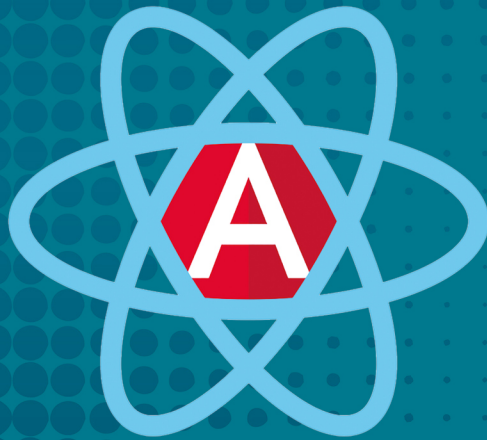


Building Web Applications Using **Angular** & **React**



Lesson 6

React: Advanced Techniques

Contents

Using Destructuring.....	3
Default Values for props.....	8
Styles and Components.....	10
State	14
Hooks.....	27
Homework	40

Using Destructuring

We came across destructuring syntax when we studied JavaScript. It is a possibility to assign an array or object to a set of variables by breaking it into pieces. Typical syntax is as follows:

```
let [variable list] = array name
```

For instance,

```
let arr = [7, 88, -3];  
/*  
  a = 7  
  b = 88  
  c = -3  
*/  
let [a, b, c] = arr;
```

In our code, the value of the zero element was assigned to **a**, the value of the first element to **b**, and the value of the second element to **c**. Destructuring goes from left to right. It means that the value of the zero element of the array falls into the leftmost element of the initialization list. Nothing happens to the array to the right of the equal sign. Another example:

```
/*  
  d = 88  
  e = -3  
*/  
let [, d, e] = arr;
```

It is the same `arr` array, but we skip the leftmost element upon destructuring. As a result, the value of the zero element of the array is ignored, and we begin with the first one, whose value falls into `d`.

When destructuring, you can use the `...` operator.

It is called a `spread` operator. If it is specified, it means that you should write all the array elements remaining to the right of the equal sign in the variable that follows it. This operator will be rightmost in the list of variables. Let's consider an example:

```
let arr = [7, 88, -3];  
/*  
    f = 7  
    arr2 = [88,-3]  
*/  
  
let [f, ...arr2] = arr;
```

The value of the zero element is written in `f` followed by `...arr2`. It means that we create the `arr2` variable and ask to write all the elements remained in the `arr` array in it. The `arr2` variable will be an array with two elements: `88` and `-3`.

Destructuring is also used when working with objects. For this use `{}` instead of `[]`.

```
let obj = {name:"Bill", lastName:"White"};  
let {name,lastName} = obj;
```

We have created the `obj` object. It has two properties: `name` and `lastName`. We use destructuring syntax to extract these properties into variables of the same name: `name` and `lastName`. When destructuring an object, we should specify

names of the properties, which we want to get, on the left. For instance:

```
let obj = {name:"Bill", lastName:"White"};
let {name} = obj;
```

We only extract `name`. If you want your variable to be named differently from the object properties, you can use this syntax:

```
let obj = { name: "Bill", lastName: "White" };
let { name: n, lastName: l } = obj;
```

The value of the `name` property will be written to the `n` variable. The value of the `lastName` property will be written to `l`.

Why did we recall the destructuring? The thing is that this syntax is often used to create React apps. We will do it too, and we will start right now.

Let's create an app that prints a writer's first and last name. We will pass these parameters through `props`.

The *App.js* file:

```
import React from "react";
import ReactDOM from "react-dom";

import App from "./App";
const rootElement = document.getElementById("root");
ReactDOM.render(
  <React.StrictMode>
    <App name="Ernest" lastName="Hemingway" />
  </React.StrictMode>,
  rootElement
);
```

There is nothing new here. We pass values of the properties `name` and `lastName`. Open *App.js*:

```
import React from "react";
import "../styles.css";

export default function App(props) {
  /*
    destructure the props object
  */
  let { name, lastName } = props;
  return (
    <div className="App">
      <h1>Information about writer:</h1>
      <h2>{name}</h2>
      <h2>{lastName}</h2>
    </div>
  );
}
```

We received the `props` object and used the destructuring syntax to write the value of the `name` property to the `name` variable, the value of the `lastName` property to the `lastName` variable.

We could rename the variables where the values will be written.

```
let { name: n, lastName: l } = props;
return (
  <div className="App">
    <h1>Information about writer:</h1>
    <h2>{n}</h2>
    <h2>{l}</h2>
  </div>
);
```

In this case, we use the variables `n` and `l`. If the syntax is new to you, try it in practice. It may look a little strange, but you will quickly get used to it.

- [Link](#) to the project code.

Default Values for props

We set values for all **props** in our examples. But what happens if the user of our component sets values to two of the four properties? What happens to the code of our component if we try to access a property whose value was not set?

The answer is obvious: nothing good. The property value will be **undefined**. So what should we do? We need to set default values for our properties. They will be used when the user specifies her value. To set default values, we should use the **defaultProps** property.

Let's turn to the writer project again. Set a value for the last name inside the file *index.js*.

```
ReactDOM.render(  
  <React.StrictMode>  
    <App name="Will" />  
  </React.StrictMode>,  
  rootElement  
) ;
```

We specified a value only for **name**. Inside the *App.js* file, set default values.

```
export default function App(props) {  
  return (  
    <div className="App">  
      <h1>Information about writer:</h1>  
      <h2>{props.name}</h2>  
      <h2>{props.lastName}</h2>  
    </div>  
  ) ;  
}
```



```

/*
  Initialize default values.
  Use the syntax to create an object.
*/
App.defaultProps = { name: "William",
                      lastName: "Shakespeare" };

```

We set a default value for the `App` component. The default value for `name` is `William`. The value for `lastName` is `Shakespeare`. The component user has passed us a value only for the `name`. It means that the value for `lastName` will be default. And it will be `Shakespeare`.

The syntax for setting default values is as follows:

```
Component_name.defaultProps =
```

You can also set a value for each property separately:

```

export default function App(props) {
  return (
    <div className="App">
      <h1>Information about writer:</h1>
      <h2>{props.name}</h2>
      <h2>{props.lastName}</h2>
    </div>
  );
}

App.defaultProps.name = "Ernest";
App.defaultProps.lastName = "Hemingway";

```

► [Link](#) to the project code.

The syntax we studied is also applied for class components.

Styles and Components

We already know how to set styles in React using the `className` attribute. And you can also set styles directly inside the component itself. The `style` attribute is used for this. Create a project where we are going to do this. The content of the *App.js* file:

```
import React from "react";
import "./styles.css";

function Intro() {
  return <p style={{ color: "green" }}>
    Some intro text</p>;
}

function Main() {
  return (
    <p style={{ color: "red", fontSize: "150%" }}>
      Here must be the main idea </p>
  );
}

function End() {
  return (
    <p style={{ color: "blue",
      backgroundColor: "yellow" }}>
      It is the last part of our text </p>
  );
}

export default function App() {
  return (
    <div className="App">
```

```

    <Intro />
    <Main />
    <End />
  </div>
);
}

```

Output:

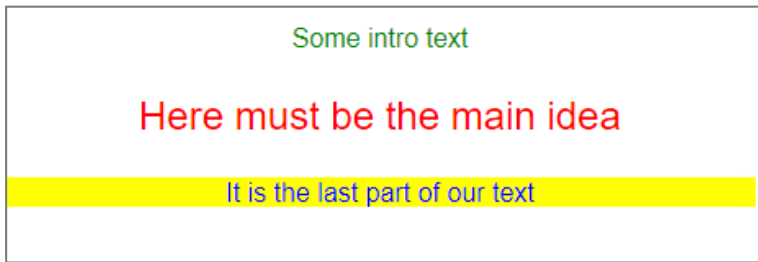


Figure 1

We have created three additional functional components in our code. Each of them is responsible for its part of the text. **Intro** — introduction, **Main** — main text, **End** — final part. We set styles for each component. Let's take **Intro** as an example. It looks like this:

```

function Intro() {
  return <p style={{ color: "green" }}>
    Some intro text</p>;
}

```

We have used the **style** attribute to set styles. Why do we use so many curly brackets? The first pair of **{}** sets a dynamic value. The second pair is used because we need to create an object containing style settings to set styles. In the example

above the object with one `color` property was created. And we created an object with two properties `color` and `fontSize` for the `Main` component.

```
function Main() {
  return (
    <p style={{ color: "red", fontSize: "150%" }}>
      Here must be the main idea</p>
  );
}
```

We set two properties — `color` and `backgroundColor` — for the `End` component.

```
function End() {
  return (
    <p style={{ color: "blue",
      backgroundColor: "yellow" }}>
      It is the last part of our text</p>
  );
}
```

► [Link](#) to the project code:

Styles for class components are set by the same principle. Let's create a class component that displays text.

```
export default class App extends React.Component {
  render() {
    return (
      <p
        style={{
          fontSize: "125%",
          color: Math.random() < 0.5 ? "red" : "green"
        }}>
```

```
    Toto, I've got a feeling we're not  
    in Kansas anymore  
  </p>  
  );  
}  
}
```

There is a value set for the font size in the code. Font color is also selected randomly in the code. It can be red or green, depending on the value returned by [Math.random](#).

- [Link](#) to the project code.

When might we need to set styles directly from the code? For example, if we want to change styles dynamically, depending on some condition.

State

One of the main React concepts is state. [State](#) allows you to store the internal state of the component. State stores important attributes of a component. Values from the state are used when displaying a component. If an attribute is not involved in the component's render, it should not be put inside a state.

We will begin the introduction to the concept of [state](#) with class components.

Create a project Click Counter. The interface of this project is just one button. When the app starts, it displays only the initial value **0**. Each click on the button increases the counter value. The first click will change the value to **1**, the second to **2**, and so on.

We see this when the app starts: **0**.

After the first click: **1**.

After five clicks: **5**.

Let's take a look at the project's code. The file *App.js*:

```
import React from "react";

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentValue: 0
    };
  }
}
```

```

render() {
  const handleClick = () => {
    this.setState({currentValue:
                    this.state.currentValue +
                    1 });
  };
  return <button onClick={handleClick}>
    {this.state.currentValue}
  </button>;
}
}
export default class App extends React.Component {
  render() {
    return (
      <>
        <Counter />
      </>
    );
  }
}

```

We have two components. The `App` is already familiar to us. It keeps it standard. The second component — `Counter` — is a button to be clicked. Let's analyze the code of this component in detail. We begin with a constructor.

```

constructor(props) {
  super(props);
  this.state = {
    currentValue: 0
  };
}

```

The constructor takes `props` as a parameter. The first line of the constructor — `super(props);` — calls the constructor of the parent class. We must do it for correct operation.

The next line sets a value for the component's state. When we talk about the component's state, we actually talk about an object containing a set of properties. To access the state object inside the constructor, we should use `this`. The component's state is a property named `state`. A full statement for setting the initial values is as follows:

```
this.state = {  
  currentValue: 0  
};
```

We use curly brackets because `state` is an object that can have many properties. In our case, our state has one attribute — `currentValue`. We set it to `0`.

Let's move on to the `render` method.

```
render() {  
  const handlerClick = () => {  
    this.setState({ currentValue:  
      this.state.currentValue + 1 });  
  };  
  
  return <button onClick={handlerClick}>  
    {this.state.currentValue}  
  </button>;  
}
```

Let's analyze it, and we will begin from the end.

```
return <button onClick={handlerClick}>  
  {this.state.currentValue}  
</button>;
```


We described the appearance of our component in `return`. We also specified that the `handlerClick` handler is to be called upon button click. We added text to the button. The text on the button is the `currentValue` attribute of our state. Remember how we said earlier that state attributes must be involved in the component's render? Our code proves this. To read the `currentValue` value, we need to use `this.state.currentValue`.

When the app starts, the `currentValue` is 0. Let's now dive into the click handler code:

```
const handlerClick = () => {  
  this.setState({ currentValue:  
    this.state.currentValue + 1 });  
};
```

It looks menacing, but it is not that scary, actually. Our handler is an arrow function. We have already encountered this approach before.

Let's ask ourselves, "What should happen in the handler code?" We should add one to the current value and update the text on the button. Our current value is the state value named `currentValue`. It means that we need to increase the value of this variable by one. To update the value of the state variable, we need to call a special method named `setState`. It updates the specified state attributes. If one of the attributes has changed, you should reset it. It will automatically save its value. In our case, we should set a new value for `currentValue`.

```
this.setState({ currentValue:  
  this.state.currentValue + 1 });
```

And we should pass an object containing the changed value as a parameter.

```
currentValue: this.state.currentValue + 1
```

To set the new value, we take the current value and increase it by one. There is no other code in our handler! So where is the interface updated?

This is where the magic of React works for us. Once we change the state, React updates only those parts of the page that use the changed value. It means that the text on the button has changed automatically. This mechanism will only work when you use `setState`.

Notice the implicit use of Virtual DOM. Changes take place there first, and only after this React checks the difference between the DOM page and Virtual DOM. The found changes will be made to the DOM page. In this case, the page will not be redrawn entirely. Only the updated parts will.

► [Link](#) to the project code.

When programmers talk about `props`, they often use the word `immutable`. This way they want to emphasize that `props` is an immutable value. It comes into the component and contains the initial settings. They do not need to be changed. When they talk about `state`, they use another word — `mutable`. This emphasizes that the state is a mutable value and it HAS TO BE changed.

Let's create a new version of the project. In the new version, our `Counter` component will take the initial value and the increment value taken from the outside. We will use `props` to set properties. The *App.js* code is as follows:

```

import React from "react";

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentValue: props.startValue
    };
  }

  render() {
    const handlerClick = () => {
      this.setState({
        currentValue: this.state.currentValue +
          this.props.incValue
      });
    };
    return <button onClick={handlerClick}>
      {this.state.currentValue}
    </button>;
  }
}

export default class App extends React.Component {
  render() {
    return (
      <>
        <Counter startValue={0} incValue={3} />
      </>
    );
  }
}

```

What changes does the new version have? We use **props** to set the **startValue** (*initial value*) and **incValue** (*increment value*).

The component's code has different contents of the **set-State**. Now we increase the current value by **incValue**.

```
this.setState({
  currentValue: this.state.currentValue +
    this.props.incValue
});
```

There are no other changes in the code.

Application start: .

After the first click: .

After the second click: .

We hope that you understand how both projects work.

► [Link](#) to the project code:

And let's modify our example once again. In the previous examples, we changed the text on the button. In the new example, the text will change in `div` upon button click. This is how our app will look:

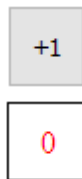


Figure 2

After the first click:

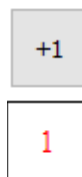


Figure 3

To solve this problem, we will create two components. One for the button, one for `div`. The main question is where we will store the state. It should be accessed from the button and `div`, after all. If you have this problem, the best choice is to store the state in the class that has access to the button and `div`.

The entire logic of working with state must be in it. In our case, the perfect candidate for storing state is the `App` class.

Let's look at the code of the `App.js` file. Begin with the full code. Look it through from top to bottom.

```
import React from "react";
import "../styles.css";

class Button extends React.Component {
  render() {
    const btnClick = () => {
      this.props.onClickAct(this.props.btText);
    };
    return (
      <button className="Button" onClick={btnClick}>
        +{this.props.btText}
      </button>
    );
  }
}

class Display extends React.Component {
  render() {
    return <div className="Display">
      {this.props.displayText}
    </div>;
  }
}
```

```

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentValue: 0
    };
  }

  render() {
    const incButtonVal = 1;
    const handleClick = incValue => {
      this.setState({
        currentValue: this.state.currentValue +
                      incValue
      });
    };

    return (
      <>
        <Button btText={incButtonVal}
                  onClickAct={handleClick} />
        <Display displayText={this.state.currentValue}/>
      </>
    );
  }
}

```

And now let's analyze it. Begin with the [App](#) class. It contains the entire logic of working with the state.

```

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currentValue: 0
    };
  }
}

```

```

}
render() {
  const incButtonVal = 1;
  const handlerClick = (incValue) => {
    this.setState({
      currentValue: this.state.currentValue +
        incValue
    });
  };

  return (
    <>
      <Button btText={incButtonVal}
        onClickAct={handlerClick} />
      <Display displayText={this.state.currentValue}/>
    </>
  );
}

```

The constructor's body remains unchanged. Major changes are in the `render` method.

```

const incButtonVal = 1;
const handlerClick = (incValue) => {
  this.setState({
    currentValue: this.state.currentValue +
      incValue
  });
};

```

Our handler gets one argument now: the `incValue`. Notice that the code that works with the state is in the `App` class, not in the `button` class. This is done because the code for changing state must be in the same class where this state is declared.

```

return (
  <>
    <Button btText={incButtonVal}
            onClickAct={handlerClick} />
    <Display displayText={this.state.currentValue}/>
  </>
);

```

`return` also looks differently. We create two components: `Button` and `Display` (`div` for display), pass the following properties to the button: `incButtonVal` (increment value) and `handlerClick` (reference to our state change function). `onClickAct` is an attribute of our button class. We use it to call `handlerClick` inside the `Button` class. We pass the current value of our counter to the `Display` component. When the app starts, it is equal to 0. Let's consider the `Button` class now.

```

class Button extends React.Component {
  render() {
    const btnClick = () => {
      this.props.onClickAct(this.props.btText);
    };
    return (
      <button className="Button" onClick={btnClick}>
        +{this.props.btText}
      </button>
    );
  }
}

```

The button class has only the `render` method. The button click handler `btnClick` calls the state change function from the `App` class.


```
const btnClick = () => {
  this.props.onClickAct(this.props.btText);
};
```

The name of this function is in the `OnClickAct` property. We pass the increment value to it as a parameter. It is in the `btText` property.

```
return (
  <button className="Button" onClick={btnClick}>
    +{this.props.btText}
  </button>
);
```

We describe the created button in `return`. We set its appearance using styles from the `style.css`. Indicate that the `btnClick` handler is to be called upon button click. The plus sign before `{this.props.btText}` allows displaying `+` before the property value.

The `Display` component has a simple implementation:

```
class Display extends React.Component {
  render() {
    return <div className="Display">
      {this.props.displayText}</div>;
  }
}
```

Every time we change the state, React automatically updates text inside `div`.

Be sure to practice this.

- [Link](#) to the project code.

New ES features allow us to reduce the cost of writing a class component. Instead of using a constructor like this:

```
constructor(props) {  
  super(props);  
  this.state = {  
    currentValue: 0  
  };  
}
```

We can assign a value to `state` as a class field:

```
export default class App extends React.Component {  
  state = {currentValue:0};  
}
```

In this case we do not need to create a constructor and describe its body.

Hooks

We laid the groundwork for working with a state inside class components. So how do we work with a state in functional components? To do this, we will use the hook mechanism.

Hooks are a set of useful React functions used to solve various problems. Why such a strange name? Hooks are used to catch or hold something. You can consider a hook in React as a tool that will hook a solution to a specific problem.

Let's begin getting to know about hooks with a state hook.

Meet `useState` — a state hook. The syntax of `useState` is as follows:

```
useState(start_value_for state)
```

The `useState` function returns an array containing a link to a state value in the zero element and a function to update state in the first element. We will use the destructuring syntax, which we talked about earlier, to write these values into variable.

Example of the `useState` call (we will create a state variable with the initial value of 0):

```
const [counterVal, setCounterVal] = useState(0);
```

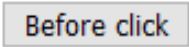
We call `useState` and set the initial value 0 for the state variable. The reference for accessing this value is written in the `counterVal` variable. The reference to a function used to set a new value for the state variable is written into the `setCounterVal` variable. Notice that we do not need to implement a function

body to update this state variable. React will do it for us. We need to call this function by specifying `setCounterVal` and pass a new state value as a parameter. The names `counterVal` and `setCounterVal` are quite random. They can be anything. The main thing is they should make sense. If you want to store multiple values in a state, you need to call `useState` once for each value. Each time the `useState` is called, the pair of variables to the left of the equal sign must be different.

Remember to import `useState` before use! Otherwise, you will get an error. In order to use `useState`, we need to use a named import.

Let's create the first project for using state inside functional components. There will be a button with the initial caption in our project. The text will change upon button click.

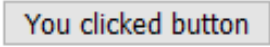
The initial appearance of the app:



Before click

Figure 4

After the click:



You clicked button

Figure 5

The content of the *App.js* file:

```
import React, { useState } from "react";
import "./styles.css";

function Button(props) {
  /*
    Set the state hook
```

btText will contain the currentstate value.
 setBtText is a function for setting a new value
 to the state.
 We will pass the new value to it.
 The implementation of this function remains
 with React.

btText and setBtText are random names,
 you can choose any other name instead.

```
*/

const [btText, setBtText] = useState("Before click");

const btClick = () => {
  setBtText("You clicked button");
};
return <button onClick={btClick}>{btText}</button>;
}

export default function App() {
  return (
    <>
      <Button />
    </>
  );
}
```

Let's analyze important points of this code.

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

In addition to React, we import the `useState` hook here.
 The hook is set inside the functional component `Button`.

```
const [btText, setBtText] = useState("Before click");
```

The initial value for the state is the Before click line. Read access to the state variable is written in the `btText`. A reference to the function for changing the state value is written in `setBkText`. You cannot change state through `btText`. You can do this only through the update function. In this case, React guarantees a successful update of the state followed by a successful update of the interface. And a little more of the component's code:

```
const btClick = () => {  
  setBtText("You clicked button");  
};  
  
return <button onClick={btClick}>{btText}</button>;
```

As usual, indicate that we respond to the button click. The most important line here is this:

```
setBtText("You clicked button");
```

We call the state update function and pass a new value.

This approach may look unusual at first, but it will not cause you any difficulties.

► [Link](#) to the project code.

Let's modify this project. Now the state will store not only the caption on the button, but also the button's background color. The caption and background color will change upon click.

The initial appearance of the app:

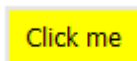


Figure 6

After the click:



You clicked me

Figure 7

App.js code:

```
import React, { useState } from "react";
import "./styles.css";

function Button() {
  /*
    We will store two attributes in the state caption
    on the button and the button's background color
  */
  const [bkColor, setBkColor] = useState("yellow");
  const [text, setText] = useState("Click me");
  const btnClick = () => {
    setBkColor("red");
    setText("You clicked me");
  };

  return (
    <button
      style={{ backgroundColor: bkColor, height: "30px"}}
      onClick={btnClick}>
      {text}
    </button>
  );
}

export default function App() {
  return (
    <>
      <Button />
    </>
  );
}
```

The first thing in which this code differs from the previous one is that we call `useState` twice.

```
/*
  We will store two attributes in the state caption
  on the button and the button's background color
*/
const [bkColor, setBkColor] = useState("yellow");
const [text, setText] = useState("Click me");
```

We must do it in order to set two attributes for our state: text and background color.

The second difference: we call a specific function to update each attribute in the button handler:

```
const btnClick = () => {
  setBkColor("red");
  setText("You clicked me");
};
```

The third difference: when creating the button, we set its styles using the `style` attribute:

```
return (
  <button
    style={{backgroundColor: bkColor,
             height: "30px"}}
    onClick={btnClick}>
    {text}
  </button>
);
```

The takeaway of this project is as follows: you call `useState` as many times as many state attributes you need to store.

- [Link](#) to the project code.

Remember some important rules of hooks:

1. Use hooks only at the top level. It means that you should put a hook at the beginning of the functional component's code.
2. Do not use hooks inside loops, conditional operators, or nested functions. Instead, always use hooks only at the top level of React functions.
3. Do not call hooks from regular JavaScript functions.

Let's create the third project to work with a state hook in a functional component. Repeat our button counter project. When the project launches, there will be the initial value of 0 on the button. Every click will increase the value by one.

The appearance of the project at the start: .

After the first click: .

After three clicks: .

App.js code:

```
import React, { useState } from "react";
import "./styles.css";

function Button(props) {
  /*
    Created a state variable
    Indicated the function name for changing the state
    The function body for changing the currentValue
    will be automatically created by React
  */

  const [currentValue, setCurrentValue] =
    useState(props.startVal);
```

```
// click handler
function btClick() {
  /*
    When clicked, increase the current value by one
  */
  setCurrentValue(currentValue + 1);
}

return <button onClick={btClick}>{currentValue}</button>;
}

export default function App() {
  return (
    <>
      <Button startVal={0} />
    </>
  );
}
```

To set the hook, we use the familiar `useState` function.

```
const [currentValue, setCurrentValue] =
  useState(props.startVal);
```

In this case, the access to the state value is written into the `currentValue` variable. The reference to the function for changing the state is in the `setCurrentValue`.

The value on the button increases upon click. For this we call this inside the click handler:

```
setCurrentValue(currentValue + 1);
```

We add one to the current value and pass a new value to `setCurrentValue`.

- [Link](#) to the project code:

Let's develop one more project to consolidate the information about hooks. The appearance of the project:



Figure 8

The background color of `div` changes upon button click. If we click on the red button, the result will be as follows:



Figure 9

If the green button is clicked, the appearance changes:



Figure 10

The interface of our app has three buttons and `div` whose background color changes. Where should we store the state

variable? In the buttons or in the [div](#)? It should be stored in the app component. The function that changes state should be there too. We have already done it in the class component code.

App.js code:

```
import React, { useState } from "react";
import "./styles.css";

function Button(props) {
  const handlerClick = () => {
    props.onClickAct(props.bkColor);
  };
  return (
    <button
      className="Button"
      onClick={handlerClick}
      style={{ backgroundColor: props.bkColor }}>
      {props.text}
    </button>
  );
}

function DisplayBlock(props) {
  return (
    <div class="DisplayBlock"
      style={{ backgroundColor: props.bkColor }}>
      Some text
    </div>
  );
}

export default function App() {
  /*
    Configure the initial state
    displayBkColor = white
  */
}
```

```

const [displayBkColor,
      setDisplayBkColor] = useState("white");

/*
  Create a function to call state change.
  We will call it from the Button component.
  The reference to it will be passed through
  onClickAct
*/
const stateFunc = (newBkColor) => {
  setDisplayBkColor(newBkColor);
};
return (
  <>
    <Button bkColor="red" text="Red"
      onClickAct={stateFunc} />
    <Button bkColor="green" text="Green"
      onClickAct={stateFunc} />
    <Button bkColor="yellow" text="Yellow"
      onClickAct={stateFunc} />
    <DisplayBlock bkColor={displayBkColor} />
  </>
);
}

```

Let's briefly analyze the application code. The code of the [App](#) component:

```

export default function App() {
  /*
    Configure the initial state
    displayBkColor = white
  */
  const [displayBkColor, setDisplayBkColor] =
    useState("white");

```

```

/*
  Create a function to call state change.
  We will call it from the Button component.
  The reference to it will be passed through
  onClickAct
*/
const stateFunc = (newBkColor) => {
  setDisplayBkColor(newBkColor);
};

return (
  <>
    <Button bkColor="red" text="Red"
      onClickAct={stateFunc} />
    <Button bkColor="green" text="Green"
      onClickAct={stateFunc} />
    <Button bkColor="yellow" text="Yellow"
      onClickAct={stateFunc}/>
    <DisplayBlock bkColor={displayBkColor} />
  </>
);
}

```

We hooked the state and created the `stateFunc` function to update the state. This function takes one value at the input: new background color for `div`. Pass the reference to `stateFunc` to the class of the `Button` component through the `onClickAct` attribute.

The variable containing state is passed to the `DisplayBlock` component.

```

function Button(props) {
  const handlerClick = () => {
    props.onClickAct(props.bkColor);
  };
}

```

```

return (
  <button
    className="Button"
    onClick={handlerClick}
    style={{ backgroundColor: props.bkColor }}
  >
    {props.text}
  </button>
);
}

```

In the code of the `Button` component inside the click handler, call the state update function from the `App` component through the `onClickAct` attribute. Pass the background color this button is responsible for as a parameter.

```

function DisplayBlock(props) {
  return (
    <div class="DisplayBlock"
      style={{ backgroundColor: props.bkColor }}>
      Some text
    </div>
  );
}

```

The code of the `DisplayBlock` displays `div` with the background color which is now in the state. When the value updates, the background color of `div` redraws.

The `style.css` file sets styles for `Button` and `DisplayBlock`.

Carefully analyze the example code before you do your homework.

- [Link](#) to the project code:

Homework

1. Create an app Quote of the Day. When a button is clicked, a quote of the day and info about the author appear. When another button is clicked, the quote hides. Use class components, props, state, and styles.
2. Create an app Magic 8-Ball. Use functional components. You can find the description of the ball [here](#). Use state, props, and styles.
3. Create a functional component to generate random numbers. The start and end values of the range are set through [props](#). Clicking on the component generates a new number.
4. Create a functional component Traffic Light. It should display available combinations of colors. Use styles, props, and state.
5. Implement Task 4 through class components.



Lesson 6

React: Advanced Techniques

© STEP IT Academy, www.itstep.org.

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.