are only for pure React purposes, such as creating components or writing JSX elements.

## Creating Function Components: React Made Simple

Okay, let's break it down into bite-sized pieces. In React, a component is like a mini-tool that does one specific job, usually showing stuff on the screen and updating it when needed. These components come together like building blocks to make a full-blown React app. It's like making a sandwich: you've got components for the bread, the filling, and all the tasty layers.

Now, there are two ways to create React components. In the past, we used fancy JavaScript classes, but these days, we often use function components. They're the new cool kids on the block, thanks to something called Hooks (we'll dive into those later).

Here's a simple example:

```
import React from 'react';

function MyComponent() {
  return <h1>Hello, I'm a functional React Component!</h1>;
}

export default MyComponent;
```

In this code, we're defining a function called MyComponent. Inside that function, we return something special: a piece of React magic in JSX. This makes a basic React functional component.

And that last line? It's like saying, "Hey, make sure you can use MyComponent elsewhere!" So, even if it seems a bit mysterious, you're on your way to mastering React step by step.

## Naming Your Functional Component: Easy as Pie!

Great! You've created a JavaScript function, and that's how you declare a new functional component.

```
function MyComponent() {
  return <h1>Hello world</h1>;
}
```

But here's the next step: you've got to give your component a name. In the finished example, we called our component "MyComponent":

Now, here's the trick: when you name a function component, it must start with a capital letter, and it's usually done in PascalCase. This might seem like a small detail, but it's super important in React.

Why? Well, React needs to tell the difference between your components and regular HTML tags. So, capitalization is like a secret code that says, "Hey, I'm a React component, not an ordinary tag!" If you start with a lowercase letter, React might get confused and look for an HTML tag instead. And trust me, you don't want that confusion. So, always start with a capital letter, and you'll be golden!

Function component instructions Importing React and ReactDOM:

In the first line of App.js and index.js, there's an import statement that brings in two important things. React is like a magic toolbox with all the tools React needs to work. It's stored in an object.

ReactDOM is another toolbox, but this one helps React work with web pages. It's also stored in an object. Creating a Function Component:

On the third line of App.js, we do something cool. We create what's called a "function component." Think of this as a blueprint for something we want to show on our webpage.

But wait, we can't see it just yet. It's like creating a recipe, but we haven't cooked anything yet. Naming the Component:

When we create a function component, we need to give it a name. But it's not just any name; it has to start with a capital letter.

We follow a rule: start with a capital letter, and then you can use small letters.

This naming is important because it tells React, "Hey, this is a special thing I want to use." Component Body:

Now, here's where the magic happens. Inside our component, there's a pair of curly braces {}.

Between those curly braces, we write instructions for our component. It's like telling it what to show on the webpage.

Even though it might look like a strange mix of HTML and JavaScript, these instructions are what React understands.

So, imagine it like this: we import tools, create a blueprint, give it a name, and then write down what it should look like. Later on, we'll put it on our webpage for everyone to see!

## The return keyword in functional components

When you define a functional component in React, you are essentially creating a function that returns a React element. The return keyword is used to tell React what to render.

For example, the following function defines a component that renders a button:

```
function Button() {
  return <button>Click me</button>;
}
```

The return keyword tells React to render the <button> element. The text inside the <button> element is the text that will be displayed on the button.

The return keyword is always required in functional components. If you do not return a React element, React will not know what to render.

I hope this helps!

Here are some additional things to keep in mind about the return keyword in functional components:

The return value of a functional component can be any valid React element. This includes components, text, images, and other elements.

You can also use the return keyword to return a function. This is useful for creating components that are stateful or that need to perform some kind of calculation before rendering.

The return keyword can be used anywhere in a functional component, but it is most commonly used at the end of the function body.

## Importing and exporting React components

To use a React component, you first need to import it into the file where you want to use it. You can do this using the import keyword. For example, to import the MyComponent component from the App.js file, you would use the following code:

import MyComponent from './App';

Once you have imported the component, you can use it in your code by calling its name. For example, to render the MyComponent component in the DOM, you would use the following code:

<MyComponent />

You can also export components from your own files. To do this, you use the export keyword. For example, to export the MyComponent component from the App.js file, you would use the following code:

export default MyComponent;

The default keyword tells React that this is the default export from the file. This means that you can import the component using the import keyword without specifying a name.

You can also export multiple components from the same file using named exports. To do this, you use the curly braces ({}) around the names of the components you want to export. For example, the following code exports the MyComponent and YourComponent components from the App.js file:

```
export { MyComponent, YourComponent };
```

Once you have exported a component, you can import it from any other file in your project.

## Using and rendering a component

Now that we have defined a function component, we can use it to render something on the browser.

To do this, we use the <MyComponent /> syntax. This is similar to the HTML syntax for rendering an element, but it is actually a React element.

If we want to nest other components inside our component, we can use the <MyComponent>
<OtherComponent /> syntax.

To actually render our component to the browser, we need to use the ReactDOM.createRoot() and ReactDOM.render() methods from the react-dom library. These methods are used to create a React root container and render a React element in it.

The ReactDOM.createRoot() method takes a DOM element as its argument. This DOM element is the place where the React component will be rendered.

The ReactDOM.render() method takes a React element and a DOM element as its arguments. The React element is the component that we want to render, and the DOM element is the place where we want to render it.

Here is an example of how to use the ReactDOM.createRoot() and ReactDOM.render() methods:

const root = ReactDOM.createRoot(document.getElementById('app')); root.render(<MyComponent />);

In this example, we first create a React root container by calling the ReactDOM.createRoot() method. We pass the document.getElementById('app') element as the argument to this method. This means that the React component will be rendered inside the  element in our HTML file.

Next, we call the ReactDOM.render() method to render the <MyComponent /> component in the root container.

Once we have called the ReactDOM.render() method, the <MyComponent /> component will be rendered in the browser.

I hope this helps!

Here are some additional things to keep in mind about using and rendering components: You can use the ReactDOM.createRoot() method only once in your application.

You can call the ReactDOM.render() method multiple times to render different components in the same root container.

You can also use the ReactDOM.render() method to update an existing component.

## Summary

In this lesson, you've delved into a fundamental concept of React: components. Let's quickly review what you've learned:

React applications are built using components.

Components handle the task of displaying parts of the user interface.

To create and use components, you need to import both the react and reactDOM libraries. Components can be defined as JavaScript functions, known as function components.

Function component names should begin with a capital letter, and Pascal case is the convention for naming them.

Function components must return React elements using JSX syntax. You can export and import React components across different files.

You can use a React component by including its name in an HTML-like self-closing tag.

To display a React component, you use .createRoot() to specify a container and then call .render() on it. It might seem like a lot, but understanding components is essential to grasp the power of React!

## Sing Multiline JSX in a Component Made Easy

In this lesson, we'll make JSX and React components less confusing. You'll get the hang of JSX and React components and pick up some cool tips along the way.

Imagine this chunk of fancy web stuff:

```
<blockquote>
  <p>
    The world is full of objects, I do not wish to add any more.
  </p>
  <cite>
    <a target="_blank" href="https://en.wikipedia.org/wiki/Douglas_Huebler">
      Douglas Huebler
    </a>
  </cite>
</blockquote>
```

But when your stuff gets a bit longer and spans multiple lines, you should always put it in these round brackets!

Event Listener and Event Handlers in a Component

Your function components can include event handlers. With event handlers, we can run some code in response to interactions with the interface, such as clicking.

```
function MyComponent() {
  function handleHover() {
    alert('Stop it. Stop hovering.');
  }

  return <div onHover={handleHover}></div>;
}
```

In the above example, the event handler is handleHover(). It is passed as a prop to the JSX element . We'll discuss more on props in a later lesson, but for now, understand that props are information passed to a JSX tag.

The logic for what should happen when the  is hovered on is contained inside the handleHover() function. The function is then passed to the  element.

Event handler functions are defined inside the function component and, by convention, start with the word "handle" followed by the type of event it is handling.

There's a small quirk you should watch out for. Take a look at this line again:

```
return <div onHover={handleHover}></div>;
```

The handleHover() function is passed without the parentheses we would typically see when calling a function. This is because passing it as handleHover indicates it should only be called once the event has happened. Passing it as handleHover() would trigger the function immediately, so be careful!

## Summary

You did it! You've completed the lesson on React components. Let's quickly go over what you've learned:

Nesting JSX: You can put lots of JSX stuff together by enclosing them in a "parent" element in your component.

Using Variables: You can use variables inside React components to inject dynamic data.

Logic in Components: You can do smart thinking by putting your logic above the stuff you want to show on the screen.

Conditional Display: Components can be clever by showing different things based on certain conditions.

Handling Events: You can make components do stuff when something happens, like when you click a button.

You've tackled some pretty complex stuff, so give yourself a pat on the back! Now, you're ready to learn how components fit into the bigger picture of building things with React.

## Components Working Together

In a React app, you can have lots and lots of these things called components.

Now, here's the cool part: each of these components might not seem like a big deal by itself. They're like small puzzle pieces. But when you put them all together, they can create something huge and super interesting.

So, in simple words, React apps are like puzzles made from these components. But what really makes React shine is how these pieces talk to each other.

This lesson is all about how these pieces work together. After you're done with this lesson, you'll know:

How these components can talk to each other.

How we can keep these pieces in different files, all neat and organized.

## Returning another component

In React, each component is responsible for one piece of the interface. As the application grows in complexity, components need to be able to interact with each other to support the features needed.

One way that components can interact with each other is by returning other components. This is useful when you want to reuse a component or when you want to pass data from one component to another.

For example, the PurchaseButton component can be returned by the ItemBox component. This means that the PurchaseButton component will be rendered inside the ItemBox component.

Here is an example of how to return another component:

```
function PurchaseButton() {
  return <button onClick={()=>{alert("Purchase Complete")}}>Purchase</button>
}

function ItemBox() {
  return (
    <div>
      <h1>50% Sale</h1>
      <h2>Item: Small Shirt</h2>
      <PurchaseButton />
    </div>
  );
}
```

In this example, the PurchaseButton component is returned by the ItemBox component. This means that the PurchaseButton component will be rendered inside the ItemBox component.

Here are some additional things to keep in mind about returning components:

You can return any valid React component from a component. This includes components, text, images, and other elements.

You can also return a function from a component. This is useful for creating components that are stateful or that need to perform some kind of calculation before rendering.

The component that you return must be defined before you return it.

## Using a Component in a Simple Way

Remember how we've done this before? In the previous lessons, when we made these little worker components, we usually put them all together in one big file called "App.js." Then, inside this "App" thing, we imported these worker components and showed them on the screen!

It's kinda like playing with building blocks. Imagine we have a special block called "Button," and we want to use it in our main tower, which is "App."

```
import Button from './Button'  // We get our "Button" from another place.

function App() {  // This is our main tower, "App."
  return <Button />;  // We put our "Button" block right here in our tower.
}

export default App;  // And then we say, "Here's our tower, ready to go!"
```

Here's what we do:

export default App; // And then we say, "Here's our tower, ready to go!"

Remember this? This is super handy because it lets us break our tower (component) into smaller pieces (components), and we can use those pieces whenever we want. It's like building with LEGO blocks!

Review

Awesome work! You've finished this short but essential lesson. Let's go over what you've learned one more time:

A React app can have lots of these things called components.

Components can work together by making one component show another component.

When components team up, it's like breaking a big problem into smaller, more manageable pieces. You can keep these pieces in their own files and use them again whenever you need them.

## Understanding Props: Passing Info Between Components

In the world of React, think of components as the tiny building blocks of a bigger picture – your app's interface.

Now, these components often need to chat with each other, share stuff, and work together. That's where "props" come in.

Props are like messages that one component can send to another. These messages can be used to change how a component looks or behaves.

So, when components talk through props, it's like they're sharing secrets that help them work together better. By the end of this lesson, you'll know how to:

Send, receive, and show props.

Use props to make components do different things.

Create special commands for components and share them. Deal with stuff inside a component.

Set up backup plans for props, just in case. Let's dive in!

## Getting to Know a Component's Props

Okay, so, here's the deal: Every component has a thing called "props."

Think of "props" as a container that holds special information about a component.

Now, you've actually seen this in action, maybe without realizing it. Imagine a simple button in regular HTML:

```
<button type="submit" value="Submit"> Submit </button>
```

In this button, we're passing two pieces of info: the "type" and the "value." Depending on the "type" we give it, the button behaves differently. Well, guess what? We can do the same thing with our own components!

Props are like the special details you give a component so it knows how to act.

To get at these props, it's kinda like using ingredients in a recipe. You just say "props" and then use a dot like this:

```
props.name
```

This would get you the "name" info from the props. It's like saying, "Hey, component, tell me your 'name'!"

## Providing Information to a Component using Props

To unlock the power of props, we need to send data to a React component. In the previous task, our Props-Displayer component was just sitting there, looking lonely, because we hadn't told it what to display.

So, how do we do that? Simple! We give the component a special attribute, like this:

```
<Greeting name="Jamel" />
```

Imagine you want to send the message, "We're great!" to another component. Here's how you'd do it:

```
<SloganDisplay message="We're great!" />
```

See how it works? To send data to a component, you need to give a name to the information you're sending. In this case, we used the name "message," but you can choose any name you like.

And if you need to send something that's not just plain text, wrap it in curly braces, like this:

```
<Greeting myInfo={["Astronaut", "Narek", "43"]} />
```

In this example, we're sending a bunch of information to <Greeting />. Any values that aren't plain text are wrapped in curly braces.

## Displaying Information in a Component with Props

Think of props as little notes you pass to a component to tell it how to look or behave.

We've figured out how to send these notes to a component. Now, let's make sure the component actually uses them to show something. Here's the trick: when you're making a function component, you can tell it to expect these notes by putting props in the parentheses:

```
function Button(props) {
  return <button>{props.displayText}</button>;
}
```

In this example, props is like a box that holds all our notes. To use a specific note, you just say props.displayText.

Alternatively, because props is an object, you can make it simpler using curly braces like this:

```
function Button({displayText}) {
  return <button>{displayText}</button>;
}
```

Now, when you use this component, you can send it notes like displayText="Click Me", and it will show them on a button. Easy, right?

## How a Component Shows Props

Imagine props as special instructions you give to a component to make it look or work the way you want.

We've already learned how to give these instructions to a component. Now, let's see how the component actually uses them to show something.

Here's the deal: when you're creating a function component, you need to tell it to expect these instructions by putting props in the parentheses like this:

```
function Button(props) {
  return <button>{props.displayText}</button>;
}
```

In this example, props acts like a container holding all the instructions. To use a specific instruction, you simply say props.displayText.

But wait, there's an easier way! Since props is like a box of labeled instructions, you can directly grab the instruction you want using curly braces like this:

```
function Button({ displayText }) {
  return <button>{displayText}</button>;
}
```

Now, when you use this component, you can give it instructions like displayText="Click Me", and it will use them to display something on a button. It's like magic, but really it's just props!

## Sending and Receiving Info between Components

Imagine props like little packages you send to a component to tell it stuff. First, you send a package to a component like this:

```
<Greeting firstName="Esmerelda" />
```

Then, to see what's inside the package and use it, you do this:

```
return <h1>{props.firstName}</h1>;
```

Now, the most common thing we do with props is sending info from one component to another. In React, props always go in one direction, from the parent component (the boss) to the child component (the helper).

Here's a simple example: In the code below, App is like the boss, and Product is the helper. App sends three pieces of info (name, price, and rating) to Product, and Product listens and uses them.

```
function App() {
    return <Product name="Apple Watch" price={399} rating="4.5/5.0" />;
}
```

Remember, once info is in a component's hands (like Product), it can't be changed. If Product wants new info, it has to ask its boss (App) to send it.

## Rendering different UI based on props

Props are like the data that is passed into a component. You can use props to make decisions about what to render.

For example, the LoginMsg component takes a prop called password. The LoginMsg component can use the password prop to decide whether to render a message that says "Sign In Successful" or a message that says "Sign In Failed".

Here is the code for the LoginMsg component:

```
function LoginMsg(props) {
  if (props.password === 'a-tough-password') {
    return <h2>Sign In Successful.</h2>;
  } else {
    return <h2>Sign In Failed..</h2>;
  }
}
```

In this example, the props.password prop is compared to the string 'a-tough-password'. If the two strings are equal, the LoginMsg component will render a message that says "Sign In Successful". Otherwise, the LoginMsg component will render a message that says "Sign In Failed".

The password prop is not displayed in either case. It is only used to make a decision about what to render.

This is a common technique in React. It is a way to use props to make your components more dynamic and interactive.

## Putting an event handler in a function component

In React, you can pass functions as props. This is especially common with event handler functions.

An event handler function is a function that is called when an event happens. For example, a click event handler function is called when a button is clicked.

To define an event handler function in a function component, you need to create a method on the component. The method name should be the name of the event that you want to handle.

For example, the following code defines a click event handler function:

```
function MyComponent() {
  const handleClick = () => {
    // Do something when the button is clicked.
  };

  return (
    <button onClick={handleClick}>Click me</button>
  );
}
```

The handleClick method is called when the button is clicked. The onClick prop is used to attach the handleClick method to the button.

Receiving an event handler as a prop

In React, you can pass functions as props. This is especially common with event handler functions.

In the previous exercise, you passed a function called talk() from the Talker component to the Button component. The Button component needs to attach the talk() function to the button element as an event handler.

To do this, the Button component needs to define an onClick prop. The value of the onClick prop should be the talk() function.

Here is the code for the Button component:

```
function Button(props) {
  return (
    <button onClick={props.talk}>Click me</button>
  );
}
```

In this code, the onClick prop is used to attach the talk() function to the button element. When the user clicks on the button element, the talk() function will be called.

## Naming event handlers and props

When you pass an event handler as a prop, you need to choose two names: the name of the event handler itself, and the name of the prop that you will use to pass the event handler.

The name of the event handler should be descriptive of the event that it is handling. For example, if the event handler is handling a click event, the name of the event handler could be handleClick.

The name of the prop should be the word on followed by the name of the event. For example, if the event handler is handling a click event, the name of the prop could be onClick.

Here is an example:

```jsx
function MyComponent() {
  function handleClick() {
    // Do something when the button is clicked.
  }

  return (
    <button onClick={handleClick}>Click me</button>
  );
}
```

In this example, the event handler is called handleClick and the prop is called onClick.

The naming convention for event handlers and props is not enforced by React, but it is a good convention to follow. It makes your code more readable and understandable.

Here is a table that summarizes the naming convention:

| Event type | Event handler name | Prop name |
| --- | --- | --- |
| Click | handle Click | onClick |
| Hover | handleHover | onHover |

| | | |
|---|---|---|
| Focus | handleFocus | onFocus |
| Mousedown | handleMouseDown | onMouseDown |
| Mouseup | handleMouseUp | onMouseUp |
| Mouseenter | handleMouseEnter | onMouseEnter |
| Mouseleave | handleMouseLeave | onMouseLeave |
| Change | handleChange | onChange |
| Submit | handleSubmit | onSubmit |

## The children prop

Every component's props object has a property named children. The children prop returns everything that is between a component's opening and closing JSX tags.

For example, the following code:

```
<BigButton>
  I am a child of BigButton.
</BigButton>
```

would have the children prop set to the text "I am a child of BigButton."

The children prop can be used to pass content to a component. This is useful for making components more flexible and reusable. For example, the following code:

```
<BigButton>
  <LilButton />
</BigButton>
```

If a component has more than one child between its JSX tags, then the children prop will return those children in an array. However, if a component has only one child, then the children prop will return the single child, not wrapped in an array.

## Adding a Safety Net for Missing Props

Let's keep it simple! Imagine you have a component called Button, and it expects to receive something called "text" to display inside a button.

But what if nobody sends any "text" to Button? If that happens, Button will just show nothing. Not great, right? It would be better if Button could show a default message.

Well, you can make that happen in three ways!

First, you can add a special thing called defaultProps right inside the component:

```
function Example(props) {
  return <h1>{props.text}</h1>
}

Example.defaultProps = {
  text: 'This is default text',
};
```

Or, you can set the default value right in the function definition like this:

```
function Example({text='This is default text'}) {
   return <h1>{text}</h1>
}
```

Lastly, you can put the default value right inside the function itself:

```
function Example(props) {
  const {text = 'This is default text'} = props;
  return <h1>{text}</h1>
}
```

So, if an <Example /> doesn't get any "text", it will just show "This is default text" instead. But if it does get some "text," it will show that instead. It's like a safety net for your components!

## Props Simplified: A Quick Recap

We've wrapped up our props lesson, and here's a simple summary of what you've learned:

Sending Messages: You send messages (props) to a component by adding special instructions (attributes) to it.

Reading Messages: To understand these instructions, you just say props.instructionName. Displaying Messages: You make the component show these instructions in the user interface (UI). Decision Power: Props help you decide what to display based on the instructions you get.

Action Time: You can make components do things by creating special actions (event handlers) inside them and passing them as instructions (props).

Listening for Actions: When you want a component to listen for specific actions (like a click), you attach these actions to it.

Name Game: There are rules for naming actions and their attributes, but don't get too bogged down in the details.

Inside Info: You can also work with any content placed inside a component using props. children. Safety Net: You can provide backup instructions (default values) for props to prevent empty displays. It might seem like a lot, but don't fret! With more practice, you'll become a props pro in no time.

## Why Bother with Hooks?

Ever wondered how to add special powers to your function component? Or make your app react to changes in data?

Well, that's where React Hooks come in! They're like magic tools for function components.

Think of Hooks as special functions that help us handle the inner workings of our components. They let us manage things like data and what our app should show after it's displayed to the user.

React gives us a bunch of built-in Hooks to work with, like useState(), useEffect(), useContext(), useReducer(), and useRef(). You can check them all out in the React documentation.

In this lesson, we're going to learn how to:

Create a "smart" function component that knows stuff. Use the State Hook to give our component a memory. Set up initial states and change them.

Create special functions for handling events.

Use clever tricks with arrays and objects to make our component even more powerful.

## Updating State in Function Components Made Easy

Let's dive into the world of the State Hook, which is like the go-to tool for building React components. To use it, we import it from React like this:

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

When you use useState(), it gives you an array with two things:

The current state: This is the current value of your data.

The state setter: It's like a magic button that lets you change the data.

You can then use these two things to keep track of data and change it when needed. To use them, you can give them names with array destructuring, like this:

```
const [currentState, setCurrentState] = useState();
```

Let's see a practical example. Here's a function component using the State Hook:

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

function Toggle() {
  const [toggle, setToggle] = useState();

  return (
    <div>
      <p>The toggle is {toggle}</p>
      <button onClick={() => setToggle("On")}>On</button>
      <button onClick={() => setToggle("Off")}>Off</button>
    </div>
  );
}
```

In this example, we have a variable called toggle, and we use setToggle() to change its value. When you click the buttons, it's like telling React, "Hey, update this value!" React then re-renders the component with the new value.

The cool thing is that React keeps track of this value between renders, thanks to useState(). It's like having a built-in memory for your data!

## Use State Setter Outside of JSX

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

Let's understand how to work with changing text in a text input field. Don't worry; we'll keep it simple! First, we import useState from React, like this:

Now, check out this code:

```
export default function EmailTextInput() {
  const [email, setEmail] = useState('');

  const handleChange = (event) => {
    const updatedEmail = event.target.value;
    setEmail(updatedEmail);
  }

  return (
    <input value={email} onChange={handleChange} />
  );
}
```

Here's the breakdown:

We use array destructuring to create a variable called email and a function called setEmail. These come from useState().

email holds the current value of our state (it's like a memory). setEmail is like a special button to change the email value.

We set up an onChange event listener for our input. It's like listening for when someone types. When the user types, it calls a function called handleChange.

Inside handleChange, we get the new email value from the input and use setEmail to update email. Now, it's common to make this even simpler:

```
const handleChange = (event) => setEmail(event.target.value);
```

```
const handleChange = ({ target }) => setEmail(target.value);
```

Or, if you want to get fancy:

All these codes do the same thing, so pick the one you like best. The idea is to separate the code that changes data from the code that handles what's shown in the UI. This makes your code easier to read and work with!

## Updating state with a callback function

In React, state updates are asynchronous. This means that there are some scenarios where parts of your code will run before the state is finished updating.

To avoid this, it is best practice to update a state with a callback function. This ensures that the state is updated before the next part of your code runs.

Here is an example of how to update state with a callback function:

```
const [count, setCount] = useState(0);
const increment = () => setCount(prevCount => prevCount + 1);
```

In this code, the increment() function is called when the button is clicked. The increment() function then calls the setCount() state setter with a callback function.

The callback function takes the previous value of the count state as an argument. The value returned by the callback function is then used as the new value of the count state.

In this case, the callback function simply adds 1 to the previous value of the count state.

You can also just call setCount(count + 1) and it would work the same in this example. However, it is safer to use the callback method because it ensures that the state is updated before the next part of your code runs.

## Arrays in State

Imagine you're building a pizza ordering app in React. To handle the different pizza topping options, you'll want to use JavaScript arrays. Don't worry; it's simpler than it sounds.

Here's a breakdown of the code for your pizza app:

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

// Static array of pizza topping options offered.
const options = ['Bell Pepper', 'Sausage', 'Pepperoni', 'Pineapple'];

export default function PersonalPizza() {
  // Initialize the selected toppings as an empty array.
  const [selected, setSelected] = useState([]);

  // Define an event handler to toggle the selected toppings.
  const toggleTopping = ({ target }) => {
    const clickedTopping = target.value;
    setSelected((prev) => {
      // Check if the clicked topping is already selected.
      if (prev.includes(clickedTopping)) {
        // Remove the clicked topping from the state.
        return prev.filter(t => t !== clickedTopping);
      } else {
        // Add the clicked topping to our state.
        return [clickedTopping, ...prev];
      }
    });
  };
  return (
    <div>
      {/* Render buttons for each pizza topping option. */}
      {options.map(option => (
        <button value={option} onClick={toggleTopping} key={option}>
          {selected.includes(option) ? 'Remove ' : 'Add '}
          {option}
        </button>
      ))}
      {/* Display the selected toppings in the order. */}
      <p>Order a {selected.join(', ')} pizza</p>
    </div>
  );
}
```

1. The `options` array contains the names of all available pizza toppings. It's static data that doesn't change, and it's defined outside the component.

2. The `selected` array represents the toppings the user selects for their personal pizza. This is dynamic data that changes based on user actions. We initialize it as an empty array.

We use the `.map()` method to create buttons for each topping in the `options` array, and when a button is clicked, the `toggleTopping()` event handler is called. This handler uses the event object to figure out which topping was clicked.

When you update an array in state, you don't just add new data to the old array. You replace the old array with a new one. This means you need to copy over any data you want to keep from the previous array. That's what the spread syntax (`...prev`) does.

We also use array methods like `.includes()`, `.filter()`, and `.map()`. These methods are handy for working with arrays in JavaScript.

Don't worry if this feels a bit overwhelming at first. You don't need to be a JavaScript expert to build React apps, but improving your JavaScript skills can make your journey more enjoyable and productive. Happy coding!

## Using objects in state

In React, we can use objects in state. This is useful when we want to store related data together. For example, we could use an object to store the user's name, email address, and password.

To use an object in state, we need to use the useState() hook with an object as the first argument. For example:

```
const [user, setUser] = useState({
  name: "",
  email: "",
  password: ""
});
```

This code creates a state variable called user. The value of user is an object with the properties name, email, and password.

We can update the value of user by calling the setUser() function. For example:

```
const handleChange = (event) => {
  const { name } = event.target;
  setUser({
    ...user,
    [name]: event.target.value
  });
};
```

This code defines an event handler called handleChange(). This event handler is called when the user changes the value of an input field.

The handleChange() function updates the value of the user state variable. The new value of user is a copy of the old value, with the value of the input field updated.

The square brackets around the name property name in the setUser() function are called a computed property name. This allows us to use the string value stored in the name variable as a property key.

## Using separate hooks for separate states

In React, we can use the useState() hook to create state variables. These state variables can be used to store data that changes over time.

Sometimes, it can be helpful to create separate state variables for data that changes separately. This can make our code more readable and easier to maintain.

For example, let's say we have a component that displays the current weather. We could create a single state variable to store all of the weather data. However, this would make our code more complex and difficult to maintain.

Instead, we could create separate state variables for each piece of weather data. For example, we could create a state variable for the temperature, a state variable for the humidity, and a state variable for the wind speed.

This would make our code much easier to read and maintain. If we needed to update the temperature, we would only need to update the state variable for the temperature. We would not need to update the state variables for the humidity or the wind speed.

Here is an example of how we could create separate state variables for the weather data:

```
const [temperature, setTemperature] = useState(72);
const [humidity, setHumidity] = useState(45);
const [windSpeed, setWindSpeed] = useState(10);
```

This code creates three state variables: temperature, humidity, and windSpeed. The setTemperature(), setHumidity(), and setWindSpeed() functions are used to update the corresponding state variables.

We can use these state variables to display the current weather data in our component. For example, we could use the temperature state variable to set the value of an h1 element.

```
<h1>The current temperature is {temperature} degrees Fahrenheit</h1>
```

We could also use the humidity and windSpeed state variables to set the values of other elements in our component.

Using separate state variables for data that changes separately can make our code more readable and easier to maintain. This is a good practice to follow when working with React.

## Summary

Let's wrap up what we've learned about building stateful function components using the useState React Hook. This is the simplified version:

Data in React: React helps us display both static and changing data in our web applications.

Hooks: Hooks are special functions in React that allow us to manage dynamic data in function components. Using the State Hook: We can use the State Hook like this:

```
const [currentState, stateSetter] = useState(initialState);
```

currentState holds the current state value. stateSetter is a function to update the state.

initialState sets the initial value of the state when the component first renders.

Event Handling: We can call state setters in event handlers. Event handlers can be simple ones right in the JSX or more complex ones defined outside of JSX.

State Setter Callbacks: When the next state depends on the previous state, we use a state setter callback function. It ensures that we're working with the most up-to-date state.

Arrays and Objects in State: We can organize related data that often changes together using arrays and objects. The spread syntax (...prev) helps us copy the previous state when updating.

Best Practice: It's usually better to have multiple, simpler states instead of one complex state object. This keeps our code clean and manageable.

That's the essential summary of what we've covered in this lesson. React makes it easier to work with data and build dynamic web applications!

## Why use useEffect?

Before Hooks, function components could only accept data in the form of props and return some JSX to be rendered. However, the useState() hook allows us to manage dynamic data, in the form of component state, within our function components.

The useEffect() hook allows us to run some JavaScript code after each render. This code can be used to: Fetch data from a back-end service.

Subscribe to a stream of data.

Manage timers and intervals.

Read from and make changes to the DOM.

Components will re-render multiple times throughout their lifetime. These key moments present the perfect opportunity to execute these "side effects".

The useEffect() hook can be used to run code at three key moments:

When the component is first added, or mounted, to the DOM and renders.

When the state or props of the component change, causing the component to re-render. When the component is removed, or unmounted, from the DOM.

We can use the useEffect() hook to run code that only needs to be executed once, such as fetching data from a back-end service. We can also use the useEffect() hook to run code that needs to be executed every time the component renders, such as subscribing to a stream of data.

The useEffect() hook is a powerful tool that can be used to manage side effects in our React components.

## The useEffect() hook

The useEffect() hook is a React hook that can be used to run some JavaScript code after each render. This code can be used to fetch data from a back-end service, subscribe to a stream of data, manage timers and intervals, or read from and make changes to the DOM.

The useEffect() hook takes two arguments:

A callback function that is called after each render.

An optional dependency array that specifies which values should trigger a re-render of the effect.

The callback function of the useEffect() hook is called after each render, even if the component's props or state have not changed. This means that we can use the useEffect() hook to run code that needs to be executed every time the component renders, such as fetching data from a back-end service or subscribing to a stream of data.

The dependency array of the useEffect() hook specifies which values should trigger a re-render of the effect. If any of the values in the dependency array change, the useEffect() hook will be re-run. This can be used to ensure that the effect is only re-run when the data that it depends on changes.

Here is an example of how to use the useEffect() hook to fetch data from a back-end service:

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

function App() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // Fetch data from the back-end service.
    fetch('/api/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return (
    <div>
      <h1>The data is: {data}</h1>
    </div>
  );
}
```

In this example, the useEffect() hook is used to fetch data from the back-end service. The dependency array is empty, which means that the effect will be re-run every time the component renders.

The useEffect() hook is a powerful tool that can be used to run side effects in our React components. It can be used to fetch data from a back-end service, subscribe to a stream of data, manage timers and intervals, or read from and make changes to the DOM.

## Cleaning up effects

Some effects require cleanup. For example, we might want to add event listeners to some element in the DOM. When we add event listeners to the DOM, it is important to remove those event listeners when we are done with them to avoid memory leaks.

A memory leak is a situation where a program allocates memory but never deallocates it. This can cause the program to run out of memory and crash.

To avoid memory leaks, we need to clean up after our effects. This means removing any event listeners that we have added to the DOM.

We can do this by returning a cleanup function from our useEffect() hook. The cleanup function will be called when the component is unmounted or re-rendered.

Here is an example of how to clean up an effect:

```
useEffect(() => {
  // Add an event listener to the DOM.
  document.addEventListener('keydown', handleKeyPress);

  // Specify how to clean up after the effect.
  return () => {
    // Remove the event listener from the DOM.
    document.removeEventListener('keydown', handleKeyPress);
  };
});
```

In this example, the handleKeyPress() function is called every time the user presses a key. When the component is unmounted, the removeEventListener() function is called to remove the event listener from the DOM.

This prevents a memory leak from occurring.

It is important to note that the cleanup function is optional. If we do not return a cleanup function, React will not remove any event listeners that we have added to the DOM.

This can lead to memory leaks, so it is important to return a cleanup function from our useEffect() hook whenever we add event listeners to the DOM.

## Controlling when effects are called

The useEffect() hook calls its first argument (the effect) after each time a component renders. However, we can control when the effect is called by passing an empty array to the useEffect() hook as the second argument. This second argument is called the dependency array.

The dependency array is used to tell the useEffect() method when to call our effect and when to skip it. Our effect is always called after the first render, but only called again if something in our dependency array has changed values between renders.

In the example you provided, the dependency array is empty. This means that the effect will only be called after the first render. The cleanup function will be called when the component is unmounted.

If we wanted the effect to be called every time the component renders, we would pass an array of values to the dependency array. For example:

```
useEffect(() => {
  // This effect will be called every time the component renders.
}, [someValue, anotherValue]);
```

The useEffect() hook is a powerful tool that can be used to run side effects in our React components. By controlling when the effect is called, we can ensure that it is only called when necessary.

## Fetching data

When building software, we often start with the default behavior and then modify it to improve performance.

The default behavior of the useEffect() hook is to call the effect function after every single render.

We can pass an empty array as the second argument for useEffect() if we only want our effect to be called after the component's first render.

In this exercise, we will learn to use the dependency array to further configure exactly when we want our effect to be called.

When our effect is responsible for fetching data from a server, we pay extra close attention to when our effect is called. Unnecessary round trips back and forth between our React components and the server can be costly in terms of:

Processing Performance

Data usage for mobile users API service fees

If the data that our components need to render does not change, we can pass an empty dependency array so that the data is fetched after the first render. When the response is received from the server, we can use a state setter from the useState() hook to store the data from the server's response in our local component

state for future renders. Using the useState() and useEffect() hooks together in this way is a powerful pattern that saves our components from unnecessarily fetching new data after every render!

An empty dependency array signals to the useEffect() hook that our effect never needs to be re-run, that it does not depend on anything. Specifying zero dependencies means that the result of running that effect will not change and calling our effect once is enough.

A dependency array that is not empty signals to the useEffect() hook that it can skip calling our effect after re-renders unless the value of one of the variables in our dependency array has changed. If the value of a dependency has changed, then the useEffect() hook will call our effect again!

Here is an example from the official React docs:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if the value stored by count changes
```

In this example, the count variable is the dependency. The useEffect() hook will only be called again if the value of count changes.

## Rules of hooks

There are two main rules to keep in mind when using hooks:

Only call hooks at the top level.

Only call hooks from React functions.

The first rule is important because React keeps track of the data and functions that we are managing with hooks based on their order in the function component's definition. If we call a hook inside of a loop, condition, or nested function, React will not be able to track it properly and it may cause unexpected behavior.

The second rule is important because hooks are designed to be used in function components. If we try to use a hook in a class component, React will throw an error.

Here is an example of how to avoid breaking the first rule:

```
function MyComponent() {
  // This is okay because we are calling the hook at the top level.
  useEffect(() => {
    // Do something with the hook.
  });

  if (someCondition) {
    // This is not okay because we are calling the hook inside of a condition
    useEffect(() => {
      // Do something with the hook.
    });
  }
}
```

Here is an example of how to avoid breaking the second rule:

```
class MyComponent extends React.Component {
  // Cant use a hook in class
  useEffect(() => {
    // Do something with the hook.
  });
}
```

## Separating hooks for separate effects

When we have multiple values that are closely related and change at the same time, it can make sense to group them together in a collection like an object or array. However, packaging data together can also add complexity to the code responsible for managing that data.

For example, in the code below, we are using a single useEffect() hook to manage both the position and menuItems data:

```
const [data, setData] = useState({ position: { x: 0, y: 0 } });

useEffect(() => {
  get('/menu').then((response) => {
    setData((prev) => ({ ...prev, menuItems: response.data }));
  });

  const handleMove = (event) =>
    setData((prev) => ({
      ...prev,
      position: { x: event.clientX, y: event.clientY },
    }));

  window.addEventListener('mousemove', handleMove);

  return () => window.removeEventListener('mousemove', handleMove);
}, []);
```

This code is a bit complex because it is trying to do two things at once: fetch the menu items from the server and update the position data. It would be better to separate these concerns into two separate use-Effect() hooks:

```
// Handle menuItems with one useEffect hook.
const [menuItems, setMenuItems] = useState(null);

useEffect(() => {
  get('/menu').then((response) => setMenuItems(response.data));
}, []);

// Handle position with a separate useEffect hook.
const [position, setPosition] = useState({ x: 0, y: 0 });

useEffect(() => {
  const handleMove = (event) =>
    setPosition({ x: event.clientX, y: event.clientY });

  window.addEventListener('mousemove', handleMove);

  return () => window.removeEventListener('mousemove', handleMove);
}, []);
```

This code is much easier to understand because each useEffect() hook is only responsible for one thing. This makes it easier to add to, reuse, and test the code.

In general, it is a good idea to separate concerns by managing different data with different Hooks. This will make your code easier to understand, maintain, and extend.

## Review

In this lesson, we learned how to write effects that manage timers, manipulate the DOM, and fetch data from a server. With the useEffect() hook, we can perform these types of actions in function components with ease!

Here are the main concepts from this lesson:

The useEffect() hook is a function that we can use to run code after each render.

The useEffect() hook takes two arguments: the effect function and the dependency array. The effect function is the code that we want to run after each render.

The dependency array is an array of variables that the effect function depends on.

If the dependency array is empty, the effect function will only be called once, after the first render.

If the dependency array is not empty, the effect function will be called again whenever any of the variables in the dependency array change.

The useEffect() hook also returns a cleanup function, which is called when the component is unmounted. The useEffect() hook is a powerful tool that can be used to run side effects in our React components. By understanding how to use the useEffect() hook, we can write more efficient and reusable code.

Here are some tips for using the useEffect() hook:

Use the dependency array to avoid unnecessary re-renders. Use the cleanup function to prevent memory leaks.

Group related effects together.

Separate concerns by using different hooks for different tasks.

## Getting Started with Styling in React

Alright, let's talk about making your React app look good! Styling is super important because it affects how users feel about your app and helps it stand out.

As your app gets bigger and fancier, styling becomes a big deal. You need a good plan to keep your styles neat and tidy.

In this lesson, we'll start with the basics of styling in React. We'll chat about two ways to do it: one where you put styles right in your components, and another where you use special style files.

We'll also look at some React-specific rules for styling. Then, we'll get into something called "CSS modules." It sounds fancy, but it's a way to keep your styles neat and easy to reuse.

By the end of this lesson, you'll know how to make your React components look great, and you'll do it in a way that's organized and easy to manage. So, let's dive into React styling together!

## Different Ways to Style React Components

In React, there are several ways to style your components. We'll talk about two of them here: inline styles and style object variables.

Inline Styles are styles that you write directly as attributes in your JSX, like this:

```
<h1 style={{ color: 'red' }}>Hello world</h1>
```

Notice the double curly braces. The outer ones indicate that it's JavaScript code, while the inner ones create a JavaScript object.

But if you have lots of styles, this can get messy. So, there's another way.

You can store your styles in a style object variable. First, you create an object with your styles, like this:

```
const darkMode = {
  color: 'white',
  background: 'black',
};
```

Then, you can use that object to style your component, like so:

```
<h1 style={darkMode}>Hello world</h1>
```

When you're styling components in JSX, there are some rules to follow.

In React, we use camelCase to write CSS property names. This means that if you want to set a CSS property like background-color, you should write it as backgroundColor in React. This is because hyphens (-) are reserved in JavaScript for subtraction, so we use camelCase to stay consistent.

In JavaScript, style values are typically written as strings, even for numeric values. For instance, you write '450px' or '20%'. If you use a number, React assumes it's in pixels. For example, { fontSize: 30 } means a font size of 30 pixels.

If you want to use units other than 'px', you should use a string. For example, { fontSize: "2em" }. You can still specify 'px' within a string, but it's not necessary for most styles.

There are a few styles where you don't need to worry about specifying 'px' because they typically don't use it.

## Multiple Stylesheets

When your React application starts to grow, managing styles can get tricky. While inline styles and style objects work, they can become hard to manage as your app becomes more complex.

One smart way to keep your styles organized, modular, and reusable is by using separate stylesheets for each component.

You can import a stylesheet like this:

```
<h1 style={darkMode}>Hello world</h1>
```

However, when you have multiple stylesheets and some use the same class names, it can lead to style clashes.

To avoid this, you can use CSS modules. This ensures that the styles you import are only available for the component that imported them. CSS modules automatically generate unique class names, saving you from naming conflicts.

To use CSS modules, name your stylesheet like this:

fileName.module.css

This tells React to treat it as a CSS module. Then, import it into your component file:

```
import styles from './fileName.module.css';
```

Now, the styles object contains the class selectors from fileName.module.css. You can apply these styles using the className attribute, not class, since class is a reserved JavaScript keyword.

This is the recommended way to style React components, aligning with React's philosophy of composition and maintainability.

## Review

Congratulations on completing the lesson on styling in React apps! Here is a recap of what you learned:

There are 4 ways to style React components: inline styling, object variable styling, stylesheets, and CSS modules.

Inline styling is the simplest way to style a component. It involves adding style attributes to the component's HTML tag.

Object variable styling is similar to inline styling, but it involves storing the styles in an object variable. Stylesheets are files that contain CSS code. They can be imported into React components and used to style them.

CSS modules are a way to scope CSS styles to specific components. This can help to prevent style conflicts.

Here are some additional things to keep in mind when styling React components:

Style names in React must be in camelCase. For example, background-color becomes backgroundColor.

In React, a number style value is automatically interpreted with px. You can use CSS variables to create reusable styles.

You can use CSS media queries to style components differently for different screen sizes.

## React Forms Made Simple

Let's simplify how forms work in React. Imagine filling out a form on a website. In a traditional setup, the server only knows what you've typed when you hit the "submit" button, and this can cause delays and conflicts.

In React, we do things differently. We want the server to know about every keystroke you make, right as you type it. This way, everything stays in sync, and there are no conflicts. It's like having a real-time conversation with the server, making your application smoother and more responsive.

## React Forms and onChange

In traditional HTML forms, the browser handles the form's state and only sends data to the server when you hit the submit button. However, React handles forms a bit differently.

In React, the form's state is managed by the component itself, and updates happen through the onChange event. When you interact with an input field, like typing or deleting characters, the onChange event triggers and updates the component's state.

This means React can immediately show any changes you make and update the view in real-time. It's a more responsive and dynamic way to handle forms.

## Setting Initial State for the Input

Great! When someone types or deletes something in the <input> field, the handleUserInput() function will update userInput with the <input>'s text.

Now, we need to set the initial state for userInput using the State hook. What should the initial value be?

Since userInput will be displayed in the <input> element, we should start with an empty string. That way, when a user first visits the page, the input field will be empty, and it will look like they haven't typed anything yet.

Updating an Input's Value

When a user types in the <input> field, it triggers a change event that calls the handleUserInput() function. That's great!

handleUserInput() sets userInput to the current text in the <input> field, which is also good.

However, there's a problem: even though userInput gets updated, the <input> element's value prop doesn't update.

In React, we use the value prop of an input element to control and synchronize its value with the component's state. Without setting the value prop, changes in the input won't reflect in the component's state, which can lead to inconsistencies and issues.

To ensure that the input's value matches the component's state, we set the value prop and use the onChange event to update the state when the user types. When the state changes, the component re-renders, and the value prop reflects the new value from the component's state.

This approach ensures that the component's state serves as the "source of truth" for the input's value, making form data consistent and easily manageable. For instance, in a login form with email and password inputs, we'd set the value prop for both inputs and update the component's state whenever the user types in a new email or password. This way, the form data remains up-to-date with the user's input.

Controlled vs. Uncontrolled Inputs

When working with React forms, you'll often come across two terms: controlled components and uncontrolled components.

An uncontrolled component is one that manages its own internal state. It's like a self-sufficient entity. For instance, think of a typical <input type='text' /> element. It appears as a text box on the screen. If you want to know what text is currently in the box, you can directly ask the <input> element like this:

```
let input = document.querySelector('input[type="text"]');
let typedText = input.value; //typedText now holds the text in the text box
```

In this case, the <input> element keeps track of its own text. It maintains its internal state by remembering data about itself. So, it's an uncontrolled component.

On the other hand, a controlled component doesn't maintain its own state. It relies on someone else to manage it, typically through props. Most React components are controlled components.

In React, when you give an <input> element a value attribute, it becomes controlled. It stops using its internal storage and starts looking to the state for its value. This is considered a more "React" way of handling things.

In our exercises, you saw that the page updated every time we typed into the input. React controlled the input's value with the state. So, we've been working with controlled components all along!

You can find more detailed information about controlled and uncontrolled components in the React documentation.

reat job! You've just created your first React form. Let's quickly recap what you've learned:

In a React form, the component manages the form's state, and updates are triggered by the onChange event when the user interacts with the form.

The onChange event uses an event handler to capture changes in the form's input fields and determine the appropriate actions to take in response to those changes.

React forms utilize the State hook to store the input field's value in the component's state. The state can then be updated using the state setter function.

React forms can be categorized as controlled or uncontrolled components. Most React forms are controlled, meaning they control the input's value with the component's state.

Congratulations! You've completed this lesson, and the skills you've acquired will prove to be valuable as you continue to build more React applications!