JSX

JSX

- JSX — это расширение синтаксиса JavaScript, которое чем-то похоже на XML.
- Одно из преимущества XML — сбалансированные открывающие и закрывающие теги. Благодаря этому читать большие деревья проще, чем вызовы функций или объектные литералы.
- В конечном счете, JSX делает код более удобочитаемым и упрощает определение Virtual DOM.
- Нет необходимости использовать JSX вместе с React. Можно просто использовать обычный JS. Тем не менее, мы рекомендуем использовать JSX, так как это понятный и знакомый синтаксис для определения древовидных структур с атрибутами.

ПРЕОБРАЗОВАНИЕ JSX В JS

- Поскольку браузеры не умеют интерпретировать JSX, его необходимо преобразовать в JS.
- Популярные инструменты: babelify с предустановкой react, webpack с предзагрузчиком react (его мы и будем использовать)
- Преобразования JSX в JS:

```
// Input (JSX):
var app = <Nav color="blue" />;
// Output (JS):
var app = React.createElement("Nav",
            {color:"blue"});
```

```
// Input (JSX):
var person = <Person
       name={window.isLoggedIn ? window.name : ''} />;
// Output (JS):
var person = React.createElement(
    Person,
    {name: window.isLoggedIn ? window.name : ''}
);
```

ВЫРАЖЕНИЯ JSX

- С помощью JSX можно создавать дополнительные семантические структуры: Form, Form.Row, Form.Header:

```
var App = (
  <Form>
    <Form.Header />
    <Form.Row>
      <Form.Input/>
    </Form.Row>
  </Form>
);
```

- Атрибуты:

```
<Person loggedIn={window.isLoggedIn ? true : false} />
```

- Логические атрибуты:

```
<Person loggedIn={true} />
```

- Комментарии:

```
return {/* child comment, put {} around */}
```

- Стили:

```
<Person style={{color:black}} />
```

- Настраиваемые атрибуты:

```
<div data-custom-attribute="foo" />
```

- Спец. веб-возможности:

```
<div aria-hidden={true} />
```

- Логические:

```
let loggedIn = false;
return (
  {loggedIn || <authContainer/>}
)
```

- Логические:

```
<div id={if (condition) { 'msg' }}>Hello World!</div>
```

ЧТО ВАЖНО ЗНАТЬ О JSX

- При отображении компонента можно возвращать только один узел!
- Скажем, если вам нужно вернуть список divs, вы должны включить компоненты в оболочку div, span или любого другого компонента.
- Не забывайте, что JSX конвертируется в обычный JS, а возврат двух функций не имеет смысла с точки зрения синтаксиса. И не размещайте более одного дочернего элемента в тернарной операции.
- В JSX возможно только <MyComponent />, <MyComponent> невозможен. Все теги должны быть закрытыми.

ВСТРОЕННЫЕ HTML-ТЕГИ

```
// React определяет все стандартные HTML-теги, которые
// используются на постоянной основе. Их можно рассматривать
// как любые другие компоненты React.
render((
    <div>
      <button />
      <code />
      <input />
      <label />
      <p />
      <pre />
      <select />
      <table />
      <ul />
    </div>
  ),
  document.getElementById('app')
);
```

УСЛОВИЯ ИСПОЛЬЗОВАНИЯ HTML-ТЕГОВ

```
// Они отображаются, как обычно, за исключением свойства "foo",
// которое не является узнаваемым свойством кнопки.
// В этом случае в консоли появится предупреждение.
render((
      <button foo="bar">
         My Button
      </button>
   ),
   document.getElementById('app')
);

// Выдается ошибка "ReferenceError", так как
// имена тегов чувствительны к регистру. Это противоречит
// обязательному условию написания имен HTML-тегов в нижнем регистре.
render(
   <Button />,
   document.getElementById('app')
);
```

ИНКАПСУЛЯЦИЯ HTML

Вместо печати сложной разметки, мы просто используем свой пользовательский тег:

```
class MyComponent extends Component {
  render() {
    // Все компоненты имеют метод "render()", который
     // возвращает определенную разметку JSX. В этом случае MyComponent
     // инкапсулирует большую HTML-структуру.
    return (
      <section>
        <h1>My Component</h1>
        <p>Content in my component...</p>
      </section>
    );
  }
}
```

ВЛОЖЕННЫЕ ЭЛЕМЕНТЫ

```
export default class MySection
extends Component {
  render() {

render((
      <section>
        <h2>My Section</h2>
        <button>
          MyButton Text
        </button>
      </section>

  ),
  document.getElementById('app')
);
```

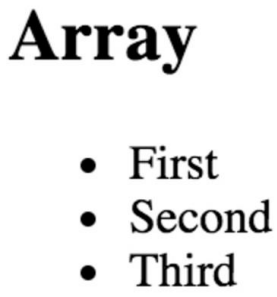
```
export default class MyButton
    extends Component {
  render() {
    return (
      <button>
          MyButton Text
      </button>
    );
  }
}
```

ДИНАМИЧЕСКИЕ ЗНАЧЕНИЯ СВОЙСТВ И ТЕКСТ


```
const enabled = false;
const text = 'A Button';
const placeholder = 'input value...';
const size = 50;
render((
    <section>
      <button disabled={!enabled}>{text}</button>
      <input placeholder={placeholder} size={size} />
    </section>
  ),
  document.getElementById('app')
);
```

МЭППИНГ МАССИВА В HTML

```
// Массив, который мы хотим отобразить в виде списка...
const array = [ 'First', 'Second', 'Third' ];
render((
  <section>
    <h1>Array</h1>

     { /* Мэппит массив в array в массив <li>.
      Обратите внимание на свойство "key" в "<li>".
      Оно необходимо для повышения производительности,
            и React выдаст предупреждение в случае его отсутствия. */ }
    <ul>
       {array.map((i,idx) => (
        <li key={idx}>{i}</li>
       ))}
    </ul>
  </section>
), document.getElementById('app') );
```



ДЕКЛАРИРОВАНИЕ ОБРАБОТЧИКОВ СОБЫТИЙ

```
export default class MyButton extends Component {
  // Обработчик событий нажатия кнопки мыши: здесь
  // ничего не происходит, кроме регистрации события.
  onClick() {
    console.log('clicked');
  }

  // Отображает элемент "<button>" с
  // обработчиком событий "onClick", установленным на
  //  метод "onClick()" этого компонента.
  render() {
    return (
      <button onClick={this.onClick}>
          {this.props.children}
      </button>
    );
  }
}
```

ССЫЛКА НА ОБЪЕКТ, В КОТОРОМ ПРОИЗОШЛО СОБЫТИЕ

```
export default class MyInput extends Component {
  // Запускается, когда значение вводимого текста изменяется...
  onChange(event) {
        console.log(event.currentTarget.value);
        console.log(this.value);
        // оба выведут текущее значение <input>
  }
  // Запускается, когда значение вводимого текста теряет фокус...
  onBlur() { console.log('blured'); }

  // Элементы JSX могут иметь столько свойств
  // обработчика событий, сколько необходимо.
  render() {
    return (
      <input onChange={this.onChange} onBlur={this.onBlur} />
    );
  }
}
```

и **event.currentTarget**, и **this** ссылаются на <input>, в котором произошло событие

Hel

тут объект-событие передается неявно

ССЫЛКА НА ОБЪЕКТ, В КОТОРОМ ПРОИЗОШЛО СОБЫТИЕ

```
export default class MyInput extends Component {
  // Запускается, когда значение вводимого текста изменяется...
  onChange(event) {
            console.log(event.currentTarget.value);
            console.log(this);
  }
  // Запускается, когда значение вводимого текста теряет фокус...
  onBlur() { console.log('blured'); }

  // Элементы JSX могут иметь столько свойств
  // обработчика событий, сколько необходимо.
  render() {
    return (
      <input onChange={this.onChange.bind(this)} onBlur={this.onBlur} />
    );
  }
}
```

event по-прежнему указывает на событие, но **this** указывает на объект MyInput, в котором выполняется метод

используем bind(this), чтобы контекст не терялся

ССЫЛКА НА ОБЪЕКТ, В КОТОРОМ ПРОИЗОШЛО СОБЫТИЕ

```
export default class MyInput extends Component {
  // Запускается, когда значение вводимого текста изменяется...
  onChange(event) {
            console.log(event.currentTarget.value);
            console.log(this);
             // оба выведут текущее значение <input>
  }
  // Запускается, когда значение вводимого текста теряет фокус...
  onBlur() { console.log('blured'); }

  // Элементы JSX могут иметь столько свойств
  // обработчика событий, сколько необходимо.
  render() {
    return (
      <input onChange={e=>this.onChange(e)} onBlur={this.onBlur} />
    );
  }
}
```

event.currentTarget указывает на <input>, а **this** - на объект компонента

явная передача объекта события

ВСТРАИВАЕМЫЕ ОБРАБОТЧИКИ СОБЫТИЙ

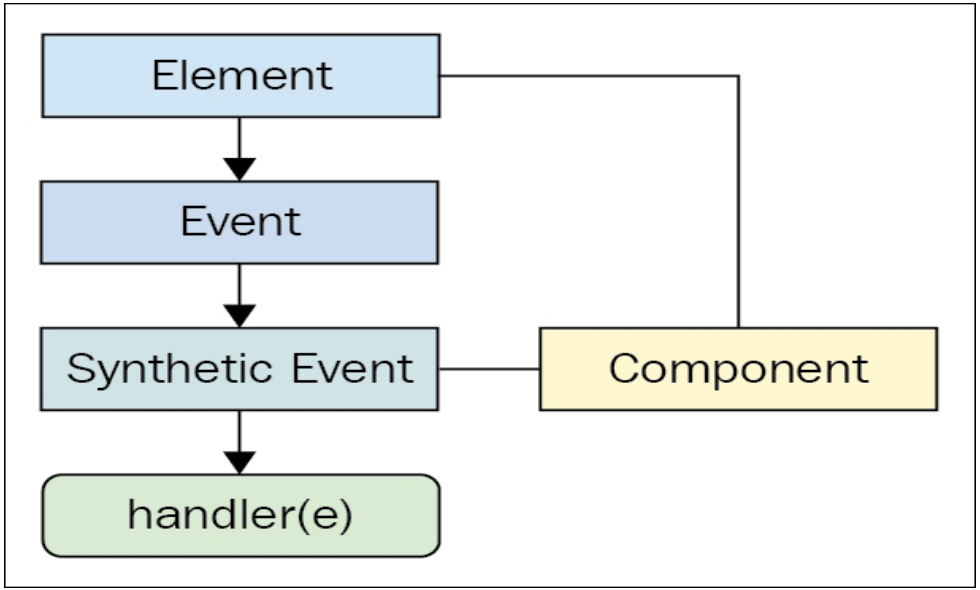
```
export default class MyButton extends Component {

  // Отображает элемент button с обработчиком "onClick()".
  // Эта функция декларируется вместе JSX и
  // используется в сценариях, когда необходимо
  // вызвать другую функцию.
  render() {
    return (
      <button
        onClick={e => console.log('clicked', e)}
       >
          {this.props.children}
      </button>
    );
  }
}
```

REACT: СОБЫТИЯ

- SyntheticEvent
- Обработчики событий будут передаваемыми экземплярами SyntheticEvent, кросс-браузерной оболочки для нативных событий браузера.
- Эта оболочка имеет такой же интерфейс, как и нативное событие браузера, включая stopPropagation() и preventDefault(), но при этом события работают одинаково для всех браузеров.
- Если вам по каким-то причинам необходимо базовое событие браузера, для его получения можно просто воспользоваться атрибутом nativeEvent.

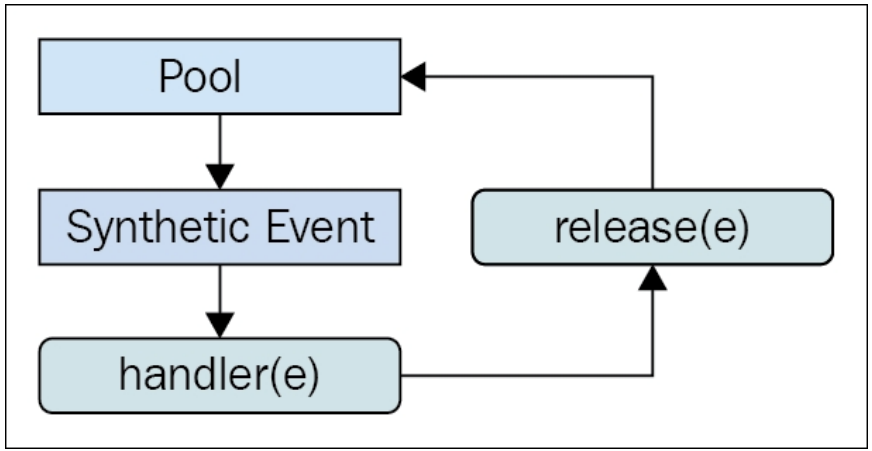
ОБЪЕКТЫ – ИСКУССТВЕННЫЕ СОБЫТИЯ

Когда вы прикрепляете функцию обработчика событий к элементу DOM с помощью нативной функции addEventListener(), при обратном вызове ей будет передан аргумент события. Функции обработчика событий в React также передаются аргументу события, но это будет не стандартный экземпляр События. Этот экземпляр называется **SyntheticEvent** и представляет собой простую оболочку для экземпляров нативных событий.



ПУЛИНГ СОБЫТИЙ

Современные приложения реагируют на множество событий, даже если обработчики событий ничего с ними не делают. Это может быть проблемой, если React постоянно должен создавать новые экземпляры искусственных событий. В React эта проблема решается созданием **пула искусственных экземпляров**. Как только возникает какое-то событие, оно берет экземпляр из пула и передает его свойства. Когда обработчик событий заканчивает обработку, экземпляр искусственного события возвращается обратно в пул, как показано ниже:



ПРОБЛЕМА С ПУЛИНГОМ

```
export default class MyButton extends Component {
  onClick(e) {
    console.log('clicked', e.currentTarget.style); // Этот способ отлично работает.
    fetchData().then(() => {
      // Попытка асинхронно перейти к "currentTarget"
       // оказывается неудачной, так как все его свойства
       // были обнулены для того, чтобы экземпляр
       // можно было использовать повторно.
      console.log('callback', e.currentTarget.style);
    });
  }
  render() {
    return (
      <button onClick={this.onClick}> {this.props.children} </button>
    );}
}
```

let style = e.currentTarget.style;

```
function fetchData() {
  return new Promise(
  (resolve) => {
    setTimeout(() => {
      resolve();
    }, 1000);
  });
}
```

ПРАКТИКА

Блок 1.
Задание 2.
JSX.