

**top**

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# Создание веб-приложений с использованием Angular & React

# Урок № 6

React:  
расширенные  
приемы

## Содержание

Использование деструктуризации .....	3
Значения по умолчанию для props.....	8
Стили и компоненты.....	11
Состояние (State).....	15
Хуки .....	29
Домашнее задание.....	43

# Использование деструктуризации

Во время изучения JavaScript вам встречался синтаксис деструктуризации. Это возможность присвоения массива или объекта набору переменных, разбив его на набор частей. Типичная форма синтаксиса выглядит так:

```
let [список переменных] = имя массива
```

Например,

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

В коде значения нулевого элемента было присвоено **a**, значение первого элемента **b**, значение второго элемента **c**. Деструктуризация идет слева направо. Это означает, что значение нулевого элемента массива попадает в самый левый элемент списка инициализации. С массивом справа от знака равенства ничего не происходит. Ещё пример:

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

Тот же массив `arr`, но мы пропускаем самый левый элемент при деструктуризации. В результате этого, значение нулевого элемента массива игнорируется, и мы начинаем сразу с первого, чьё значение попадает в `d`.

При использовании деструктуризации можно использовать оператор «...».

Он называется `spread` оператор. Если он указан это означает, что в переменную после него нужно записать все оставшиеся элементы массива справа от знака равенства. Этот оператор будет самым правым в списке переменных. Рассмотрим на примере:

```
let arr = [7, 88, -3];

/*
  f = 7
  arr2 = [88, -3]
*/

let [f, ...arr2] = arr;
```

В `f` записывается значение нулевого элемента, потом идет конструкция `...arr2`. Это значит, что мы создаём переменную `arr2` и просим записать в неё все оставшиеся элементы массива `arr`. Переменная `arr2` будет массивом с двумя элементами `88` и `-3`.

Деструктуризация используется и при работе с объектами. При работе с объектами нужно использовать `{}` вместо `[]`.

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

Мы создали объект `obj`. В нём два свойства `name` и `lastName`. Мы используем синтаксис деструктуризации для извлечения этих свойств в одноименные переменные `name` и `lastName`. При деструктуризации объекта, в левой части вы должны указывать имена свойств, которые хотите получить из него. Например:

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

Мы извлекаем только `name`. Если вы хотите, чтобы ваши переменные назывались отлично от свойств объекта, можно использовать такой синтаксис:

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

Значение свойства `name` будет записано в переменную `n`. Значение свойства `lastName` будет записано в `l`.

Зачем мы вспоминали деструктуризацию? Дело в том, что этот синтаксис часто используется при создании React приложений. Мы тоже это будем делать и начнем прямо сейчас.

Создадим приложение, отображающее имя и фамилию писателя. Эти параметры мы передадим через `props`. Файл `index.js`:

```
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
);
```

Тут нет ничего нового. Мы передаём значения свойств `name` и `lastName`. Откроем *App.js*:

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

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

Мы получили объект `props` и используя синтаксис деструктуризации записали значение свойства `name` в переменную `name`, значение свойства `lastName` в переменную `lastName`.

Мы могли бы переименовать переменные, куда будут записаны значения.

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

В этом случае мы используем переменные `n` и `l`. Если этот синтаксис вам был раньше не знаком, попробуйте его на практике. Он может выглядеть немного странно, но вы к нему быстро привыкнете.

Ссылка на проект: <https://codesandbox.io/s/destructuring-vjliz>.

# Значения по умолчанию для props

В наших примерах мы задавали значения для всех **props**. А что будет, если пользователь нашего компонента задаст значения двум из четырёх свойств? Что произойдет в коде нашего компонента при попытке обратиться к свойству, чьё значение не было задано?

Ответ очевиден: ничего хорошего. Значение свойства будет равно **undefined**. Что же делать? Нам нужно задать значения по умолчанию для наших свойств. Они будут использованы, когда пользователь не укажет своё значение.

Для задания значений по умолчанию нам нужно использовать свойство **defaultProps**.

Снова обратимся к проекту о писателях. Не зададим значение для фамилии внутри файла *index.js*.

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

Мы указали значение только для **name**. Внутри файла *App.js* задаём значения по умолчанию.

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



```

    <h1>Information about writer:</h1>
    <h2>{props.name}</h2>
    <h2>{props.lastName}</h2>
  </div>
);
}

/*
  Инициализируем значения по умолчанию.
  Используем синтаксис создания объекта
*/

App.defaultProps = { name: "William",
                      lastName: "Shakespeare" };

```

Мы задали значение по умолчанию для компонента `App`. Значение по умолчанию для `name` «William». Значение для `lastName` «Shakespeare». Пользователь компонента передал нам значение только для `name`. Это значит, что значение для `lastName` будет задано по умолчанию. И оно будет равняться «Shakespeare».

Синтаксис для задания свойств по умолчанию

```
Имя_компонента.defaultProps =
```

Также можно задать значение для каждого свойства отдельно:

```

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

```

```
        </div>
    );
}
App.defaultProps.name = "Ernest";
App.defaultProps.lastName = "Hemingway";
```

Ссылка на проект: <https://codesandbox.io/s/defaultprops-xqtgn>.

Изученный нами синтаксис применяется также и для классовых компонент.

# Стили и компоненты

Мы уже умеем задавать стили в React используя атрибут `className`. Вы также можете устанавливать стили непосредственно внутри самого компонента. Для этого используется атрибут `style`. Создадим проект, где мы это будем делать. Содержимое файла *App.js*:

```
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 />
    </div>
  );
}
```

```

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

```

Результат работы проекта:

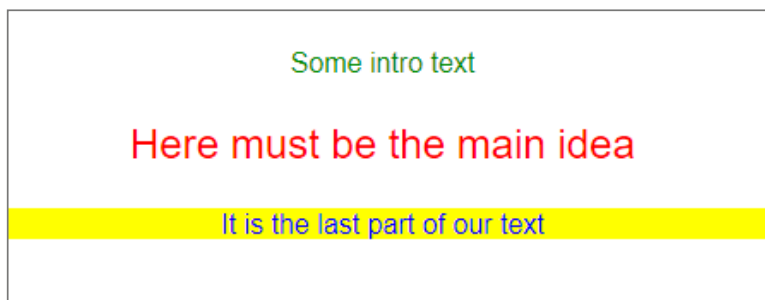


Рисунок 1

Мы создали в нашем коде три дополнительных функциональных компонента. Каждый из них отвечает за свою часть текста. **Intro** — введение, **Main** — основной текст, **End** — завершающая часть. Для каждого из компонентов мы задали стили. Например, для **Intro** это выглядит так:

```

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

```

Для задания стилей мы использовали атрибут **style**. Почему мы используем такое количество фигурных скобок? Первая пара **{}** используется, потому что мы задаём динамическое значение. Вторая пара скобок

используется, потому что для задания стилей нужно создать объект, содержащий настройки стилей. В примере выше создан объект с одним свойством `color`. А вот для компонента `Main` мы создаём объект с двумя свойствами `color` и `fontSize`:

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

Для компонента `End` мы задаём два свойства: `color` и `backgroundColor`.

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

Ссылка на проект: <https://codesandbox.io/s/functionstyle-xz0ke>.

Задания стилей для классовых компонентов выполняется по тому же принципу. Создадим классový компонент, отображающий текст.

```
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>
  );
}
}

```

В коде задано значение для размера шрифта. Также в коде выбирается случайным образом цвет шрифта. Он может быть красным или зелёным в зависимости от значения, которое вернет `Math.random`.

Ссылка на проект: <https://codesandbox.io/s/classstyle-c15i7>.

Когда может понадобиться задание стилей непосредственно из кода? Например, для динамического изменения стилей в зависимости от некоторого условия.

# Состояние (State)

**Одна из главных концепций React** — это **state** (состояние). **State** позволяет хранить внутреннее состояние компонента. Внутри состояния содержатся важные характеристики компонента. Значения из состояния используются при отображении компонента. Если характеристика не участвует в рендере компонента, её не нужно помещать внутрь состояния.

Знакомство с понятием **state** мы начнем с классовых компонент.

Создадим проект «Счетчик нажатий». В интерфейсе этого проекта будет только одна кнопка. При старте приложения на ней будет выведено начальное значение **0**. При каждом клике по кнопке мы будем увеличивать значение счетчика. После первого клика на кнопке будет значение **1**, после второго **2** и так далее.

При старте приложения мы увидим: .

После первого клика: .

После пяти кликов: .

Начнем знакомство с кодом проекта. Файл *App.js*:

```
import React from "react";

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

```

        currentValue: 0
    };
}

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

export default class App extends React.Component {
    render() {
        return (
            <>
                <Counter />
            </>
        );
    }
}

```

У нас есть два компонента. Уже знакомый нам [App](#). В нём всё достаточно стандартно. Вторым компонентом [Counter](#) — это кнопка, по которой мы будем кликать. Разберем код этого компонента по частям. Начнем с конструктора:

```

constructor(props) {
    super(props);
    this.state = {

```



```

        currentValue: 0
    };
}

```

Конструктор принимает в качестве параметра `props`. Первой строкой `super(props)`; конструктор нашего компонента вызывает конструктор родительского класса. Мы обязаны это сделать для корректной работы.

Следующей строкой мы задаём значение для состояния компонента. Фактически, когда мы говорим о состоянии компонента, мы говорим об объекте, содержащем набор свойств. Для доступа к объекту состояния внутри конструктора надо использовать `this`. Состояние компонента — это свойство под названием `state`. Полная конструкция для задания начальных значений состоянию выглядит так:

```

    :
    this.state = {
        currentValue: 0
    };

```

Фигурные скобки используются, потому что `state` это объект, который может содержать много свойств. В нашем случае у нашего состояния есть одна характеристика — это `currentValue`. Мы задали ей значение 0.

Теперь перейдем к методу `render`.

```

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

```

```

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

```

Разбираем по частям, но начнем с конца.

```

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

```

В `return` мы описали внешний вид нашего компонента. Указали, что при клике нужно вызывать обработчик `handlerClick`. Задали текст на кнопке. Текст на кнопке — это характеристика `currentValue` нашего состояния. Помните выше мы говорили, что характеристики состояния должны участвовать в рендере компонента? Наш код подтверждает это. Для доступа к чтению значения `currentValue` нужно использовать `this.state.currentValue`.

При старте приложения значение `currentValue` равняется `0`. Теперь погрузимся в код обработчика клика:

```

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

```

Выглядит немного зловеще, но не всё так страшно. Наш обработчик — это стрелочная функция. С этим подходом мы уже встречались ранее.

Зададимся вопросом, что должно происходить в коде обработчика? Мы должны к текущему значению прибавить единицу и обновить текст на кнопке. Наше текущее значение это переменная состояния под названием `currentValue`. Значит нам нужно увеличить значение этой переменной на единицу. Для обновления значения переменной состояния нужно вызывать специальный метод `setState`. Он обновляет указанные характеристики состояния. Если какая-то характеристика не изменилась, её не нужно задавать заново. Она автоматически сохранит своё значение. В нашем случае мы должны задать новое значение для `currentValue`.

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

В качестве параметра нам нужно передать объект, содержащий измененное состояние.

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

Для установки нового значения мы берем текущее значение и увеличиваем его на единицу. Больше никакого кода в нашем обработчике нет! А где же обновление интерфейса?

Здесь за нас работает магия React. Как только мы изменили состояние, React обновил самостоятельно только те части страницы, где эта измененная величина использовалась. Это значит, что текст на кнопке изменился автоматически. Этот механизм будет работать только, если вы используете `setState`.

Обратите внимание, что здесь неявно задействуется Virtual DOM. Изменения происходят сначала в нём и только потом React проверяет отличия между DOM страницы и Virtual DOM. Найденные изменения будут внесены в DOM страницы. При этом страница не будет подвергаться целиком перерисовке. Изменяться только обновленные части.

Ссылка на проект: <https://codesandbox.io/s/classstate-14b5p>.

Когда говорят о **props** часто употребляют термин **immutable**. Так хотят подчеркнуть, что **props** это неизменяемая величина. Она приходит в компонент и содержит начальные настройки. Их не надо менять. Когда говорят о **state** употребляют другой термин **mutable**. Тем самым подчеркивают, что состояние изменяемая величина и его НУЖНО менять.

Создадим новую версию проекта. В новой версии наш компонент **Counter** будет принимать начальное значение и величину приращения извне. Для задания свойств мы будем использовать **props**. Код *App.js*:

```
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} />
      </>
    );
  }
}

```

Какие есть изменения в новой версии? Мы используем **props** для задания **startValue** (начальное значение) и **incValue** (величина приращения).

В коде компонента у нас изменилось значение содержимое **setState**. Теперь мы увеличиваем текущее значение на величину **incValue**.

```

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

```

Других изменений в коде нет.

Старт приложения 0.

После первого клика: 3.

После второго клика: 6.

Мы надеемся, что механика работы обоих проектов вам понятна.

Ссылка на код проекта: <https://codesandbox.io/s/class-state2-8lhzf>.

И ещё раз модифицируем наш пример. В предыдущих примерах мы меняли текст на самой кнопке. В новом примере по нажатию на кнопку текст будет меняться в `div`. Вот так будет выглядеть наше приложение:

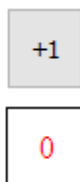


Рисунок 2

После первого нажатия:

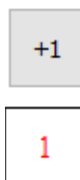


Рисунок 3

Для решения задачи мы создадим два компонента. Один для кнопки, один для `div`. Главный вопрос, который нужно решить, где мы будем хранить состояние? Ведь

к нему нужно иметь доступ и в кнопке, и в `div`. Если у вас стоит такая проблема, лучше всего хранить состояние в классе, который имеет доступ к кнопке и `div`.

Вся логика по работе с состоянием должна содержаться в нём. В нашем случае идеальным кандидатом для хранения состояния является класс `App`.

Посмотрим код файла `App.js`. Начнем с полного кода. Просмотрите его сверху вниз.

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

```

Теперь давайте разбирать код по частям. Начнем с класса **App**. В нём мы сохраним всю логику по работе с состоянием.

```

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}/>
  </>
);
}

```

Тело конструктора не изменилось. Главные изменения в методе `render`

```

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

```

Наш обработчик теперь получает один аргумент: величину приращения `incValue`. Обратите внимание, что код по работе с состоянием находится в классе `App`, а не в классе кнопки. Это сделано, потому что код изменения состояния должен находиться в том классе, где это состояние объявлено.

```

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

```

`return` также выглядит иначе. Мы создаём два компонента: `Button` (кнопка) и `Display` (div для отображения). Кнопке передаём два свойства: `incButtonVal` (величина приращения) и `handlerClick` (ссылка на нашу функцию изменения состояния). `onClickAct` — это атрибут нашего класса кнопки. Именно через него мы вызовем `handlerClick` внутри класса `Button`. Компоненту `Display` мы передаём текущее значение нашего счетчика. При старте приложения оно равно 0. Теперь рассмотрим класс `Button`.

```

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

```

В классе кнопки у нас есть только метод `render`. Обработчик нажатия на кнопку `btnClick` вызывает функцию изменения состояния из класса `App`.

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

Имя этой функции находится в свойстве `OnClickAct`. В качестве параметра мы передаём ей величину приращения. Она находится в свойстве `btText`

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

В `return` мы описываем создаваемую кнопку. Задаём её внешний вид при помощи стилей из файла `style.css`. Указываем, что при клике на кнопку нужно вызвать обработчик под названием `btnClick`. Плюс перед `{this.props.btText}` позволяет отобразить «+» перед значением свойства.

Компонент `Display` реализован очень просто:

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

Каждый раз, когда мы изменяем состояние, React, автоматически обновляет текст внутри `div`.

Обязательно проработайте этот пример на практике.

Ссылка на код проекта: <https://codesandbox.io/s/class-state3-yuolm>.

Благодаря новым возможностям ES мы можем немного сократить затраты по написанию классового компонента. Вместо использования конструктора в таком стиле

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

Мы можем присвоить значение `state`, как полю класса

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

В этом случае нам не понадобится создавать конструктор и описывать его тело.

# Хуки

Мы заложили фундамент по работе с состоянием внутри классовых компонент. Как же работать с состоянием в функциональных компонентах? Для этого мы будем использовать механизм хуков.

**Хуки** — это набор полезных React функций, которые используются для решения разных проблем. Почему такое странное название? В переводе с английского языка хук — это крючок, который используется для того, чтобы зацепить или удерживать что-то. На хук в React можно смотреть, как на крючок, который зацепит решение конкретной проблемы. Наше знакомство с хуками мы начнем с хука по работе с состоянием.

Встречайте `useState` — хук состояния. Синтаксис `useState`:

```
useState(начальное_значение_для_состояния)
```

Функция `useState` возвращает массив, содержащий в нулевом элементе ссылку на значение состояния и функцию для обновления состояния в первом элементе. Для записи этих величин в переменные мы будем использовать синтаксис деструктуризации, о котором мы говорили ранее.

Пример вызова `useState` (мы создадим переменную состояния с начальным значением 0):

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

Мы вызываем `useState` и задаём начальное значение `0` для переменной состояния. Ссылка на доступ к этому значению записывается в переменную `counterVal`. В переменную `setCounterVal` — записывается ссылка на функцию для задания нового значения переменной состояния. Обратите внимание, нам не нужно будет реализовывать тело функции для обновления этой переменной состояния. Это за нас сделает `React`. Нам нужно будет вызвать эту функцию указав имя `setCounterVal` и передать в качестве параметра новое значение для состояния. Имена `counterVal` и `setCounterVal` были выбраны нами произвольно. Они могут быть любыми. Главное, чтобы они несли смысловую нагрузку. Если вы хотите хранить в состоянии несколько значений, вам нужно вызвать `useState` по разу для каждого значения. Для каждого нового вызова `useState`, пара переменных слева от знака равенства должна быть другой.

Не забудьте импортировать `useState` перед использованием! Иначе произойдет ошибка. Для использования `useState` нам нужно будет использовать именованный импорт.

Создадим первый проект по использованию состояния внутри функциональных компонент. В нашем проекте будет кнопка с начальной надписью. После клика на кнопке, текст на кнопке будет изменен.

Начальный вид приложения:

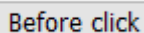


Рисунок 4

После клика:

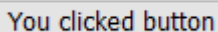


Рисунок 5

## Содержимое файла *App.js*:

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

function Button(props) {
  /*
    Установка хука состояния
    btText -будет содержать текущее значение состояния
    setBtText - функция для задания нового значения
                состоянию
    Мы будем передавать новое значение в неё.
    Реализация этой функции остаётся за React

    btText и setBtText это произвольные названия
    можно выбирать любые
  */

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

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

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

## Рассмотрим важные моменты этого кода:

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

Здесь кроме **React**, мы импортируем хук **useState**.

Хук мы устанавливаем внутри функционального компонента **Button**:

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

Начальное значение для состояния строка **Before click**. Доступ на чтение к переменной состояния записывается в **btText**. В **setBkText** записывается ссылка на функцию для изменения значения состояния. Нельзя менять состояние через **btText**. Это можно делать только через функцию обновления. В этом случае **React** гарантирует вам успешное обновление состояния, а за ним успешное обновление интерфейса. Ещё немного кода компонента:

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

Как обычно мы указываем, что реагируем на клик кнопки. Самая важная строка здесь:

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

Мы вызываем функцию обновления состояния и передаём новое значение. Сначала такой подход может выглядеть непривычным, но со временем он не будет вызывать у вас трудностей.

Ссылка на проект: <https://codesandbox.io/s/functionstate-w2uin>.

Модифицируем этот проект. Теперь мы будем хранить в состоянии не только надпись на кнопке, но



и цвет фона кнопки. После клика будем менять надпись и цвет фона.

Стартовый вид приложения:



Рисунок 6

После клика:



Рисунок 7

Код *App.js*

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

function Button() {
  /*
    Мы будем хранить в состоянии две характеристики
    надпись на кнопке и цвет фона кнопки
  */
  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 />
    </>
  );
}
```

Первое отличие по сравнению с предыдущим примером состоит в том, что мы два раза вызываем `useState`.

```
/*
  Мы будем хранить в состоянии две характеристики
  надпись на кнопке и цвет фона кнопки
*/

const [bkColor, setBkColor] = useState("yellow");
const [text, setText] = useState("Click me");
```

Мы должны это сделать, чтобы задать две характеристики нашего состояния: текст и цвет фона.

Второе отличие: в обработчике кнопки мы вызываем для обновления каждой характеристики конкретную функцию:

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

Третье отличие: при создании кнопки мы устанавливаем ей стили при помощи атрибута `style`:

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

Главный вывод этого проекта: сколько характеристик состояния нужно хранить, столько раз и вызывается `useState`.

Ссылка на проект: <https://codesandbox.io/s/funcstate-double-mmfzo>.

Запомните несколько важных правил по работе с хуками:

1. Используйте хуки только на верхнем уровне. Это означает, что надо ставить хук в начале кода функционального компонента.
2. Не используйте хуки внутри циклов, условных операторов или вложенных функций. Вместо этого всегда используйте хуки только на верхнем уровне React-функций.
3. Не вызывайте хуки из обычных функций JavaScript

Создадим третий проект по работе с хуком состояния в функциональном компоненте. Повторим наш проект кнопки-счётчика. Когда проект будет запущен на кнопке будет начальное значение **0**. При каждом клике значение будет увеличиваться на единицу.

Внешний вид проекта при старте: 0.

После одного клика: 1.

После трёх кликов: 3.

Код файла *App.js*:

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

function Button(props) {
  /*
    Создали переменную состояния счетчик
    Указали название функции для изменения состояния
    Тело функции для изменения currentValue за нас
    создаст React
  */

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

  // обработчик нажатия
  function btClick() {
    /*
      При клике увеличиваем текущее значение
      на единицу
    */
    setCurrentValue(currentValue + 1);
  }
  return <button onClick={btClick}>{currentValue}</button>;
}

export default function App() {
  return (
    <>
```

```

    <Button startVal={0} />
  </>
);
}

```

Для установки хука мы используем знакомую функцию `useState`:

```

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

```

В этом случае доступ к значению состояния записывается в переменную `currentValue`. Ссылка на функцию для изменения состояния в `setCurrentValue`.

Значение на кнопке мы увеличиваем после клика. Для этого мы вызываем внутри обработчика клика:

```

setCurrentValue(currentValue + 1);

```

К текущему значению мы добавляем единицу и передаём новое значение в `setCurrentValue`.

Ссылка на код проекта: <https://codesandbox.io/s/functionstate2-xobdo>.

Для закрепления хука состояния создадим ещё один проект. Внешний вид проекта:



Рисунок 4

После нажатия на кнопку цвет фона `div` будет меняться. Если мы нажмем на красную кнопку результат будет такой:



Рисунок 5

После нажатия на зелёную кнопку внешний вид меняется:



Рисунок 6

В интерфейсе нашего приложения три кнопки и `div`, чей цвет фона мы меняем. Где хранить переменную состояния? В кнопках или в `div`? Надо хранить в компоненте приложения. Там же должна быть функция для изменения состояния. Мы уже делали так в примере на классовые компоненты.

Код файла *App.js*:

```
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() {
  /*
    Настраиваем начальное состояние
    displayBkColor = white
  */
  const [displayBkColor,
    setDisplayBkColor] = useState("white");

  /*
    Создали функцию, которая будет вызывать изменение
    состояния
    Мы её будем вызывать из компонента Button
    Ссылку на неё мы передадим через атрибут
    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} />
  </>
);
}

```

Кратко проанализируем код приложения. Код компонента App:

```

export default function App() {
  /*
    Настраиваем начальное состояние
    displayBkColor = white
  */
  const [displayBkColor, setDisplayBkColor] =
    useState("white");
  /*
    Создали функцию, которая будет вызывать изменение
    состояния
    Мы её будем вызывать из компонента Button
    Ссылку на неё мы передадим через атрибут
    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} />
  </>
);
}

```

Мы поставили хук на состояние. И создали функцию `stateFunc`, которая обновляет состояние. На входе эта функция принимает одно значение: новый цвет для фона `div`. Ссылку на `stateFunc` передали классу компонента `Button` через атрибут `onClickAct`.

Компоненту `DisplayBlock` передали переменную, содержащую состояние.

```

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

```

В коде компонента `Button` в обработчике клика мы вызываем функцию обновления состояния из компонента `App` через атрибут `onClickAct`. В качестве параметра передаём тот цвет фона, за который отвечает данная кнопка.

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

Код компонента `DisplayBlock` отображает `div` с тем цветом фона, который сейчас находится в состоянии. Когда значение обновляется, цвет фона `div` перерисовывается.

В файле `style.css` определены стили для `Button` и `DisplayBlock`.

Внимательно разберите код примера перед выполнением домашнего задания.

Ссылка на код проекта: <https://codesandbox.io/s/functionstate3-svfgw>.

# Домашнее задание

---

1. Создайте приложение «Цитата дня». По нажатию на кнопку должна появляться цитата дня с информацией об авторе. По нажатию на другую кнопку цитата должна скрываться. Используйте классовые компоненты, `props`, `state` и стили.
2. Создайте приложение Магический Шар предсказаний. Используйте функциональные компоненты. Описание работы шара доступно по [ссылке](#). Используйте состояние, `props`, стили.
3. Создайте функциональный компонент для генерации случайных чисел. Начальное и конечное значение диапазона чисел задаётся через `props`. По клику на компоненте генерируется новое число.
4. Создайте функциональный компонент Светофор. Он должен отображать различные доступные комбинации цветов. Используйте стили, `props`, состояние.
5. Реализуйте четвертое задание через классовые компоненты.