Creating and Nesting components--

React apps are made out of components.

A component is a piece of UI that has its own logic and appearance.

React components are JS functions that return markup

Declaration of MyButton--

```
function MyButton(){
  return(
    <button>Im a button</button>
  );
}
```

Nesting MyButton into another component--

```
export default function MyApp(){
  return(
    <div>
    <h1>Welcome to my app</h1>
    <MyButton/>
    <div>
  );
}
```

IMP-React component should start with a capital letter--

while html tags must be lowercase

```
function MyButton(){
  return(
        <button>
        I'm a button
        </button>
    );
}

export default function MyApp(){
    return(
        <div>
        <h1>Welcome to my App</h1>
        <MyButton>
        <div>
    );
}
```

export default keywords specify the main component in the file.

The markup syntax you've seen above is called JSX.

JSX is stricter than HTML. You have to close tags like <br />. Your component also can't return multiple JSX tags. You have to wrap them into a shared parent, like a <div>...</div> or an empty <>...</> wrapper:

Adding styles--

In React, you specify a CSS class with className. It works the same way as the HTML class attribute:

```
<img className="avatar" />
```

Then you write the CSS rules for it in a separate CSS file:

```
/* In your CSS */
.avatar {
  border-radius: 50%;
}
```

Displaying data--

JSX lets you put markup into JavaScript. Curly braces let you “escape back” into JavaScript so that you can embed some variable from your code and display it to the user.

```
return (
  <h1>
    {user.name}
  </h1>
);
```

example--

```
const user = {
  name: 'Hedy Lamarr',
  imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
  imageSize: 90,
};

export default function Profile() {
  return (
    <>
      <h1>{user.name}</h1>
      <img
      className="avatar"
      src={user.imageUrl}
      alt={'Photo of ' + user.name}
      style={{
        width: user.imageSize,
        height: user.imageSize
      }}
    />
  </>
 );
}
```

In the above example, style={{}} is not a special syntax, but a regular {} object inside the style={ } JSX curly braces. You can use the style attribute when your styles depend on JavaScript variables.

Simplified Analogy--

Think of it like this:

Outer {}: JavaScript mode on!

Inner {}: Here's my object literal.

This is a common pattern for inline styles in React!

Conditional rendering--

In React, there is no special syntax for writing conditions. Instead, you'll use the same techniques as you use when writing regular JavaScript code. For example, you can use an if statement to conditionally include JSX:

```
let content;
```

```
if (isLoggedIn) {
  content = <AdminPanel />;
} else {
  content = <LoginForm />;
}
return (
  <div>
    {content}
  </div>
);
```

If you prefer more compact code, you can use the conditional ? operator--

```
<div>
  {isLoggedIn ? (
    <AdminPanel />
  ) : (
    <LoginForm />
  )}
</div>
```

When you don’t need the else branch, you can also use a shorter logical && syntax:

```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```

Rendering lists--

You will rely on JavaScript features like for loop and the array map() function to render lists of components.

For example, let’s say you have an array of products:

```
const products = [
 { title: 'Cabbage', id: 1 },
 { title: 'Garlic', id: 2 },
 { title: 'Apple', id: 3 },
];
```

Inside your component, use the map() function to transform an array of products into an array of <li> items:

```
const listItems = products.map(product=>
<li key={product.id}>
{product.title}
</li>);

return(
   <ul>{listItems}</ul>
);

const products = [
 { title: 'Cabbage', isFruit: false, id: 1 },
 { title: 'Garlic', isFruit: false, id: 2 },
 { title: 'Apple', isFruit: true, id: 3 },
];

export default function ShoopingList(){
   const listItems=products.map(product=>
   <li
   key={product.id}
   style={{
   color:product.isFruit?'magenta':'darkgreen'
   }}
   >
   {product.title}
   </li>
   );

   return(
     <ul>{listItems}</ul>
   );
}
```

Responding to events--

You can respond to events by declaring event handler functions inside your components:

```
function MyButton(){
  function handleClick(){
    alert('you clicked me!');
  }

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

Notice how onClick={handleClick} has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

Updating the screen--

Often, you'll want your component to "remember" some information and display it. For example, maybe you want to count the number of times a button is clicked. To do this, add state to your component.

First, import useState from React:

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

Now you can declare a state variable inside your component:

```
function MyButton() {
 const [count, setCount] = useState(0);
 // ...}
```

You'll get two things from useState: the current state (count), and the function that lets you update it (setCount). You can give them any names, but the convention is to write [something, setSomething].

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

  function handleClick(){
    setCount(count+1);
  }

  return (
    <button onClick={handleClick}>
    Clicked {count} times
    </button>
  );
}
```

A full code for this will be--

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

export default function MyApp(){
  return (
    <div>
    <h1>Counters that update separately</h1>
    <MyButton/>
    <MyButton/>
    </div>
  );
}

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

  function handleCLick(){
    setCount(count+1);
  }

  return (
    <button onClick={handleClick}>
    Clicked {count} times
    </button>
  );
}
```

Using Hooks--

Functions starting with use are called Hooks. useState is a built-in Hook provided by React.

You can also write your own Hooks by combining the existing ones.

Hooks are more restrictive than other functions. You can only call Hooks at the top of your components (or other Hooks). If you want to use useState in a condition or a loop, extract a new component and put it there.

Sharing data between components--

To make both MyButton components display the same count and update together, you need to move the state from the individual buttons “upwards” to the closest component containing all of them.

Now when you click either button, the count in MyApp will change, which will change both of the counts in MyButton. Here’s how you can express this in code.

```
export default function MyApp() {
 const [count, setCount] = useState(0);
```

```
  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

function MyButton() {
  // ... we're moving code from here ...
}
```

Then, pass the state down from MyApp to each MyButton, together with the shared click handler. You can pass information to MyButton using the JSX curly braces

```
export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}
```

The information you pass down like this is called props. Now the MyApp component contains the count state and the handleClick event handler, and passes both of them down as props to each of the buttons.

Finally, change MyButton to read the props you have passed from its parent component:

```
function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```

full code--

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

export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```