

ЖИЗНЕННЫЙ ЦИКЛ КОМПОНЕНТОВ, НТТР



МИХАИЛ ЛАРЧЕНКО



МИХАИЛ ЛАРЧЕНКО

Frontend разработчик



horrorgr@gmail.com



ПЛАН ЗАНЯТИЯ

1. [Работа с HTTP](#)
2. [Жизненный цикл компонентов](#)
3. [Фазы жизненного цикла](#)
4. [Первоначальная отрисовка компонента](#)
5. [Изменение состояния компонента](#)
6. [Построение круговых диаграмм](#)
7. [Удаление компонента со страницы](#)



РАБОТА С HTTP



HTTP

Поскольку React - это библиотека для создания UI, она никак не ограничивает нас в выборе инструментов для работы с HTTP.

С клиентской точки зрения - это может быть что угодно:

1. XHR
2. Fetch
3. Библиотеки:
 - axios
 - superagent
 - и любые другие библиотеки



HTTP

Для работы с HTTP нам понадобятся:

1. Браузерное API (мы будем использовать Fetch API)
2. Web Service, с которого мы будем получать данные (воспользуемся Koa для его создания)

REST API

Создадим небольшое REST API с помощью Koa:

```
npm init  
npm install forever koa koa-router koa2-cors
```

REST API

```
const http = require('http');
const Koa = require('koa');
const Router = require('koa-router');
const cors = require('koa2-cors');
const app = new Koa();

app.use(cors());

const currencies = [{ "value": 1, "title": "Российский рубль", "code": "RUR" }, ... ];

const router = new Router();

router.get('/currencies', async (ctx, next) => {
  ctx.response.body = contacts;
});

app.use(router.routes()).use(router.allowedMethods());

const port = process.env.PORT || 7070;
const server = http.createServer(app.callback());
server.listen(port);
```

REST API

.foreverignore:

```
node_modules
```

scripts в package.json:

```
"scripts": {  
  "prestart": "npm install",  
  "start": "forever server.js",  
  "watch": "forever -w server.js"  
},
```



HTTP

Это всё здорово, но ключевой вопрос - где нам делать запросы? В конструкторе?

А что если мы будем использовать polling, SSE или WebSocket'ы? Тогда где подключаться?



ЖИЗНЕННЫЙ ЦИКЛ КОМПОНЕНТОВ

СОСТОЯНИЕ CLASS-BASED КОМПОНЕНТОВ

У компонентов может быть состояние (local state). Первоначальное состояние компонента можно определять через установку свойства `state`.

Также состояние можно изменять при помощи вызова метода `setState`. При изменении состояния React вычисляет отличия между старым состоянием и новым, после чего локально их применяет и перерисовывает компонент (`render`).



CLASS-BASED КОМПОНЕНТЫ


Почему мы говорим про Class-based компоненты, хотя современным подходом считается использование functional компонентов?

Потому что методы жизненного цикла изначально появились у них (и они много где ещё используются). Изучив их методы жизненного цикла, мы поймём, как это использовать с functional компонентах.

ПРОДВИНУТОЕ УПРАВЛЕНИЕ СОСТОЯНИЯМИ

Мы можем настраивать первоначальное состояние, изменять его и задавать отображение нашего компонента в методе `render`. Порой возникают ситуации, для которых данных возможностей будет маловато.

А что если мы используем загрузку данных с сервера? Как правильно ее обрабатывать в нашем компоненте?



ПРИЛОЖЕНИЕ «КОНВЕРТЕР ВАЛЮТ»

РЕАЛИЗУЕМ ПРИЛОЖЕНИЕ «КОНВЕРТЕР ВАЛЮТ»

Наше приложение представляет собой простейший конвертер валют RUB -> USD. Пользователь вводит в поле ввода сумму в рублях и получает значение в долларах. Пока в качестве текущего курса установим константное значение **57 рублей/доллар**.

Вроде бы ничего сложного, приступим к реализации.

КОМПОНЕНТ-КЛАСС Calculator

Реализуем компонент-класс `Calculator`. Мы выбрали данный вид компонента, потому что нам необходимо хранить значение (состояние) поля ввода. Для начала опишем `state` нашего компонента:

```
1  state = {  
2    rubAmount: 0,  
3    rate: 57  
4  };
```

В состоянии хранится текущая сумма в рублях (`rubAmount`), а также курс доллара к рублю (`rate`).

ДАЛЕЕ РЕАЛИЗУЕМ МЕТОД `render`

```
1  render() {
2    const { rubAmount, rate } = this.state;
3    return (
4      <div>
5        <h3>Конвертер валют:</h3>
6        <div>Текущий курс: {rate}</div>
7        <div>
8          <span>Сумма в рублях: </span>
9          <input
10             type="text"
11             placeholder="Сумма в рублях"
12             onChange={this.handleAmountChange}
13             value={rubAmount}/>
14        </div>
15        <span>Сумма в долларах: {this.calcUSDsum()}</span>
16      </div>
17    );
18  }
```

СОДЕРЖАНИЕ `render`

Метод `render` включает в себя контролируемое поле ввода для ввода суммы в рублях, а также блок ``, в котором содержится сумма в долларах.

ОБРАБОТЧИК `handleAmountChange`

Для изменения `rubAmount` был написан следующий обработчик `handleAmountChange`:

```
1  handleAmountChange = event => {  
2    this.setState({  
3      rubAmount: event.target.value  
4    });  
5  }
```

Данный обработчик срабатывает на событие поля ввода `change` и устанавливает новое значение `rubAmount`.

ФУНКЦИЯ РАСЧЕТА СУММЫ

Значение суммы в долларах рассчитывается функцией `calcUSDsum`:

```
1  calcUSDsum() {  
2    const { rubAmount, rate } = this.state;  
3    return (rubAmount / rate).toFixed(4)  
4  }
```



РЕЗУЛЬТАТ

Наш компонент работает, но пока он малополезен, поскольку мы не используем реальные данные о текущем курсе валют. Чтобы исправить это, настроим загрузку данных о курсе валют с внешнего сервиса.



ЗАГРУЗКА КУРСА ВАЛЮТ С ВНЕШНЕГО СЕРВИСА

В качестве источника информации о курсе валют будем использовать созданный нами веб-сервис. На самом деле, в интернете множество сервисов, которые предоставляют подобную информацию в форматах XML, JSON, JSONP, так что данный выбор не принципиален.

ПОЛУЧАЕМ ДАННЫЕ

Необходимо отправить GET-запрос на адрес <http://localhost:7070/currency>.

Сервер возвращает нам данные в следующем формате:

```
1  [{
2    "value":1,
3    "title":"Российский рубль",
4    "code":"RUR"
5      },
6      // ...
7      {
8    "value":5325.28,
9    "title":"Японских иен",
10   "code":"JPY"
11   }]
```




ОБРАБОТКА ЗАГРУЗКИ ДАННЫХ

Так, актуальные данные у нас есть, но мы опять возвращаемся к нашему вопросу:

Как правильно обрабатывать загрузку данных в нашем компоненте?

ДОБАВИМ СПЕЦИАЛЬНУЮ КНОПКУ

Например, можно инициировать загрузку данных по определенному событию, например, клику мыши. Добавим в наш компонент специальную кнопку, которая по клику будет загружать данные с сервиса и актуализировать курс валют:

```
1  <button onClick={this.loadActualRate}>
2    Загрузить курс валют
3  </button>
```

ENV

Достаточно плохой идеей является хардкодить URL сервиса прямо в код компонента.

Чтобы избежать этого, воспользуемся возможностью, предлагаемой `create-react-app`: мы можем создать файл `.env`, в котором прописать необходимые нам константы:

```
1 | REACT_APP_CURRENCY_URL=http://localhost:7070/currency
```

Префикс `REACT_APP_` обязателен.

.ENV

После чего мы сможем использовать:

```
1  loadActualRate = () => {  
2    fetch(process.env.REACT_APP_CURRENCY_URL)  
3    .then(response => response.json())  
4    .then(rates => {  
5      const findUSD = rate => rate.code === 'USD';  
6      const rate = rates.find(findUSD).value  
7      this.setState({ rate });  
8    });  
9  }
```

fetch

Для загрузки информации используем функцию `fetch` («улучшенный» XMLHttpRequest). Сама функция `fetch` возвращает нам Promise.

Promise хороши тем, что при помощи метода `then` мы можем составить цепочку из функций. Данные функции вызываются последовательно и модифицируют ответ сервера.

В первом `then` мы извлекаем из ответа JSON-данные, а во втором — устанавливаем новое значение курса рубля.

КАК АВТОМАТИЗИРОВАТЬ ЗАГРУЗКУ?

Загрузка работает, теперь осталось придумать, как это можно автоматизировать, чтобы не мучать пользователя нашего приложения.

Давайте поразмышляем: мы настраиваем состояние компонента в его конструкторе... Конструктор вызывается один раз при создании компонента...

Может, производить загрузку данных в конструкторе нашего компонента?

ДОПИШЕМ НАШ КОНСТРУКТОР.

Теперь метод `loadActualRate` вызывается в нем:

```
1  constructor(...params) {  
2      super(...params);  
3      this.loadActualRate();  
4  }
```



ЗАРАБОТАЛО!

Обновляем нашу страницу и видим, что курс валют загружается автоматически, не требуя от пользователей никаких усилий.

ТАК НЕ ДЕЛАЕТСЯ

На самом деле, использование конструктора класса для загрузки данных с сервера является антипаттерном. Почему?

Когда мы создаём абстракции с помощью классов и методов в любом языке программирования, мы придерживаемся ряда принципов. И один из них: функция или метод должны решать только одну задачу.

И задача конструктора React-компонента — инициализировать компонент. Поэтому в нем нет места для асинхронных вызовов.

Вот мы и подошли к теме нашей лекции «Жизненный цикл компонента».



ОСНОВНЫЕ ФАЗЫ ЖИЗНЕННОГО ЦИКЛА



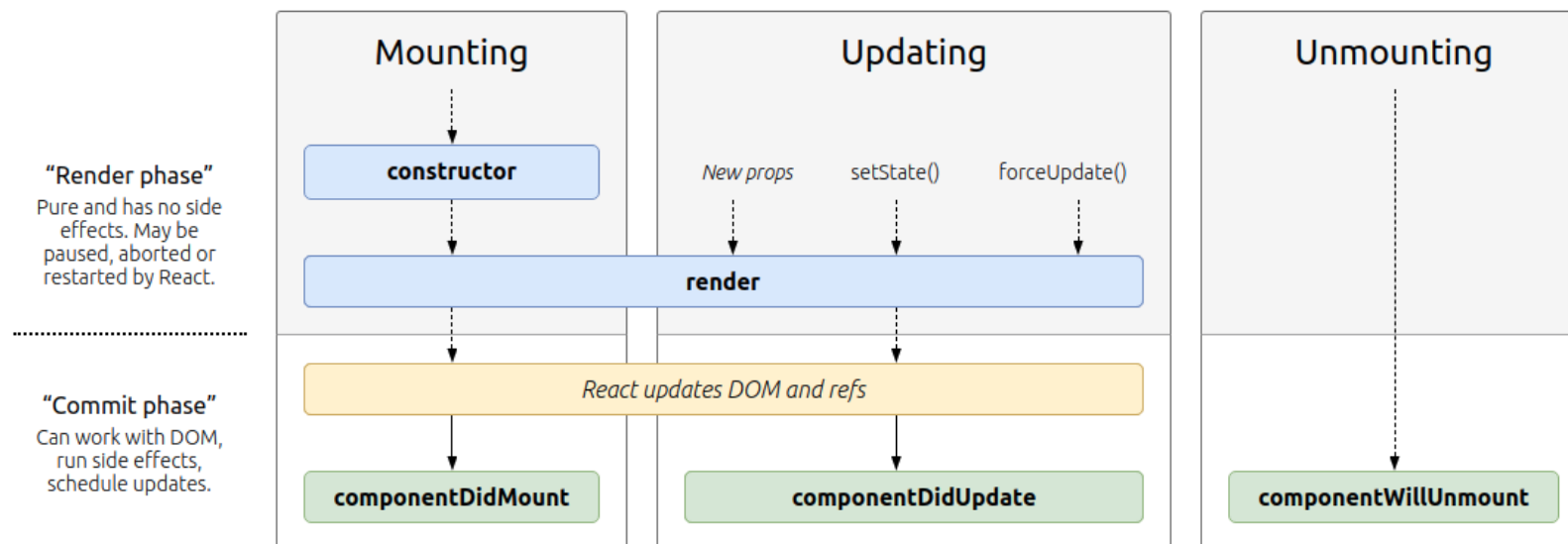
«ЖИЗНЕННЫЙ ЦИКЛ»

Что вообще понимается под термином «*жизненный цикл*»?

Если взять определение из биологии, то:

Жизненный цикл (lifecycle) — закономерная смена поколений (этапов в жизни), характерных для данного вида живых организмов.

ЖИЗНЕННЫЙ ЦИКЛ



Источник: <http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

ЖИЗНЕННЫЙ ЦИКЛ REACT-КОМПОНЕНТА

Условно React-компонент также можно сравнить с живым организмом, ведь у него тоже есть этапы жизненного цикла.

Если у организма это рождение, развитие (бодрствование) и смерть, то у React-компонента это:

1. Первоначальная отрисовка компонента (*Mounting*).
2. Обновление (*Updating*).
3. Удаление компонента со страницы (*Unmounting*).



МЕТОДЫ ЖИЗНЕННОГО ЦИКЛА

У всех этих этапов жизненного цикла фаз есть методы, так называемые *lifecycle-методы*.

Давайте остановимся на каждом из этапов подробнее и посмотрим, как можно использовать их методы.

ПЕРВОНАЧАЛЬНАЯ ОТРИСОВКА КОМПОНЕНТА (INITIAL RENDER, MOUNT PHASE)

СОЗДАДИМ КОМПОНЕНТ

Начнем с самого начала, с момента, когда у нас пока ничего нет. Мы реализовали наш компонент и решили добавить его на страницу:

```
1  class LifeComponent extends Component {  
2    render() {  
3      return <div>{this.props.name}</div>  
4    }  
5  }  
6  LifeComponent.defaultProps = {  
7    name: 'Компонент'  
8  };  
9  ReactDOM.render(  
10    <LifeComponent />,  
11    document.getElementById( 'root' )  
12  );
```


ПРОЦЕСС СОЗДАНИЯ ОБЪЕКТА

Перед началом создания нашего объекта React запрашивает у класса свойства по умолчанию (`defaultProps`). Если они определены, то он передает их в метод `constructor`, в противном случае передается пустой объект. Можем убедиться в этом сами.

Давайте реализуем в нашем компоненте метод `constructor` и выведем в консоль содержимое `props`:

```
1  constructor(props) {  
2    super(props);  
3    console.log(props);  
4  }
```

СМОТРИМ КОНСОЛЬ

```
▼ Object {name: "Компонент"} ⓘ  
  name: "Компонент"  
  ► __proto__: Object  
> |
```

А теперь уберем defaultProps :

```
▼ Object {} ⓘ  
  ► __proto__: Object  
> |
```

Полученные props могут быть использованы для инициализации первоначального состояния (*local state*) компонента.



render

После этого вызывается метод `render` – будет отрисован.

КОМПОНЕНТ ЕЩЕ НЕ ПОЯВИЛСЯ

На данной стадии компонент еще не появился на странице, следовательно, мы не можем взаимодействовать с его DOM. В этом легко убедиться:

```
1  class LifeComponent extends React.Component {  
2      constructor(props) {  
3          super(props);  
4          this.divRef = React.createRef();  
5      }  
6      render() {  
7          console.log(this.divRef);  
8          console.log(document.querySelector('.simple-div'));  
9          return (  
10             <div className="simple-div"  
11                 ref={this.divRef}>  
12                 {this.props.name}  
13             </div>  
14         );  
15     }
```

КАК ИЗМЕНИЛСЯ `LifeComponent`

Мы добавили в наш метод `render` сохранение ссылки на DOM-элемент `<div>`. Теперь мы можем взаимодействовать с элементом по ссылке `this.divElement`.

СМОТРИМ В КОНСОЛЬ

В результате получаем:

```
{current: null}  
null
```

Получается, что если нам требуется что-то сделать в реальном DOM, то ни `constructor`, ни `render` (первой отрисовке) нам не подходят.

Может быть, есть еще какие-нибудь методы жизненного цикла?

componentDidMount

Да, конечно! На стадии первоначального рендеринга есть также метод `componentDidMount`, который тоже вызывается один раз, но уже после того, как компонент был добавлен на страницу.


Добавим его в наш код:

```
1  componentDidMount() {  
2    console.log(this.divRef);  
3    console.log(document.querySelector('.simple-div'));  
4  }
```

ЭЛЕМЕНТ УЖЕ ДОБАВЛЕН НА СТРАНИЦУ

И видим, что элемент уже добавлен на страницу. И с ним можно работать:

```
1 | {current: div.simple-div}  
2 | <div class='simple-div'></div>
```

ИСПОЛЬЗОВАНИЕ МЕТОДА

componentDidMount

МЕТОД `componentDidMount`

Метод `componentDidMount` очень часто используется в компонентах. В основном — для загрузки данных с внешнего ресурса, для работы с DOM, инициализации сторонних библиотек, создания интервалов, таймеров, подписок (SSE, WS).

Вернемся к нашему приложению «Конвертер валют». Мы остановились на том, что решили загружать данные в конструкторе класса, однако это далеко не самое подходящее решение.

ИЗМЕНИМ «КОНВЕРТЕР ВАЛЮТ»

Теперь мы узнали про метод `componentDidMount`. Давайте перенесем загрузку данных в него:

```
1  componentDidMount() {  
2    this.loadActualRate();  
3  }
```

Обновляем страницу и видим, что все по-прежнему работает.

ПОРЯДОК ВЫЗОВА МЕТОДОВ (INITIAL RENDER, MOUNT PHASE)



ЛОГИРОВАНИЕ LIFECYCLE-МЕТОДОВ

Напишем компонент, в котором вызов каждого из методов будет логироваться в консоль браузера.

СОЗДАЕМ КЛАСС

```
1  class LifeComponent extends React.Component {
2    constructor(props) {
3      super(props);
4      console.log('1. Компонент был настроен');
5    }
6    render() {
7      console.log('2. Компонент монтируется');
8      return <div>{this.props.name}</div>;
9    }
10   componentDidMount() {
11     console.log('3. Компонент был примонтирован');
12   }
13 }
```

ЗАДАДИМ defaultProps И ВЫВЕДЕМ

```
1  LifeComponent.defaultProps = {  
2    name: 'Компонент'  
3  };  
4  
5  ReactDOM.render(  
6    <LifeComponent />,  
7    document.getElementById('root')  
8  );
```



ВЫВОД В КОНСОЛЬ

И вот, что мы получаем в консоли:

1. Компонент был настроен
2. Компонент монтируется
3. Компонент был примонтирован

ИЗМЕНЕНИЕ СОСТОЯНИЯ КОМПОНЕНТА (PROPS CHANGE И STATE CHANGE, UPDATING PHASE)

ПРИЧИНЫ ИЗМЕНЕНИЯ СОСТОЯНИЯ КОМПОНЕНТА

Состояние компонента может изменяться по нескольким причинам:

- при изменении внутреннего состояния компонента;
- и при изменении его `props`, которые были получены от родителя.

В обоих случаях компонент будет перерисован.

До этого мы уже сталкивались с подобным поведением, но при этом не вдавались в подробности жизненного цикла компонента.

LIFECYCLE-МЕТОДЫ НА СТАДИИ ИЗМЕНЕНИЯ СОСТОЯНИЯ

Как вы уже могли догадаться, на стадии изменения состояния компонента также есть свои lifecycle-методы:

1. `render`
2. `componentDidUpdate`

Рассмотрим данные методы на конкретных примерах.



ПОСТРОЕНИЕ КРУГОВЫХ ДИАГРАММ

CHART.JS

При работе часто возникают задачи по визуализации набора каких-либо данных. Допустим, нас попросили построить круговую диаграмму популярности языков программирования.

Для построения самой диаграммы возьмем стороннюю библиотеку Chart.js. Подключить ее можно из CDN по ссылке — <https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.js> (или через пакет npm — `chart.js`).

Возьмем пример диаграммы с сайта библиотеки и имплементируем его для нашего случая.

СОЗДАДИМ КЛАСС `CircleChart`

```
1  class CircleChart extends React.Component {  
2    constructor(...params) {  
3      super(...params);  
4      this.data = {  
5        datasets: [{  
6          data: [40, 90, 30],  
7          backgroundColor: ['yellow', 'red', 'green']  
8        }],  
9        labels: ['JavaScript', 'Java', 'C#']  
10     };  
11   }  
12 }
```

ДОПОЛНИМ `CircleChart` МЕТОДАМИ

```
1  componentDidMount() {  
2    this.chart = new Chart("myChart", {  
3      type: 'doughnut',  
4      data: this.data  
5    });  
6  }
```

```
1  render() {  
2    return (  
3      <div>  
4        <h2>Популярность языков программирования</h2>  
5        <canvas id="myChart" width="100" height="100" />  
6      </div>  
7    );  
8  }
```

ЛОГИКА НАШЕГО КОМПОНЕНТА

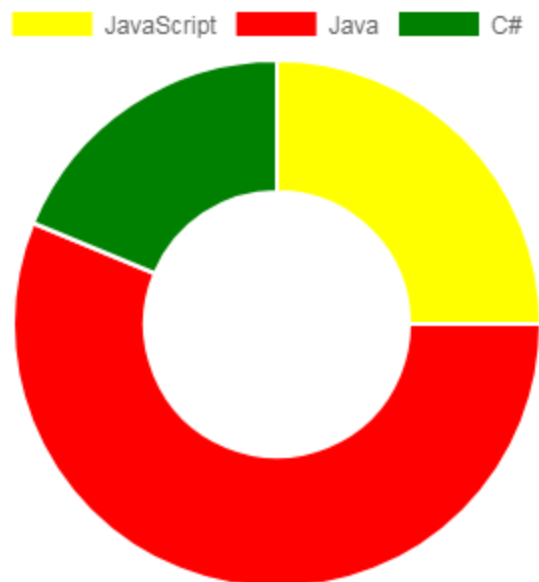
В конструкторе компонента задаем начальные параметры диаграммы.

В методе `componentDidMount` производим инициализацию библиотеки. Как мы уже и говорили ранее, данный метод предназначен для работы со сторонними библиотеками и взаимодействиями с DOM.

РЕЗУЛЬТАТ

Если все сделали правильно, то получим следующий результат:

**Популярность языков
программирования**



ДОБАВИМ ПОЛЕ ВВОДА

Хорошо, а теперь добавим поле ввода для ввода своих значений:

```
1  render() {  
2    return (  
3      <div>  
4        <h2>Популярность языков программирования</h2>  
5        <div>JavaScript:  
6          <input type="text" name="jsRate"  
7            value={this.state.jsRate}  
8            onChange={this.handleChange} />  
9        </div>  
10       // ... и тоже самое для других языков  
11       <canvas id="myChart" width="100" height="100" />  
12     </div>  
13   );  
14 }
```

ДОБАВИМ ОПРЕДЕЛЕНИЕ СОСТОЯНИЕ

```
1 state = {  
2   jsRate: 40,  
3   javaRate: 90,  
4   sharpRate: 30  
5 };
```

НАПИШЕМ ЛОГИКУ ОБРАБОТЧИКА

```
1  handleChange = event => {  
2    this.setState({  
3      [event.target.name]: event.target.value  
4    });  
5  }
```



ДИАГРАММА НЕ МЕНЯЕТСЯ

Попробуем ввести в них значение — видим, что диаграмма остается неизменной.

На самом деле, ничего удивительного в этом нет, ведь мы задали начальные значения нашей диаграммы в конструкторе, а после никак их не изменяем.

МЕСТО ИНИЦИАЛИЗАЦИИ ДИАГРАММЫ

Напомню, что мы инициировали круговую диаграмму в методе `componentDidMount`:

```
1  componentDidMount() {  
2    this.chart = new Chart("myChart", {  
3      type: 'doughnut',  
4      data: this.data  
5    });  
6  }
```

Этот метод вызывается один раз при создании компонента.

ОБРАТИМСЯ К ДОКУМЕНТАЦИИ

Этот метод вызывается один раз при создании компонента. А нам нужно перерисовать диаграмму, и в документации к Chart.js есть пример обновления диаграммы:

```
1 chart.data.labels.push(label);
2 chart.data.datasets.forEach((dataset) => {
3     dataset.data.push(data);
4 });
5 chart.update();
```

Где мы можем это сделать?

ИНИЦИАЛИЗАЦИЯ В МЕТОДЕ `render` ?

Во-первых, у нас возникнут проблемы с первичной отрисовкой, потому что объект `Chart` еще не создан, он будет создан в `componentDidMount`.

Во-вторых, метод `render` для создания внешнего вида компонента, а не для вызова различных API.

МОМЕНТ ИЗМЕНЕНИЯ НАШЕГО КОМПОНЕНТА

Следовательно, нам необходимо отловить момент изменения нашего компонента, установить новые значения нашей диаграммы и перерисовать ее.

Для этого есть метод: `componentDidUpdate`.

После того, как наш компонент был успешно обновлен, вызывается метод `componentDidUpdate`. Зачастую применяется для внесения необходимых изменений в DOM средствами не библиотеки React.

НАСТРОИМ ОБНОВЛЕНИЕ ДАННЫХ

В данном примере мы вызываем метод `updateChart`, который предназначен для обновления нашей диаграммы:

```
1  componentDidUpdate() {  
2    this.updateChart();  
3  }  
4  
5  updateChart() {  
6    this.chart.data.datasets[0].data = [  
7      this.state.jsRate,  
8      this.state.javaRate,  
9      this.state.sharpRate  
10   ];  
11   this.chart.update();  
12 }
```

РЕАЛИЗУЕМ КОМПОНЕНТНЫЙ ПОДХОД

Зачастую возникает ситуация, когда наши компоненты используют свойства, которые были переданы им от своих родителей.

Давайте проиллюстрируем данный случай, используя все тот же пример с диаграммой. Разделим наш компонент на два:

- `App` (управляющие элементы);
- `CircleChart` (круговая диаграмма).

А также внесем пару изменений.

СОЗДАДИМ КЛАСС App

```
1 class App extends React.Component {  
2   state = {  
3     jsRate: 40,  
4     javaRate: 90,  
5     sharpRate: 30  
6   };  
7 }
```

ДОБАВИМ App МЕТОД render

```
1  render() {
2    return (
3      <div>
4        <h2>Популярность языков программирования</h2>
5        <div>JavaScript:
6          <input type="text" name="jsRate" value={this.state.jsRate}
7            onChange={this.handleChange} />
8        </div>
9        // ... и тоже самое для других языков
10       <CircleChart rates={[
11         this.state.jsRate,
12         this.state.javaRate,
13         this.state.sharpRate ]} />
14     </div>
15   );
16 }
```

ДОБАВИМ В `App` ОБРАБОТЧИК

```
1  handleChange = event => {  
2    this.setState({  
3      [event.target.name]: event.target.value  
4    });  
5  }
```

Наш компонент `App` готов!

Приступим к самой диаграмме: `CircleChart`

СОЗДАДИМ КЛАСС `CircleChart`

```
1  class CircleChart extends React.Component {  
2  
3      constructor(...params) {  
4          super(...params);  
5  
6          this.data = {  
7              datasets: [{  
8                  data: params[0].rates,  
9                  backgroundColor: ['yellow', 'red', 'green']  
10             }],  
11             labels: [ 'JavaScript', 'Java', 'C#' ]  
12         };  
13     }  
14 }
```

ДОБАВИМ В CircleChart LIFECYCLE-МЕТОДЫ И updateChart

```
1  componentDidMount() {  
2    this.chart = new Chart("myChart", {  
3      type: 'doughnut',  
4      data: this.data  
5    });  
6  }  
7  componentDidUpdate(prevProps, prevState) {  
8    if (this.props.rates !== prevProps.rates) {  
9      this.updateChart(this.props.rates);  
10   }  
11 }  
12 updateChart(rates) {  
13   this.chart.data.datasets[0].data = rates;  
14 }
```


ОСТАЛОСЬ РЕАЛИЗОВАТЬ ПРЕДСТАВЛЕНИЕ

Добавим в `CircleChart` мето `render`:

```
1  render() {  
2    return <canvas id="myChart" width="100" height="100"/>;  
3  }
```

Выведем наш компонент в DOM:

```
1  ReactDOM.render(  
2    <App />,  
3    document.getElementById( 'root' )  
4  );
```

componentDidUpdate

Теперь, поскольку данные в `CircleChart` поступают от родителя `App` через `props`, мы можем использовать специальный метод `componentDidUpdate`.


В качестве аргументов данный метод получает предыдущие `props` и `state` свойства компонента (поэтому мы должны проверять в `if`, что `props` изменились, чтобы не уйти в рекурсию).

Стоит отметить, что данный метод не вызывается при первоначальном рендеринге компонента.

ПЕРЕРИСОВКА ДИАГРАММЫ

Перерисовать нашу диаграмму можно следующим образом:

```
1  componentDidUpdate(prevProps, prevState) {  
2    if (this.props.rates !== prevProps.rates) {  
3      this.updateChart(this.props.rates);  
4    }  
5  }  
6  
7  updateChart(rates) {  
8    this.chart.data.datasets[0].data = rates;  
9    this.chart.update();  
10 }
```



ДЕТЕКТИРОВАНИЕ ИЗМЕНЕНИЯ РАЗМЕРОВ ОКНА

СЛЕДИМ ЗА ИЗМЕНЕНИЯМИ РАЗМЕРОВ СТРАНИЦЫ

Перед нами стоит задача реализовать простой компонент, который будет следить за изменениями размеров страницы.

В случае, если размер страницы был изменен, то в консоль должно быть написано сообщение `Размер страницы был изменен!`.

Назовем наш компонент `WindowResizeDetector`. После добавления этого компонента на страницу он должен следить за изменением размеров страницы.

КОД НАШЕГО КОМПОНЕНТА

```
1  class WindowResizeDetector extends React.Component {
2      componentDidMount() {
3          window.addEventListener(
4              'resize',
5              this.resizeHandler
6          );
7      }
8      resizeHandler = () => {
9          console.log('Размер страницы был изменен!');
10     }
11     render() {
12         return <div>Детектор включен</div>;
13     }
14 }
```

ЛОГИКА РАБОТЫ

WindowResizeDetector

В методе `componentDidMount` компонента `WindowResizeDetector` мы назначаем обработчик на каждое изменение размеров экрана:

```
window.addEventListener(  
  'resize',  
  this.resizeHandler  
);
```

РАСШИРЯЕМ ФУНКЦИОНАЛ

После этого нас попросили расширить функционал и добавить возможность включения и отключения детектирования изменений.

Вроде ничего сложного. Напишем наш корневой компонент `App`, в котором мы сможем включать и выключать режим детектирования изменений.

РЕАЛИЗАЦИЯ App

```
1  class App extends React.Component {  
2      state = {  
3          detectorIsEnabled: false  
4      };  
5  
6      toggleDetectorState() {  
7          this.setState(({ detectorIsEnabled }) => ({  
8              detectorIsEnabled: !detectorIsEnabled  
9          }));  
10     }  
11 }
```

ДОБАВИМ App МЕТОД render

```
1  render() {  
2    return (<div>  
3      <button  
4        onClick={this.toggleDetectorState}>  
5  
6        {this.state.detectorIsEnabled ?  
7          "Выключить детектор" : "Включить детектор"}  
8      </button>  
9  
10     {this.state.detectorIsEnabled ?  
11       <WindowResizeDetector /> : null}  
12   </div>);  
13 }
```

ПРОВЕРЯЕМ РАБОТУ ДЕТЕКТОРА

Включим наш детектор и попробуем изменить размер экрана — видим, что в консоли стали появляться сообщения.

Теперь выключим его и повторим еще раз. Сообщения в консоли не прекратили появляться.

Странно, но ведь компонента нет на странице?

componentWillUnmount

Компонента нет, но побочные эффекты в виде его обработчика остались, потому что мы забыли их удалить.

Следовательно, нам нужно убрать все побочные эффекты после удаления нашего компонента со страницы.

Для этого в React есть специальный метод жизненного цикла — `componentWillUnmount`.



УДАЛЕНИЕ КОМПОНЕНТА СО СТРАНИЦЫ

МЕТОД `componentWillUnmount`

Метод `componentWillUnmount` относится к последней стадии жизни React-компонента — *Unmount Phase*. На данной стадии обычно удаляются побочные эффекты от использования сторонних библиотек, удаляются ненужные обработчики событий.

Чтобы в нашем примере все заработало как надо, необходимо дописать метод `componentWillUnmount` и удалить в нем обработчик события `resize`.

ДОБАВИМ МЕТОД `componentWillUnmount`

```
1  componentWillMount() {  
2    window.removeEventListener(  
3      'resize',  
4      this.resizeHandler  
5    );  
6  }
```

Проверим все еще раз. Видим, что при удалении компонента сообщения перестали появляться.



ПОДВЕДЕМ ИТОГИ



ЖИЗНЕННЫЙ ЦИКЛ КОМПОНЕНТОВ REACT

Мы узнали, что у компонентов React есть свой жизненный цикл. Можно выделить три основных этапа в «жизни» каждого компонента: начальный рендеринг, обновление компонента, удаление компонента со страницы.

Каждый из этапов жизненного цикла содержит специализированные методы, которые позволяют нам контролировать процесс создания, обновления и удаления компонента со страницы.

НАЧАЛЬНЫЙ РЕНДЕРИНГ (*MOUNTING PHASE*)

1. `constructor` — инициализация компонента;
2. `render`;
3. `componentDidMount` — работа с AJAX, DOM и сторонними библиотеками, можно использовать `this.setState`.

ОБНОВЛЕНИЕ КОМПОНЕНТА (*UPDATING PHASE*)

1. `render`;
2. `componentDidUpdate`, можно использовать `this.setState`.

УДАЛЕНИЕ КОМПОНЕНТА СО СТРАНИЦЫ (*UNMOUNT PHASE*)

1. `componentWillUnmount` — удаление ненужных обработчиков.

USEEFFECT

Functional компоненты предлагают собственное решение озвученных проблем: а именно хук `useEffect`, который заменяет собой связку из `componentDidMount`, `componentDidUpdate` и `componentWillUnmount`.

С этим хуком мы детально ознакомимся на одной из следующих лекций.



Задавайте вопросы и напишите отзыв о лекции!

МИХАИЛ ЛАРЧЕНКО

 horrorgr@gmail.com