

2. JSX — язык React

Содержание урока

- Обзор;
- Элементы;
- Выражения;
- Атрибуты;
- Экранирование;
- Самозакрывающиеся теги;
- Компоненты, элементы и вложенность;
- Условный рендеринг;
- Списки;
- Обращение к компоненту через точку;
- Рендер строк и чисел;
- Фрагменты;
- Комментарии;
- Подведём итоги.

Обзор

Привет! 🙌 В этом уроке мы рассмотрим уникальный синтаксис для создания и управления разметкой в React называемый `JSX`, а также его особенности, подводные камни, на которые лучше не попадать, а также различные способы применения и композиции. 🎨

Элементы

В прошлом уроке мы познакомились с базовым типом `ReactElement`, простейшим строительным блоком React, описывающим будущий HTML-элемент. Для создания экземпляра `ReactElement` достаточно вызвать функцию `React.createElement`.

🖥️ Пример кода 2.1:

```

1  import React from 'react';
2  import ReactDOM from 'react-dom';
3
4  const linkToGoogle = React.createElement(
5    'a',
6    {
7      href: 'https://www.google.com'
8    },
9    'Google!'
10 );
11
12 ReactDOM.render(linkToGoogle, document.getElementById('app'))

```

Давай изучим этот пример построчно:

- **Строка 1**: импортируем пакет `react`. Объект, привязанный к переменной `React` содержит различные функции для создания и управления сущностями React. `createElement` — одна из них. Эта функция принимает три аргумента при вызове:
 1. Тип будущего элемента (может быть любым HTML-типом, например: `section` или `span`);
 2. Объект с описанием атрибутов будущего элемента;
 3. Контент будущего элемента: то, что находится между открывающимся и закрывающимся тегами.
- **Строка 2**: импортируем пакет `react-dom`, позволяющий отрендерить приложение в окружении браузера.
- **Строка 4**: создаём экземпляр React-элемента, вызвав функцию `React.createElement`, и привязав результат вызова к идентификатору `linkToGoogle`.
- **Строка 12**: вызываем метод `render` пакета `ReactDOM`, передав первым аргументом новосозданный React-элемент и описав DOM-селектором во втором аргументе точку в `index.html`, к которой нужно присоединить приложение.

Новосозданный экземпляр ссылки, ведущей на веб-сайт Google — это реальный пример одного элемента, используемого в приложении. 👤 Подобных элементов в реальных приложениях — тысячи. 🧑🧑 И хоть в данном подходе создания React-элемента нет ничего плохого, создав идентичный элемент с помощью синтаксиса `JSX`, можно получить большой бонус к читаемости кода, красоте и компактности: 🦄

 Пример кода 2.2:

```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const linkToGoogle = <a href='https://www.google.com'>Google!</a>;
5
6 ReactDOM.render(linkToGoogle, document.getElementById( 'app' ))

```

! Важно:

В примере кода `2.2` не использовано ни одного метода React напрямую, но в каждом модуле с использованием `JSX`, присутствие импорта React в области видимости обязательно.

Этот забавный синтаксис с тегом внутри файла JavaScript — не строка и не HTML. `JSX` — синтаксическое расширение ECMAScript. Следуя этому синтаксису, мы явно описываем то, что и как хотим увидеть на странице.

👩 Как бы шутка:

Если ты знаком с `Ember` или `Backbone`, можешь думать о `JSX` как о библиотеке-шаблонизаторе на стероидах, обладающей потенциалом языка JavaScript в полной мере.

Вызывая `ReactDOM.render`, мы передаём содержимое нашего приложения `первым` аргументом, и селектор элемента HTML `вторым`. Этот целевой HTML-элемент станет «корнем» всего приложения.

Ниже ты можешь наблюдать HTML-пример с целевым элементом с `id="app"`, в котором и «поселится» твоё приложение.

💻 Пример кода 2.3:

```

1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-
6       scale=1">
7     <title>Lectrum React course</title>
8   </head>
9   <body>
10    <div id="app"></div>
11    <script src="app.js"></script>
12  </body>
</html>

```

! Важно:

Корневым элементом приложения (например, элемент с `id="app"`) всегда должен быть непосредственный ребёнок HTML-тега `<body>`. Рендерить приложение напрямую в `<body>`, передав вторым аргументом `ReactDOM.render` селектор `document.querySelector('body')` — небезопасно. Если внутри части приложения, подконтрольного React, прокрадётся инородная сущность, возможны непредсказуемые и нехорошие последствия. 🙈 Например, Google Font Loader прикрепляет элементы `` в `<body>` на мгновение секунды. А React хочет владеть своей областью DOM дерева на 100%, любое инородное тело — недопустимо. Это — условие для успешного и эффективного прохождения цикла `reconciliation`. Вот почему необходимо рендерить приложение в специально отведенный элемент, внутри тега `<body>`.

Итак, `JSX` — это XML-подобное синтаксическое расширение, без predetermined семантики. 🙌

Предназначение `JSX` — предоставить ясный, понятный и знакомый всем нам синтаксис для определения древовидной структуры элементов и их атрибутов с любым уровнем вложенности.

📖 Официальная цитата:

„`JSX` преподносит себя как инструмент, улучшающий процесс разработки для программиста.“

Несмотря ни на что, ты всё равно можешь создавать React-приложения без `JSX`, пользуясь низкоуровневым API наподобие `React.createElement(element, props, ...children)`, `JSX` сделает твой код более элегантным.

Это удобный подход. Некоторые люди называют такой подход «синтаксическим сахаром».



Выражения

Кроме описания элементов, их атрибутов и вложенности в `JSX` также можно описывать любое JavaScript-выражение посредством описания такового внутри открывающейся и закрывающейся фигурной скобки.

💻 Пример кода 2.4:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const user = {
5   name: 'Oscar',
6   age: '20'
7 };
8
9 const message = (
```

```

10     <p>
11         Hello, my name is { user.name }. I am { user.age }. I like code,
and magic.
12     </p>
13 );
14
15 ReactDOM.render(message, document.getElementById( 'app' ))

```

В примере кода 2.4, любознательный разработчик Оскар представляет нам себя и свои интересы. 🧐 Мы ещё встретим Оскара в этом конспекте (и не только), а пока обрати внимание на строку кода 9, где объявлены два `JSX`-выражения. Всё что находится внутри — вычисляется как обычный JavaScript, а результат этого вычисления в итоге отображается на UI.

После вычисления, выражение `{ user.name }`, преобразуется в строку `Oscar` (обращение к свойству `name` объекта `user`), а выражение `{ user.age }`, преобразуется в строку `20`.

В результате, идентификатор `message` с двумя `JSX` выражениями, произведёт следующему HTML:

```

1 <p>Hello, my name is Oscar. I am 25. I like code, and magic.</p>;

```

В самом простом виде, внутри `JSX`-выражения можно отобразить значение свойства объекта (как в примере выше) или, например, переменной. Но также внутри фигурных скобок `JSX` можно описывать и более сложные конструкции.

🖥️ Пример кода 2.5:

```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const getGreeting = (isLectrumUser) => {
5
6     return (
7         isLectrumUser
8             ? 'Welcome aboard! We missed you, bro!'
9             : 'Welcome, user!'
10    );
11 };
12
13 const greeting = <h1>{ getGreeting(true) }</h1>;
14
15 ReactDOM.render(greeting, document.getElementById( 'app' ))

```

В примере кода 2.5, `JSX` содержит вызов функции. Результат в разметке будет отличаться в зависимости от аргумента вызова. 🧑

Атрибуты

Итак, React-элементы — это описание будущих HTML-элементов.

 Пример кода 2.6:


```
1 | const linkToGoogle = <a href='https://www.google.com'>Google!</a>;
```

В примере кода 2.6 мы видим элемент `<a>`, с атрибутом `href`. Как мы уже знаем, на выходе данный элемент преобразуется в соответствующий элемент `<a>`.

```
1 | <a href="https://www.google.com">Google!</a>
```

В `JSX` можно передавать React-элементам атрибуты, которые станут атрибутами будущих HTML-элементов, но есть исключения.

Дело в том, что учитывая абстрактную природу `JSX` (это синтаксическое расширение), некоторые ключевые слова в нём использовать запрещено. Например ключевые слова `class` и `for` зарезервированы языком JavaScript, поэтому их использование в `JSX` не предусмотрено. 💀

 Официальная цитата:

„`JSX` — это JavaScript, и некоторые ключевые слова, такие как `class`, или `for` — запрещено использовать в виде атрибутов. Вместо этого, элементы React ожидают альтернативные имена атрибутов DOM наподобие `className` или `htmlFor` соответственно.“

💀 Ошибка:

```
1 | <h1 class = 'welcomeMessage'>Welcome to Lectrum!</h1>
```

✅ Хорошая практика:

```
1 | <h1 className = 'welcomeMessage'>Welcome to Lectrum!</h1>
```

💀 Ошибка:

```
1 | <label for = 'email'>Email:</label>
2 | <input type = 'text' id = 'email' />
```

✅ Хорошая практика:

```
1 <label htmlFor = 'email'>Email:</label>
2 <input type = 'text' id = 'email' />
```

Есть еще несколько интересных исключений, например, атрибут `style` принимает JavaScript объект со свойствами в формате camelCase, отличающегося от CSS-формата. Данный подход предохраняет пользователей от потенциальных `cross-site scripting` атак.

🔍 Хозяйке на заметку:

`Cross-site scripting` — тип уязвимости компьютерных систем. Эта уязвимость проявляется, когда хакер включает в HTTP-ответ часть кода, который непреднамеренно вычисляется браузером, и может навредить конечному пользователю. 🧑‍💻

Хорошо, что эти детали весьма несложные. Можно запомнить один раз и всегда использовать. 👉

💻 Пример кода 2.7:

```
1 const mottoStyle = {
2   color:    'cornflowerBlue',
3   fontSize: 24
4 };
5
6 const motto = <span style = { mottoStyle }>I learn React to become the
  best developer!</span>;
```

В примере кода выше мы используем JavaScript-выражение в `jsx`, присваивая значение идентификатора `mottoStyle` как значение атрибута `style` элемента ``.

Есть ещё приятная особенность `jsx` — булевые атрибуты со значением `true` можно сокращать до простого имени атрибута. Следование этому подходу является хорошей практикой в сообществе React-разработчиков.

❌ Плохая практика:

```
1 const checkbox = <input type = 'checkbox' checked = { true } />;
```

✅ Хорошая практика:

```
1 const checkbox = <input type = 'checkbox' checked />;
```

Иногда, «выбран» элемент `<input>` или нет, определяют данные извне. В таком случае можно следовать следующему подходу.

✅ Хорошая практика:

```

1  let status;
2
3  status = true;
4
5  const checkbox = <input type = 'checkbox' checked = { status } />;

```

Давай рассмотрим ещё несколько полезных атрибутов.

Атрибут `onChange` — слушатель события `change`. Обработчик, привязанный к данному слушателю, будет вызван каждый раз при изменении значения поля формы.

Атрибут `selected` — поддерживается элементом `<option>`. `selected` используется для обозначения `<option>`, как выбранного.

Атрибут `value` — поддерживается элементами `<input>` и `<textarea>`. Используется для задания актуального значения элемента. Этот атрибут полезен при работе с контролируемыми компонентами. При работе с неконтролируемыми компонентами — пригодится атрибут `defaultValue`. О контролируемых и о неконтролируемых элементах мы узнаем чуть-чуть дальше в этом конспекте.

Атрибут `dangerouslySetInnerHTML` — это аналог `innerHTML`. В целом, задание HTML из кода таким образом — раскованное занятие, потому данный подход подвержен `cross-site scripting` (или `xss`) атакам. 🧙‍♀️

Всё-таки задать HTML-значение напрямую из React-приложения можно с помощью атрибута `dangerouslySetInnerHTML`, передав ему объект с ключом `__html`, что напомним тебе об опасности.

❌ Плохая практика:

```

1  const Purgatory = () => {
2
3      return <h1 id = 'badPractice'></h1>;
4  }
5
6  document.getElementById('badPractice').innerHTML = 'I will never use
    innerHTML property in React';

```

✅ Хорошая практика:


```

1 function createMarkup () {
2
3     return {
4         __html: 'First &middot; Second'
5     };
6 }
7
8 function MyComponent () {
9
10    return <div dangerouslySetInnerHTML = { createMarkup() } />;
11 }

```

! Важно:

В целом атрибут `dangerouslySetInnerHTML` возник не от лёгкой жизни, поэтому рекомендуется избегать его применения, кроме случаев, когда это действительно необходимо.

А вот полный список HTML атрибутов, поддерживаемых React:

```

1 accept acceptCharset accessKey action allowFullScreen allowTransparency
alt async autoComplete autoFocus autoPlay capture cellPadding cellSpacing
challenge charSet checked cite classID className colSpan cols content
contentEditable contextMenu controls coords crossOrigin data dateTime
default defer dir disabled download draggable encType form formAction
formEncType formMethod formNoValidate formTarget frameBorder headers
height hidden high href hrefLang htmlFor httpEquiv icon id inputMode
integrity is keyParams keyType kind label lang list loop low manifest
marginHeight marginWidth max maxLength media mediaGroup method min
minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required
reversed role rowSpan rows sandbox scope scoped scrolling seamless
selected shape size sizes span spellCheck src srcDoc srcLang srcSet start
step style summary tabIndex target title type useMap value width wmode
wrap

```

Экранирование

Передав внутрь `JSX` строковый литерал, его значение будет не экранировано, поэтому ты можешь описывать `JSX` также, как ты бы описывал HTML.

 Пример кода 2.8:

```
1 const validHTML = (  
2   <span>React gives me a power to write valid HTML & JSX at once!  
3   </span>  
4 );
```

Самозакрывающиеся теги

Если `JSX` тег пуст (между открывающимся и закрывающимся тегом ничего нет), он может стать самозакрывающимся.

 Пример кода 2.9:

```
1 const boxStyle = {  
2   backgroundColor: 'DarkOrange',  
3   height:         100,  
4   width:          100  
5 };  
6  
7 const box = <div style = { boxStyle } />;
```

Компоненты, элементы и вложенность

Мы узнали немного о `JSX`, и теперь настало время для настоящего мяса. Пытливый наблюдатель мог заметить, что иногда сущности React объявляются как с большой буквы, так и с маленькой. Давай разберемся в чем разница. 🤔

React предоставляет удобное разделение между элементами и компонентами — по капитализации первой буквы. Когда имя сущности пишут с маленькой буквы, это означает, что сущность — React-элемент. Когда имя сущности пишут с большой буквы, это означает, что сущность — React-компонент.

Компонент — некоторая переиспользуемая независимая сущность React. Размышляя об эффективной компоновке UI, разработчик в первую очередь делит UI на логические компоненты. 🧑

Как мы уже знаем, компоненты бывают `функциональными` и `классовыми`.

Каждый раз, объявляя функцию с именем с большой буквы, возвращающую `JSX` ты создаешь `компонент`.

 Пример кода 2.10:

```
1 const Heading = () => {  
2  
3   return <h1>Software development success story</h1>;  
4 }
```


```

4   };
5
6   const Body = () => {
7
8       return (
9           <p>Long time ago, and far-far away, the brave hero lived in his
peaceful home. His name was Oscar. Once he woke up in the morning, and
suddenly felt the unstoppable thirst of knowledge. Immediately young
hero realized that he always wanted to build beautiful web applications
using React. So he packed his bags, said goodbye to his girlfriend and
went to a journey to Lectrum. He put a lot of effort to learn in most
efficient manner, in a while he completed a React course that was
carefully designed by Lectrum developers specifically for heroes like
Oscar. And after hard but engaging journey he found his first job as a
developer. Oscar made his dream come true, met with his family and lived
his long and happy life.</p>
10      );
11  };
12
13  const SuccessStory = () => {
14
15      return (
16          <section>
17              <Heading />
18              <Body />
19          </section>
20      );
21  };

```

В примере кода `2.10`, имя идентификатора с привязкой в виде стрелочной функции `Heading` пишется с большой буквы. Поэтому `Heading` — компонент, написанный в функциональном стиле. То-же относится к компонентам `Body` и `SuccessStory`.

Компонент `SuccessStory` объединяет компоненты `Heading` и `Body` внутри элемента `<section>`.

 Хозяйке на заметку:

Обрати внимание, что `<section>` — обычный элемент, который мы пишем его с маленькой буквы. Данный элемент распечатается на странице явно в HTML-тег `<section>`.

В таком виде формируется композиция компонентов React и их вложенность. Как видишь, следуя синтаксису `jsx`, мы имеем определённую долю прозрачности в том, что происходит в коде. Вложенность определяется явно — как мы её видим. И это очень похоже на старый добрый HTML (при доступной полной мощи JavaScript). Бонусы, несущие `jsx` играют ключевую роль в больших приложениях, что не может не радовать.



! Важно:

Всегда помни о разнице в именовании компонентов с большой буквы, а элементов — с маленькой. Если написать компонент с маленькой буквы, React его проигнорирует.

👤 Ошибка:

```
1 const wrongNamedComponent = () => {  
2  
3   return <h1>I am not a valid React component! React will ignore me!  
   </h1>;  
4 }
```

Дело в том, что React думает о данном идентификаторе как об элементе `jsx`, а не как о компоненте, и не рендерит ничего.

! Важно:

Компоненты `всегда` должны возвращать один корневой элемент.

В примере кода `2.10`, компонент `SuccessStory`, объявленный выше, содержит в себе компоненты `Heading` и `Body`, с обёрткой в виде элемента `<section>`. Если убрать обёртку `<section>` и попытаться отрендерить два компонента в виде сиблингов, то будет ошибка.

👤 Ошибка:

```
1 const SuccessStory = () => {  
2  
3   return (  
4     <Heading />  
5     <Body />  
6   ):  
7 };
```


✅ Хорошая практика:

```
1 const SuccessStory = () => {  
2  
3   return (  
4     <section>  
5       <Heading />  
6       <Body />  
7     </section>  
8   ):  
9 };
```

🔍 Хозяйке на заметку:


React обладает особым механизмом для возвращения компонентом нескольких элементов, без любой обертки, используя специальный синтаксис, называемый `фрагмент` (`fragment`). Позже в этом уроке мы рассмотрим данный синтаксис.

Для того, чтобы выразить компонент в виде `ES2015-класса`, необходимо наследовать от `React.Component`.

 Пример кода 2.11:

```
1  const Heading = () => {
2
3      return <h1>Software development success story.</h1>;
4  };
5
6  const Body = () => {
7
8      return (
9          <p>Long time ago, and far-far away, the brave hero lived in his
peaceful home. His name was Oscar. Once he woke up in the morning, and
suddenly felt the unstoppable thirst of knowledge. Immediately young
hero realized that he always wanted to build beautiful web applications
using React. So he packed his bags, said goodbye to his girlfriend and
went to a journey to Lectrum. He put a lot of effort to learn in most
efficient manner, in a while he completed a React course that was
carefully designed by Lectrum developers specifically for heroes like
Oscar. And after hard but engaging journey he found his first job as a
developer. Oscar made his dream come true, met with his family and lived
his long and happy life.</p>
10         );
11     };
12
13     export default class SuccessStory extends React.Component {
14         render () {
15             return (
16                 <section>
17                     <Heading />
18                     <Body />
19                 </section>
20             );
21         }
22     }
```

В примере кода `2.11`, мы объявили компонент `SuccessStory` как ES2015-класс. Ключ в том, чтобы наследовать от `React.Component` и иметь реализацию как минимум одного метода с именем `render` (это обязательно). Метод `render` должен возвращать валидный `JSX` или `null` — как отсутствие значения для рендера.

 Хозяйке на заметку:

Технически можно вернуть `false` как отсутствие значения для рендера, но хорошей практикой является возврат именно `null`.

Классовые компоненты обладают более широкими возможностями, в сравнении с функциональными компонентами, такими как инкапсулированное состояние и методы жизненного цикла, которые мы рассмотрим в следующих частях этого конспекта.

Условный рендеринг

React расцветает во всей красе, когда необходимо совершить рендеринг с динамическим условием. В примере ниже Оскар сможет оттянуться с друзьями после работы, только если пиццерия еще открыта и всё ещё принимает посетителей. 🍷🍕

 Пример кода 2.12:

```
1  import React, { Component } from 'react';
2
3  const isPizzaOpen = true;
4
5  export default class Pizza extends Component {
6    render () {
7
8      return isPizzaOpen
9        ? <h1>Welcome, Oscar, ho-ho-ho!</h1>
10       : <h1>It is too late already! See you tomorrow!</h1>;
11    }
12  }
```

👉 Совет бывалых:


Мы рекомендуем использовать тернарный оператор JavaScript вместо `if-else` инструкций. Тернарный оператор обладает более лаконичным синтаксисом и удобным способом вернуть `null`, когда это необходимо.

В примере кода `2.12`, компонент `Pizza` отрендерится с соответствующим сообщением-приветствием в зависимости от значения идентификатора `isPizzaOpen`.

Списки

👩 Как бы шутка:

В веб-приложениях списки также важны как банан для обезьяны. 🍌🐒

 Пример кода 2.13:

```

1  const friendsOfOscar = ['Ann', 'Crystal', 'Mike', 'Jane'];
2
3  export default class OscarFriendList extends Component {
4      render () {
5          const friendList = friendsOfOscar.map(
6              (friend, index) =>
7                  <li key = { index }>{ friend }</li>
8              );
9
10         return <ul>{ friendList }</ul>;
11     }
12 }

```

👉 Совет бывалых:

При работе со списками, лучшей альтернативой императивному циклу `for` являются декларативные методы массива, такие как `map`, `filter`, `reduce` и другие. Данные методы более читаемы и лаконичны в своей природе. Они также хорошо сочетаются с декларативным стилем React, поэтому рекомендованы как предпочтительные в переборах.

Давай разберём пример кода `2.13` пошагово:

- В 1-й строке кода объявлен массив друзей Оскара. У Оскара четыре друга; 🧑🧑🧑🧑
- В 3-й строке кода объявлен компонент `OscarFriendList` с единым корневым элементом ``;
- В 5-й строке кода использован метод массива `map` для перебора списка друзей;
- В 7-й строке кода метод `map` вернёт по элементу `` на каждую итерацию, «вытаскивая» значение каждого элемента массива и передавая это значение как содержание элемента `` в виде `JSX` выражения;
- В 7-й строке кода элементу `` также присваивается атрибут `key` со значением индекса элемента на данной итерации;
- В 10-й строке кода массив с элементами `` (на этот момент времени каждый `` содержит имя друга Оскара как контент) помещается как контент для элемента `` в виде `JSX` выражения.

Атрибут `key` обязательный к использованию каждый раз при рендеринге списка. Это условие необходимо React для правильной обработки только тех элементов списка, которые изменились, были добавлены или удалены.

🔍 Хозяйке на заметку:

В реальном приложении не рекомендуется использовать индекс элемента в массиве как значение атрибута `key`. Вместо этого хорошей практикой является присваивание `key` уникального идентификатора сущности, участвующей в переборе. Но мы будем использовать индекс дальше в примерах в виду его краткости.

А вот еще пример использования атрибута `key`, с участием методов `Array.prototype.reduce` и `Array.prototype.filter`.

 Пример кода 2.14:


```
1  const posts = [
2    {
3      title:    'How I visited my old friend.',
4      likes:    44,
5      favorite: true
6    },
7    {
8      title:    'My favorite sandwich.',
9      likes:    99,
10     favorite: true
11   },
12   {
13     title:    'When I\'ve first tried broccoli.',
14     likes:    4,
15     favorite: false
16   },
17   {
18     title:    'Checked or egg?',
19     likes:    75,
20     favorite: true
21   }
22 ];
23
24 export default class OscarPosts extends Component {
25   render () {
26     const favoritePosts = posts.filter(post => post.favorite);
27
28     const listOfPosts = favoritePosts.map((post, index) => {
29       return (<li key = { index }>
30         Title: { post.title } Likes: { post.likes }
31       </li>);
32     });
33
34     const totalLikes = posts.reduce((sum, current) => {
35       return sum + current.likes;
36     }, 0);
37
38     return (
39       <div>
40         <ul>{ listOfPosts }</ul>
41         <span>Total likes: { totalLikes }</span>
42       </div>
43     );
44   }
```


! Важно:

Каждый элемент, участвующий в переборе, должен иметь атрибут `key` с уникальным значением.

Обращение к компоненту через точку


Существует способ обращения к компоненту с помощью аксессора. Это бывает удобным при работе с единым модулем, экспортирующим группу React-компонентов.

 Пример кода 2.15:

```
1  const DaytimeCodingComponents = {
2    morning () {
3
4      return <h1>I'm coding in the morning!</h1>;
5    },
6    evening () {
7
8      return <h1>I'm coding in the evening!</h1>;
9    }
10 };
11
12 const DaytimeCoding = () => {
13
14   return <DaytimeCodingComponents.morning />;
15 };
```

Попробуй сперва подумать, что отобразится на UI при рендере компонента

`DaytimeCoding`, а потом читай дальше. 


Итак, друг, если ты подумал `I'm coding in the morning!` ты был прав, ура. 

Рендер строк и чисел

Еще одна интересная возможность, имплементированная в версии `React v16.0` — возвращать компонентом строку или число в чистом виде без любой обёртки.

 Пример кода 2.16:

```
1 export default class OscarFavoriteBook extends Component {
2   render () {
3
4     return 'The Lord of The Rings';
5   }
6 }
```

 Пример кода 2.17:

```
1 export default class OscarFavoriteNumber extends Component {
2   render () {
3
4     return 91;
5   }
6 }
```


Оба примера кода валидные. В примере кода `2.16` компонент возвращает строку, которая будет отображена на UI. То же происходит с примером кода `2.17`, только в нём возвращаемое значение — число. До версии `React v16.0` данная возможность была недоступна, и любое значение нужно было оборачивать в обёртку, например — `значение`.

Это нововведение даёт больше свободы композиции и это — хорошо. 🙌

Данная особенность также распространяется на компоненты, написанные в функциональном стиле.

Фрагменты

Вместе с обновлением `React 16.0` также появилась возможность возвращать из компонента несколько сущностей без любой обёртки (раньше так было делать нельзя). Чтобы вернуть из компонента фрагмент с несколькими сущностями, их нужно поместить в литерал массива.

 Пример кода 2.18:

```

1  import React, { Component } from 'react';
2
3  export default class OscarFavoriteAffairs extends Component {
4      render () {
5
6          return [
7              <li key = 'A'>Parties</li>,
8              <li key = 'B'>Yoga</li>,
9              <li key = 'C'>Sorcery</li>,
10             <li key = 'D'>Skateboarding</li>,
11         ];
12     }
13 }

```

С использованием синтаксиса фрагментов больше нет необходимости оборачивать возвращаемое значение компонента в ненужный элемент-обёртку.

Но в случае с фрагментом в виде литерала массива, всё равно необходимо дописывать атрибут `key` каждому элементу, что не всегда бывает удобно.

Поэтому, в версии `React v16.2` добавили улучшенный синтаксис фрагментов.

 Пример кода 2.19:

```

1  import React, { Component, Fragment } from 'react';
2
3  export default class OscarFavoriteAffairs extends Component {
4      render () {
5
6          return (
7              <Fragment>
8                  <li>Parties</li>
9                  <li>Yoga</li>
10                 <li>Sorcery</li>
11                 <li>Skateboarding</li>
12             </Fragment>
13         );
14     }
15 }

```

Функционально, пример кода `2.18` идентичен примеру кода `2.19`, при этом синтаксически код `2.19` лаконичней.

Но разработчики React пошли дальше и придумали еще более красивый синтаксис.

 Пример кода 2.20:

```

1  import React, { Component } from 'react';
2

```

```

3  export default class OscarFavoriteAffairs extends Component {
4      render () {
5
6          return (
7              <>
8                  <li>Parties</li>
9                  <li>Yoga</li>
10                 <li>Sorcery</li>
11                 <li>Skateboarding</li>
12             </>
13         );
14     }
15 }


```

Фрагменты можно использовать как в классовых компонентах, так и в функциональных.

Комментарии

В `JSX` комментарии странные. Использовать комментарии HTML-стиля невозможно, так как React думает, что это настоящие ноды DOM.

Можно использовать обычные блочные комментарии JavaScript, но с условием оборачивания их фигурными скобками `JSX`.

 Пример кода 2.21:

```

1  const Sidebar = () => {
2
3      return (
4          <section>
5              { /* Here I'll create a logo when designer will send to me
its final version */ }
6              <span>Menu</span>
7              { /*
8                  Here
9                  I'll
10                 create
11                 a list
12                 with
13                 sidebar
14                 menu
15                 items
16             */ }
17          </section>
18      );
19  };

```

Подведём итоги

Мы коснулись глубинных тонкостей синтаксиса `JSX` и узнали различные способы его применения. 🦆

Давай повторим важную новую информацию, которую ты узнал во время урока:

- `JSX` — XML-подобное синтаксическое расширение JavaScript, разработанное для улучшения опыта разработчика, его производительности и экспертизы.
- Под капотом `JSX` использует низкоуровневый API (например, `React.createElement`) для создания иммутабельных JavaScript-объектов, описывающих будущий UI с определённым набором свойств.
- В `JSX` можно объявлять JavaScript-выражения используя фигурные скобки.
- Некоторые атрибуты, имена которых совпадают с ключевыми словами, зарезервированными JavaScript использовать нельзя. Вместо них стоит использовать специальные аналоги, например `className` вместо `class`.
- `JS`-элемент может быть самозакрывающимся.
- Имена компонентов в `JSX` стоит писать с большой буквы, а имена элементов — с маленькой.
- Существует возможность получить доступ к компоненту посредством обращения через аксессор, если компонент объявлен в виде модуля.
- Компоненты и элементы можно вкладывать друг в друга на любую глубину.
- Компонент можно объявить с помощью ES2015-класса или функции.
- Многострочный `JSX` можно обернуть в круглые скобки для улучшения читаемости.
- При рендеринге списков всегда нужно добавлять атрибут `key` с уникальным значением каждому элементу.
- Каждый компонент должен возвращать единый корневой элемент или `null`.
- Используя синтаксис фрагмента, можно возвращать произвольное количество элементов без любой обёртки.
- Компоненты могут возвращать строки или числа без любой обёртки.
- Компонент может отрендерить или не отрендерить значение, исходя от определенных условий.

Отлично. Мы рады, что ты уже здесь. Запасайся чаем с печеньками и гоу разузнаем о подходах стилизации React-приложений. 🧑🎨

Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту hello@lectrum.io.