

# Жизненный цикл компонента

## Введение

Ранее мы познакомились с такими интересными темами, как состояние компонента и компоненты на основе классов. Из них мы узнали, что кроме простых (*dumb*) компонентов в React есть также и умные (*class components*). Основное отличие заключается в том, что у умных компонентов есть состояние (*local state*). Первоначальное состояние компонента можно определять через `this.state = something` в конструкторе нашего компонента (`constructor`). 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`).

Таким образом, 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?

Давайте попробуем разобраться на конкретном примере. Попробуем реализовать приложение «Конвертер валют».

## 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.

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

```
constructor(...params) {  
  super(...params);  
  this.state = {  
    rubAmount: 0,  
    rate: 57  
  };  
}
```

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

```
render() {
  const { rubAmount, rate } = this.state;

  return (
    <div>
      <h3>Конвертер валют:</h3>
      <div>Текущий курс: {rate}</div>
      <div>
        <span>Сумма в рублях: </span>
        <input
          type="text"
          placeholder="Сумма в рублях"
          onChange={this.handleAmountChange.bind(this)}
          value={rubAmount}/>
      </div>
      <span>Сумма в долларах: {this.calcUSDsum()}</span>
    </div>
  );
}
```

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

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

```
handleAmountChange(event) {
  this.setState({
    rubAmount: event.target.value
  });
}
```

Данный обработчик срабатывает на событие поля ввода `change` и устанавливает новое значение `rubAmount`. Значение суммы в долларах рассчитывается функцией `calcUSDsum`:

```
calcUSDsum() {
  return (this.state.rubAmount / this.state.rate).toFixed(4)
}
```

Если все было сделано правильно, то получим следующий результат: 

<https://codepen.io/hoodsey20/pen/awxWOX>

Наш компонент работает, но пока он мало полезен, 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 ресурс - <https://neto-api.herokuapp.com/currency>. На самом деле в интернете множество сервисов, которые предоставляют подобную информацию в форматах XML, JSON, JSONP, так что данный выбор не принципиален.

Для получения текущего курса рубля к доллару необходимо отправить GET-запрос на адрес <https://neto-api.herokuapp.com/currency>.

Сервер возвращает нам данные в следующем формате:

```
[{
  "value":1,
  "title":"Российский рубль",
  "code":"RUR"
}, {
  "value":60.1836,
  "title":"Доллар США",
  "code":"USD"
}, {
  "value":68.8139,
  "title":"Евро",
  "code":"EUR"
}, {
  "value":9335.859999999999,
  "title":"Индийских рупий",
  "code":"INR"
}, {
  "value":52950.8,
  "title":"Вон Республики Корея",
  "code":"KRW"
}, {
  "value":5325.28,
  "title":"Японских иен",
  "code":"JPY"
}]
```



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

Как правильно обрабатывать загрузку данных в нашем компоненте?

Можно инициировать загрузку данных по определенному событию, например клику мыши. Это далеко не самый лучший вариант, поскольку пользователю необходимо совершать лишние действия, но пока мы не можем придумать способа лучше. Так что давайте реализуем именно так. Добавим в наш компонент специальную кнопку, которая по клику будет загружать данные с сервиса и актуализировать курс валют.


Наша кнопка:

```
<button onClick={this.loadActualRate.bind(this)}>
  Загрузить курс валют
</button>
```

Метод загрузки курса валют:

```
loadActualRate() {
  fetch('https://neto-api.herokuapp.com/currency')
    .then(response => response.json())
    .then(rates => {
      console.log(rates);
      this.setState({
        rate: rates.find(rate => rate.code === "USD").value
      });
    });
}
```

Для загрузки информации используем функцию `fetch` («улучшенный» `XMLHttpRequest`). Сама функция `fetch` возвращает нам обещание (`Promise`). Обещания 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`, а во втором — устанавливаем новое значение курса рубля.

Давайте кликнем на кнопку и посмотрим, что у нас появилось в консоли. Если в консоли появилась цифра, и значение в «Текущий курс» также изменилось, то значит все хорошо:   
<https://codepen.io/hoodsey20/pen/jwRmLz>

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?

Что же, давайте попробуем.

Допишем наш конструктор. Теперь метод `loadActualRate` вызывается в нем:

```
constructor(...params) {
  super(...params);
  this.state = {
    rubAmount: 0,
    rate: 57
  };
  this.loadActualRate();
}
```

```
}
```

Обновляем нашу страницу и видим, что курс валют загружается автоматически, не требуя от пользователей никаких усилий. <https://codepen.io/hoodsey20/pen/MoRmPL> На самом деле, использование конструктора класса для загрузки данных с сервера является антипаттерном в любом языке программирования. Конструктор используется для инициализации внутреннего состояния создаваемого объекта и в нем не место для асинхронных вызовов.

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

## Основные фазы жизненного цикла

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.

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 - первоначальная отрисовка компонента (*Initial Render*), изменение внутреннего состояния (*Props change* и *State change*), а также удаление компонента со страницы (*Component Unmount*).

У всех этих этапов жизненного цикла фаз есть методы, так называемые *lifecycle-methods*. Давайте остановимся на каждом из этапов подробнее и посмотрим как можно использовать их методы.

## Первоначальная отрисовка компонента (*Initial Render, Mount phase*)

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

```
class LifeComponent extends React.Component {
  render() {
    return <div>{this.props.name}</div>
  }
}


LifeComponent.defaultProps = {
  name: 'Компонент'
};

ReactDOM.render(
  <LifeComponent />,
  document.getElementById('root')
);
```

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`.

```
constructor(props) {  
  super(props);  
  console.log(props);  
}
```

Смотрим что в консоли: 

А теперь уберем `defaultProps`:  <https://codepen.io/hoodsey20/pen/mwgwdB> Полученные `props` могут быть использованы для инициализации первоначального состояния (*local state*) компонента.

После этого наступает стадия `componentWillMount` - компонент будет примонтирован. Данный метод вызывается один раз, непосредственно перед вызовом метода `render`.

Чем он может быть нам полезен?

Одно из возможных применений данного метода - настройка локального состояния компонента через метод `this.setState`, как замена инициализации первоначального состояния в конструкторе нашего компонента.

Если вы вызовете `this.setState` внутри этого метода, то `render()` получит обновленное состояние и будет выполнен только один раз, несмотря на изменение состояния.

Можно ли работать с компонентом через DOM на данной стадии?

На данной стадии компонент еще не появился на странице, следовательно мы не можем взаимодействовать с его DOM.

В этом легко убедиться:

```
class LifeComponent extends React.Component {  
  componentWillMount() {  
    console.log(this.divElement);  
    console.log(document.querySelector('.simple-div'));  
  }  
  
  render() {  
    return (  

```

```

    <div className="simple-div" ref={div => this.divElement = div}>
      {this.props.name}
    </div>
  );
}
}

ReactDOM.render(
  <LifeComponent />,
  document.getElementById('root')
);

```

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

Также добавили метод `componentWillMount` в котором попробуем вывести наш DOM-element при помощи `ref` ссылки и обычного `document.querySelector`.

В результате получаем: 

Видим, что наш элемент еще не был добавлен на страницу.

Может быть есть еще какие-нибудь методы жизненного цикла?


Да, конечно! Помимо `componentWillMount`, 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:

```

componentDidMount() {
  console.log(this.divElement);
  console.log(document.querySelector('.simple-div'));
}

```

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

## Использование метода `componentDidMount`

Метод `componentDidMount` очень часто используется в компонентах. В основном - для загрузки данных с внешнего ресурса, для работы с DOM или инициализации сторонних библиотек.

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

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

```
componentDidMount() {  
  this.loadActualRate();  
}
```

Обновляем страницу и видим что все по-прежнему работает. Возможно у вас возникнет вопрос:

А можно загружать данные в методе `componentWillMount`?

Можно, но есть риск того, что мы получим **непредвиденную ошибку**. Поскольку все вызовы `this.setState` в методе `componentWillMount` не приводят к вызову метода `render`. Таким образом, если загрузка данных с внешнего сервиса закончится после того как компонент будет добавлен на страницу, то компонент не будет перерисован.

## Порядок вызова методов (*Initial Render*, Mount phase)

Для закрепления материала давайте напишем компонент, в котором вызов каждого из методов будет логироваться в консоль браузера.

```
class LifeComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    console.log('1. Компонент был настроен');  
  }  
  
  componentWillMount() {  
    console.log('2. Компонент будет примонтирован');  
  }  
  
  componentDidMount() {  
    console.log('4. Компонент был примонтирован');  
  }  
  
  render() {  
    console.log('3. Компонент монтируется');  
    return (  
      <div>  
        {this.props.name}  
      </div>  
    );  
  }  
}  
  
LifeComponent.defaultProps = {  
  name: 'Компонент'  
};  
  
ReactDOM.render(  
  <LifeComponent />,
```



```
document.getElementById('root')
);
```

И вот, что мы получаем в консоли: 

## Изменение состояния компонента (*Props change* и *State change*, Updating phase)

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.

Как вы уже могли догадаться, на стадии изменения состояния компонента также есть свои `lifecycle-методы`: 1. `componentWillReceiveProps` 2. `shouldComponentUpdate` 3.

`componentWillUpdate` 4. `componentDidUpdate`

Рассмотрим данные методы на конкретных примерах.

## Построение круговых диаграмм

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/2.6.0/Chart.min.js> (или через пакет `npm` - `chart.js`).

Не будем сильно заострять на этом внимание, а просто возьмем пример диаграммы с сайта библиотеки и имплементируем его для нашего случая.

```
class CircleChart extends React.Component {

  constructor(...params) {
    super(...params);
    this.data = {
      datasets: [{
        data: [40, 90, 30],
        backgroundColor: ['yellow', 'red', 'green']
      }],
      labels: [
        'JavaScript',
        'Java',
        'C#'
      ]
    }
  }
}
```

```

    };
  }


  componentDidMount() {
    this.chart = new Chart("myChart", {
      type: 'doughnut',
      data: this.data
    });
  }

  render() {
    return (
      <div>
        <h2>Популярность языков программирования</h2>
        <canvas id="myChart" width="100" height="100" />
      </div>
    );
  }
}

ReactDOM.render(
  <CircleChart />,
  document.getElementById('root')
);

```

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.

Если все сделали правильно, то получим следующий результат: 

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

```

handleChange(event) {
  this.setState({
    [event.target.name]: event.target.value
  });
}

render() {
  return (
    <div>
      <h2>Популярность языков программирования</h2>
      <div>
        JavaScript:
        <input type="text" name="jsRate" value={this.state.jsRate}
onChange={this.handleChange.bind(this)} />
      </div>
      <div>
        Java:
        <input type="text" name="javaRate" value={this.state.javaRate}
onChange={this.handleChange.bind(this)} />
      </div>
    </div>
  );
}

```

```

    <div>
      C#:
      <input type="text" name="sharpRate" value={this.state.sharpRate}
onChange={this.handleChange.bind(this)} />
    </div>
    <canvas id="myChart" width="100" height="100" />
  </div>
);
}

```

Добавим в конструктор значения полей по-умолчанию:

```

this.state = {
  jsRate: 40,
  javaRate: 90,
  sharpRate: 30
}

```

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.

<https://codepen.io/hoodsey20/pen/awxYYB> Напомню, что мы инициировали круговую диаграмму в методе `componentDidMount`:

```

componentDidMount() {
  this.chart = new Chart("myChart", {
    type: 'doughnut',
    data: this.data
  });
}

```

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

```

chart.data.labels.push(label);
chart.data.datasets.forEach((dataset) => {
  dataset.data.push(data);
});
chart.update();

```

Где мы можем это сделать? 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.

Следовательно нам необходимо отловить момент изменения нашего компонента, установить новые значения нашей диаграммы и перерисовать ее. Для этого есть два метода:

`componentWillUpdate` и `componentDidUpdate`: - метод `componentWillUpdate` - компонент будет обновлен, - метод `componentDidUpdate` - компонент был обновлен.

Компонент `componentWillUpdate` вызывается непосредственно перед обновлением компонента (при изменении `props` или `state`) и в качестве аргументов он получает новые свойства (`nextProps`) и новое состояние компонента (`nextState`).

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

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

```
componentDidUpdate() {
  this.updateChart();
}

updateChart() {
  this.chart.data.datasets[0].data = [
    this.state.jsRate,
    this.state.javaRate,
    this.state.sharpRate
  ];
  this.chart.update();
}
```

Теперь все работает как надо. <https://codepen.io/hoodsey20/pen/PjgaOp>

## Разделим управляющие элементы и диаграмму на два компонента

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) и `CircleChart` (круговая диаграмма). А также внесем пару изменений:

```
class App extends React.Component {

  constructor(...params) {
    super(...params);
    this.state = {
      jsRate: 40,
      javaRate: 90,
      sharpRate: 30
    }
  }
```

```

    }

    handleChange(event) {
      this.setState({
        [event.target.name]: event.target.value
      })
    }

    render() {
      return (
        <div>
          <h2>Популярность языков программирования</h2>
          <div>
            JavaScript:
            <input type="text" name="jsRate" value={this.state.jsRate}
onChange={this.handleChange.bind(this)} />
          </div>
          <div>
            Java:
            <input type="text" name="javaRate" value={this.state.javaRate}
onChange={this.handleChange.bind(this)} />
          </div>
          <div>
            C#:
            <input type="text" name="sharpRate" value={this.state.sharpRate}
onChange={this.handleChange.bind(this)} />
          </div>
          <CircleChart rates={[this.state.jsRate, this.state.javaRate,
this.state.sharpRate]} />
        </div>
      )
    }
  }

  class CircleChart extends React.Component {

    constructor(...params) {
      super(...params);

      this.data = {
        datasets: [{
          data: params[0].rates,
          backgroundColor: ['yellow', 'red', 'green']
        }],
        labels: [
          'JavaScript',
          'Java',
          'C#'
        ]
      };
    }

    componentDidMount() {
      this.chart = new Chart("myChart", {
        type: 'doughnut',
        data: this.data
      });
    }
  }

```

```

    }

    componentWillReceiveProps(newProps) {
      this.updateChart(newProps.rates);
    }

    updateChart(rates) {
      this.chart.data.datasets[0].data = rates;
      this.chart.update();
    }

    render() {
      return <canvas id="myChart" width="100" height="100" />
    }
  }

ReactDOM.render(<App />, document.getElementById('root'));

```

<https://codepen.io/hoodsey20/pen/ZyZRrm> Теперь, поскольку данные в CircleChart поступают от родителя App через props, мы можем использовать специальный метод `componentWillReceiveProps`. В качестве аргументов данный метод получает новые свойства компонента (`nextProps`). Стоит отметить, что данный метод не вызывается при первоначальном рендеринге компонента.

Перерисовать нашу диаграмму можно следующим образом:

```

componentWillReceiveProps(newProps) {
  this.updateChart(newProps.rates);
}

updateChart(rates) {
  this.chart.data.datasets[0].data = rates;
  this.chart.update();
}

```

## Метод `shouldComponentUpdate` - следует обновить компонент

Как мы уже знаем, по философии React, если компонент получает новые свойства (`props`) или его локальное состояние изменяется, то вызывается `render`. Как ни странно, такое поведение можно отменить и вручную задавать случаи, когда нам следует обновлять компонент.

Вот небольшой пример (все та же диаграмма), при изменении состояния компонента вызывается тяжелая функция для расчета новых значений, из-за этого интерфейс подтормаживает и пользователю становится неудобно в нем работать:

```

componentWillUpdate(newProps) {
  if (Date.now() - this.lastUpdate > 500) {

    for (var i = 0; i < 5000000000; i++) {
      // очень сложные вычисления
    }
  }
}

```

```

    }
    this.updateChart(newProps.rates);
  }
}

updateChart(rates) {
  this.lastUpdate = Date.now();
  this.chart.data.datasets[0].data = rates;
  this.chart.update();
}

```

Чтобы этого избежать необходимо использовать метод `shouldComponentUpdate`.

Метод `shouldComponentUpdate` вызывается перед обновлением компонента. В качестве аргументов он получает новые свойства (`nextProps`) и новое состояние компонента (`nextState`).

Данный метод отвечает на вопрос: «Следует ли обновлять компонент?» и всегда должен возвращать `true` или `false`. По-умолчанию данный метод возвращает `true`. В данном методе также запрещается использовать `this.setState`, иначе наша программа уйдет в бесконечный цикл. Метод `this.setState` будет приводить к вызову `shouldComponentUpdate`, который в свою очередь будет опять вызывать `this.setState`.

```

shouldComponentUpdate(nextProps, nextState) {
  return this.props.rates.some((rate, index) => rate !== nextProps.rates[index]);
}

```

В примере выше мы говорим React, что следует обновлять компонент только тогда, когда хотя бы один из элементов массива `rates` изменился, следовательно диаграмму необходимо перерисовать.

## Компонент для детектирования изменения размеров страницы

Давайте рассмотрим еще один пример. Перед нами стоит задача реализовать простой компонент, который будет следить за изменениями размеров страницы. В случае, если размер экрана был изменен, то в консоль должно быть написано сообщение `Размер окна был изменен!`.

Назовем наш компонент - `WindowResizeDetector`. После добавлении этого компонента на страницу он должен следить за изменением размеров экрана.

Рассмотрим код нашего компонента.

```

class WindowResizeDetector extends React.Component {

  componentDidMount() {

```

```

    window.addEventListener('resize', this.windowResizeHandler);
  }

  windowResizeHandler() {
    console.log('Размер окна был изменен!');
    // ...логика обработки события
  }

  render() {
    return <div>Детектор включен</div>;
  }
}

```

<https://codepen.io/hoodsey20/pen/weZxMX> В компоненте мы видим метод `componentDidMount`, который вызывается после того как компонент был примонтирован к странице. В данном методе мы назначаем обработчик на каждое изменение экрана:

```

window.addEventListener('resize', this.windowResizeHandler);

```

Здорово, все работает. 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.

```

class App extends React.Component {

  constructor(...params) {
    super(...params)
    this.state = {
      detectorIsEnabled: false
    };
  }

  toggleDetectorState() {
    this.setState({
      detectorIsEnabled: !this.state.detectorIsEnabled
    });
  }

  render() {
    return (
      <div>
        <button onClick={this.toggleDetectorState.bind(this)}>
          {this.state.detectorIsEnabled ? "Выключить детектор" : "Включить детектор"}
        </button>
        {this.state.detectorIsEnabled ? <WindowResizeDetector /> : null}
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('root'));

```



<https://codepen.io/hoodsey20/pen/dRLjpv> Не забудьте подключить необходимые библиотеки `React` и `ReactDOM` и примонтировать компонент `App` на страницу.

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?

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 (*Component Unmount, Unmount phase*)

Метод `componentWillUnmount` относится к последней стадией 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`.

```
componentWillUnmount() {  
  window.removeEventListener('resize', this.windowResizeHandler);  
}
```

<https://codepen.io/hoodsey20/pen/gRyjej> Проверим все еще раз. Видим, что при удалении компонента сообщения перестали появляться.

## Подведем итоги

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.

Полный перечень этапов и их методов перечислен ниже:

1. Начальный рендеринг (Mounting phase)
  - 1.1 `componentWillMount` - *обновление/вычисление локального state*, можно использовать `this.setState`
  - 1.2 `render`
  - 1.3 `componentDidMount` - *работа с AJAX, DOM и сторонними библиотеками*, можно использовать `this.setState`
2. Обновление компонента (Updating phase)
  - 2.1 `componentWillReceiveProps` - отслеживание изменения отдельных props компонента, с целью обновления локального состояния или вызова библиотечных функций, можно использовать `this.setState`
  - 2.2 `shouldComponentUpdate`
  - 2.3 `componentWillUpdate`
  - 2.4 `render`
  - 2.5 `componentDidUpdate`, можно использовать `this.setState`
3. Удаление компонента со страницы (Unmount phase)
  - 3.1 `componentWillUnmount` - *удаление ненужных обработчиков*