

Цели занятия

- научиться работать с событиями в React;
- разобрать принцип неизменяемости;
- начать работать с состоянием компонента и хуком `useState`;

Полезные материалы

- [Углубленное изучение JSX](#)
- [Учим `useState` на примерах](#)
- [Неизменяемость в JavaScript](#)

Обработка событий

Обработка событий в React-элементах очень похожа на обработку событий в DOM-элементах. Но есть несколько отличий:

- События в React именуются в стиле `camelCase` вместо нижнего регистра
- При использовании React обычно не нужно вызывать `addEventListener`, чтобы добавить обработчики в DOM-элемент после его создания. Вместо этого добавьте обработчик сразу после того, как элемент отрендерился.

```
function Component() {  
  const handleClick = () => {  
    console.log("Click!");  
  };  
  
  return <button onClick={handleClick}>Click me!</button>;  
}
```

Таким образом вы можете подписаться на все DOM-события, которые есть у элементов HTML. Когда вы делали это с помощью `addEventListener`, то просто указывали название события, в React используется все те же названия, но в camelCase и с приставкой `on` в начале.

Например, вместо `addEventListener('click', handleClick)` будет `onClick(handleClick)`, а вместо `addEventListener('submit', handleSubmit)` — `onSubmit(handleSubmit)`

Обратите внимание, что рекомендуется придерживаться следующего правила: событие начинается с `on`, а обработчик события с `handle`. Это позволяет с первого взгляда на код понимать для чего нужна та или иная функция в компоненте.

```
function Component() {
  const handleSubmit = () => {
    console.log("Submit!");
  };

  // Срабатывает при вводе имени
  const handleInputName = (e) => {
    const name = e.target.value;
    console.log(`Name: ${name}`);
  };

  // Срабатывает при вводе телефона
  const handleInputPhone = (e) => {
    const phone = e.target.value;
    console.log(`Phone: ${phone}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="name" onChange={handleInputName} />
      <input name="phone" onChange={handleInputPhone} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Собственные события

Для любого компонента в React мы можем добавить собственные события, которые будут возникать в тот момент, когда нужно именно нам и передавать необходимые данные. Например, давайте создадим список кнопок, который имеет событие клика только на первую кнопку:

```
function List(props) {
  const items = [1, 2, 3];

  // Создаем свойство для события
  const { onClickFirst } = props;

  const handleClickItem = (e, item) => {
    // Пишем любую логику, которая нам нужна
    if (item !== 1) return;

    // Если было указано свойство с событием, то вызываем
    // в нужный нам момент
    if (onClickFirst) onClickFirst(e);
  };

  return (
    <div>
      {items.map((item) => (
        // Обратите внимание, параметр e это само событие
        // А вторым параметром мы можем передать все что нам нужно
        <button key={item} onClick={(e) => handleClickItem(e,
item)}>
          {item}
        </button>
      ))}
    </div>
  );
};
```

```
}
```

После чего мы можем использовать это свойство как будто это обычное событие в React:

```
function App() {  
  const handleClickFirst = (e) => {  
    console.log("Кликнули на первую кнопку");  
    console.log("А вот и объект события", e);  
  };  
  
  return <List onClickFirst={handleClickFirst} />;  
}
```

Неизменяемость

Этот принцип часто используется в React-приложениях и позволяет избегать многих ошибок. Его суть в том, что вместо прямого изменения какой-то переменной в коде мы создаём новую переменную на её основе.

Например, представьте что у нас есть переменная `count` и мы увеличим её на единицу и вывести в консоль новое значение:

```
function Component() {  
  let count = 0;  
  
  const handleClick = () => {  
    count = count + 1;  
  
    console.log(`Количество ${count}`);  
  };  
  
  return <button onClick={handleClick}>Click</button>;  
}
```

И это абсолютно корректный код и всё работает как задумано. Но такой подход напрямую изменяет значение переменной и в масштабных приложениях это может приводить к разным ошибкам, а также мы теряем доступ к предыдущему значению переменной, что не очень хорошо. Поэтому с точки зрения неизменяемости правильнее было бы написать вот так:

```
function Component() {
  const count = 0;

  const handleClick = () => {
    const newCount = count + 1;

    console.log(`Количество: ${newCount}`);
    console.log(`Предыдущее: ${count}`);
  };

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

Этот принцип на самом деле гораздо сложнее и если вы хотите углубиться в эту тему, то можно начать с [этой статьи](#). Но стоит запомнить простое правило — в React не нужно изменять переменные напрямую.

Работа с состоянием

Состояние компонента это его “память”, у каждого компонента может быть несколько состояний и они уникальны для каждого компонента. Состояние компонента можно изменять и это будет приводить к автоматическому повторному рендеру компонента.

В современном React работа с состоянием происходит с помощью функции `useState`, которую можно использовать внутри любого функционального компонента следующим образом:

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

function Component() {
  // Первый аргумент это сам счетчик, а второй функция для его
  // изменения
  const [count, setCount] = useState(0);

  const handleIncrease = () => {
    // Мы могли бы попытаться сделать просто count = count + 1
    // Но React построен на принципе неизменяемости и поэтому
    // изменять
    // состояние можно только через определенную функцию setCount
    setCount((prevCount) => prevCount + 1);

    // В данном примере prevCount это предыдущее значение счетчика
    // и React всегда нам даёт доступ к предыдущему значению
    // состояния
  };

  const handleSet = () => {
    // Но можно установить и полностью новое значение
    // Если нам это необходимо в каком-то случае
    setCount(100);
  };

  return (
    <>
      <div>`Текущее количество: ${count}`</div>
      <button onClick={handleIncrease}>Увеличить на 1</button>
      <br />
      <button onClick={handleSet}>Поставить на 100</button>
    </>
  );
}
```

Это основной принцип того, как работать с состоянием в React, больше примеров будет показано на онлайн-занятии, а также можно разобрать их в отдельной статье — [Учим useState на примерах](#).

Состояние это одна из фундаментальных вещей React, которая позволяет производить любые изменения на странице и вся последующая логика приложения будет основана на работе с состоянием, поэтому крайне рекомендую уделить этому больше времени, чтобы хорошо усвоить принцип работы.