

# 8. Проходим методы "жизненного цикла" компонента

---

## Содержание урока

---

- Обзор;
- Что такое методы «жизненного цикла»?;
- constructor;
- componentDidMount;
- static getDerivedStateFromProps;
- shouldComponentUpdate;
- render;
- getSnapshotBeforeUpdate;
- componentDidUpdate;
- componentWillUnmount;
- componentDidCatch;
- Компонент строгого режима;
- Подведём итоги.

## Обзор

---

Привет! 🙌 На данный момент мы умеем описывать разметку JSX, создавать композицию компонентов и передавать данные между ними, управлять состоянием и ещё несколько интересных приёмов. 💪 🧑

Но что делать, когда нужно гранулированно и точно тюнинговать поведение каждого компонента? 🤔

Как сделать так, чтобы компонент сделал то, `что нужно`, в то время, `когда нужно`? ⌚

Именно это нам и предстоит узнать! 🔥

К делу! 🦊

## Что такое методы «жизненного цикла»?

---

Методы «жизненного цикла» (англ. «lifecycle methods») — это методы, вызывающиеся в определённое время на протяжении и в зависимости от фазы и стадии, в которой находится компонент. 🧠 Например, когда компонент создаётся, обновляется или удаляется из DOM.

🔍 Хозяйке на заметку:

Методы «жизненного цикла» — это одна из особенностей, доступных компонентам, реализованным в виде ES2015 классов, и недоступных компонентам, реализованным в виде функции.

Код, описанный внутри любого метода «жизненного цикла», будет выполнен в определённый, детерминированный момент времени. Зная эту деталь, можно создавать более комплексные сценарии поведения компонентов. 🏃

Существует 7 методов «жизненного цикла» React:

- `componentDidMount()` ;
- `static getDerivedStateFromProps(nextProps, prevState)` ;
- `shouldComponentUpdate(nextProps, nextState)` ;
- `getSnapshotBeforeUpdate()` ;
- `componentDidUpdate(prevProps, prevState, snapshot)` ;
- `componentWillUnmount()` ;
- `componentDidCatch(error, info)` .

👉 Совет бывалых:


Чтобы было проще понять методы «жизненного цикла», можно условно их разделить семантически:


- Методы с префиксом `will` вызываются непосредственно `перед` тем, как что-либо произойдет;
- Методы с префиксом `did` вызываются непосредственно `после` того, как что-либо произошло.

Данные методы «жизненного цикла» можно разделить на логические «стадии»:

- `Mounting` :
  - constructor;
  - static `getDerivedStateFromProps`;
  - render;
  - `componentDidMount`;
- `Updating` :
  - static `getDerivedStateFromProps`;
  - `shouldComponentUpdate`;
  - render;
  - `getSnapshotBeforeUpdate`;
  - `componentDidUpdate`;
- `Unmounting` :

- `componentWillUnmount`;
- `Error handling`:
  - `componentDidCatch`.

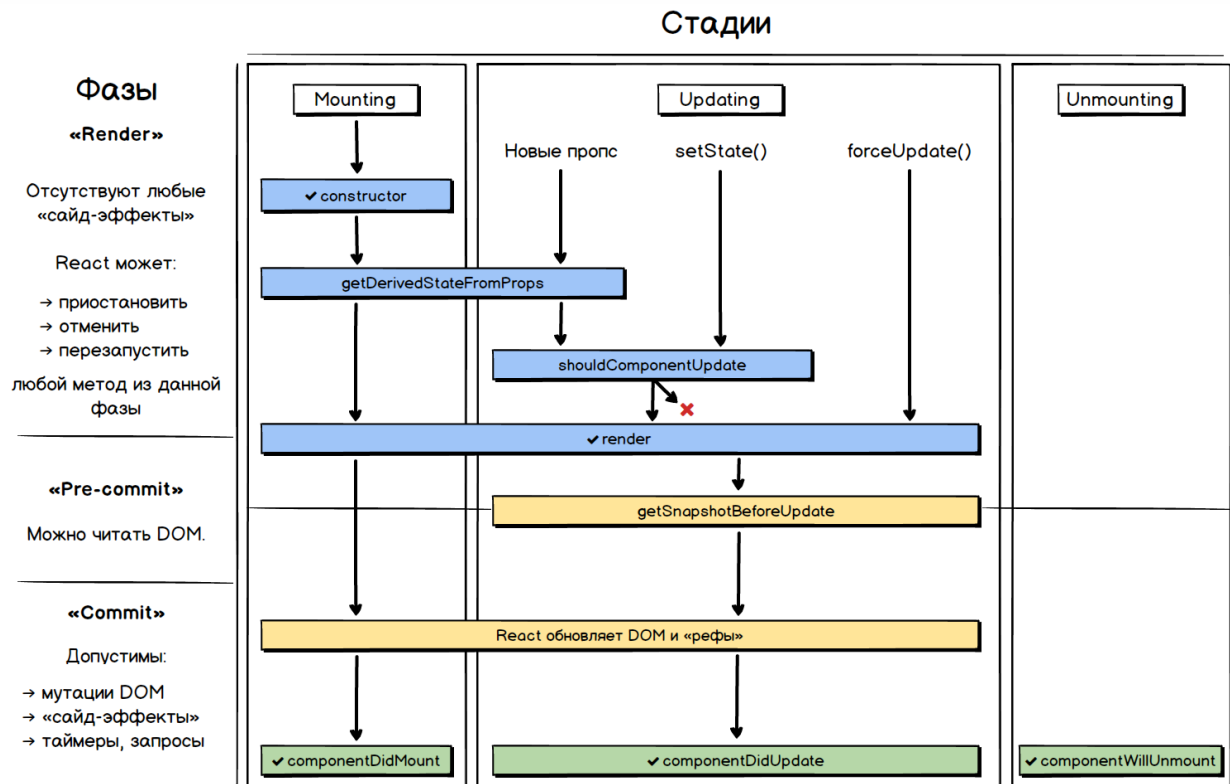
 Хозяйке на заметку:

Технически к методам «жизненного цикла» можно также отнести некоторые методы, с которыми мы уже знакомы. Например, `render`, так как этот метод также участвует в процессе «жизненного цикла» компонента, хоть и явного отношения к специализированным методам «жизненного цикла» не имеет. Плюс, учитывая, что классовой компонент — это, по сути, JavaScript-класс, у него есть конструктор, который участвует в «жизненном цикле» компонента на правах, схожих с методом `render`. Поэтому в дальнейшем рассмотрении мы будем учитывать `render` и `constructor` вместе с остальными специализированными методами «жизненного цикла». 

Также вышеописанные методы можно выделить в ещё одну категорию. Назовём её «фазы»:

- `Render`:
  - `constructor`;
  - `static getDerivedStateFromProps`;
  - `shouldComponentUpdate`;
  - `render`;
- `Pre-commit`:
  - `getSnapshotBeforeUpdate`;
- `Commit`:
  - `componentDidMount`;
  - `componentDidUpdate`;
  - `componentWillUnmount`.

Учитывая комплексность логических групп, выведенных и названных нами выше как «стадии» и «фазы», можно составить иллюстрацию для улучшения понимания происходящего:



В основе иллюстрации лежит схема, предоставленная [Dan Abramov](#), адаптированная под русский язык и формат данного конспекта. → [Оригинал](#)

Подобное разделение между «фазами» — это то, что позволит осуществлять асинхронный рендеринг, механика которого станет доступна в последующих версиях React.

👉 Совет бывалых:

Методы, изображённые на иллюстрации с «галочкой» («✓»), в реальной разработке используются чаще всего. Если тебе сложно понять данную иллюстрацию сразу — не беспокойся. Сфокусируйся на тех, что с «✓», а остальные запомнятся со временем.

Эту иллюстрацию можно использовать на протяжении текущей части конспекта как шпаргалку. 🧑🎓

## constructor

Подпись конструктора у компонента, наследующего от `React.Component`, можно выразить так:

```
1 constructor(props)
```

Конструктор компонента вызывается перед тем, как компонент замаунтится (англ. «mounting»). В конструкторе не следует:

- регистрировать подписки;

- создавать «сайд-эффекты».

Для подобных операций лучше подходит метод `componentDidMount`.

В конструкторе также можно инициализировать стейт компонента.

👉 Совет бывалых:

Перед обращением к `this` в конструкторе наследующего компонента не забывай вызвать метод `super`.

💻 Пример кода 8.1:

```
1 constructor(props) {  
2     super(props);  
3  
4     this.state = {  
5         // описание модели состояния для компонента  
6     }  
7 }
```

В примере кода 8.1 на строке кода 1 имплементируется конструктор компонента с параметром `props` и последующей передачей `props` в аргумент метода `super` на строке кода 2. Данную операцию необходимо осуществлять, когда в конструкторе необходим доступ к пропсам. А на строке кода 4 объявляется объект состояния для данного компонента.

🔍 Хозяйке на заметку:

Следуя более лаконичному синтаксису будущего, лучше объявлять состояние компонента посредством создания инициализатора свойства класса:

```
1 class Example extends Component {  
2     state = {  
3         // описание модели состояния для компонента  
4     }  
5 }
```

## static getDerivedStateFromProps

Статический метод «жизненного цикла» `getDerivedStateFromProps` вызывается во время инициализации компонента, после конструктора, а также когда компонент получает новые пропсы. 📧

🔍 Хозяйке на заметку:

Если родительский компонент триггерит ре-рендер дочернего компонента, у последнего данный метод будет вызван даже в том случае, если пропсы не изменились. А вызов `setState` внутри компонента не триггерит его `getDerivedStateFromProps`.

Этот метод может вернуть объект, описывающий модель обновления состояния, или `null` — сигнал о том, что обновление не требуется. 🙋

```
1 static getDerivedStateFromProps(nextProps, prevState) {  
2     return null; // или объект с описанием модели данных для обновления  
   стейт  
3 }
```

Для лучшего контроля из первого параметра метода можно получить доступ в виде ссылки к объекту с новыми пропсами, а со второго — к объекту прошлого состояния (до обновления компонента).

! Важно:

Из метода `static getDerivedStateFromProps` нужно обязательно вернуть «значение» (`null`) или объект с описанием обновления модели стейт. Если этого не сделать, возможно неожиданное поведение.

## shouldComponentUpdate

С помощью метода «жизненного цикла» `shouldComponentUpdate` можно информировать React, влияние измененных пропс или стейт на возвращаемую компонентом разметку. Дефолтное поведение — рендер по-умолчанию, что в целом должно подойти в большинстве ситуаций. 🧑

🔍 Хозяйке на заметку:

По сути, предназначение `shouldComponentUpdate` — это оптимизация производительности, осуществлённая путем возврата `false` из метода в том случае, когда в ре-рендере нет необходимости. Таким образом, освобождённые ресурсы CPU смогут быть направлены на выполнение более полезных инструкций.

```
1 shouldComponentUpdate(nextProps, nextState) {  
2     return true; // возврат false — выключит рендер  
3 }
```

`shouldComponentUpdate` вызовется перед рендером, когда ожидается новый объект пропс или стейт. Метод не вызывается в фазе `mounting`.

Если `shouldComponentUpdate` вернёт `false`, рендера не будет, однако это не предотвратит ре-рендер дочерних компонентов при изменении `их` стейта.

Возврат `false` из `shouldComponentUpdate` предотвратит от вызова следующие методы:

- `render`;
- `getSnapshotBeforeUpdate`;
- `componentDidUpdate`.

Альтернативой использования `shouldComponentUpdate` в явном виде является наследование компонента от `React.PureComponent`, который имплементирует `shouldComponentUpdate` с «одноуровневым сравнением по ссылке» объектов пропс и стейт по-умолчанию, однако данную опцию имеет смысл рассматривать с осторожностью, учитывая её неясность. 🤔

! Важно:

Из метода `shouldComponentUpdate` нужно обязательно вернуть «значение» — `true` или `false`. Если этого не сделать, возможно неожиданное поведение.

## render

Мы уже достаточно хорошо знакомы с этим методом, однако для закрепления понимания не будет лишним освежить знания.

Метод `render` — обязателен. При вызове он обращается к текущим пропс и стейт компонента и возвращает один из следующих типов:

- React-элемент — обычно это JSX;
- Строка или число — станут строковыми нодами в DOM;
- Порталы — их мы рассмотрим в одной из следующих частей конспекта;
- `null` и булевы значения — означают отрендерить ничего.

🔍 Хозяйке на заметку:

Метод `render` должен быть чистым. То есть не производить «сайд-эффектов» и всегда возвращать детерминированное значение. В случае, когда необходимо произвести «сайд-эффект», можно воспользоваться методом `componentDidMount`.

Однако в будущих версиях React, после реализации механизма асинхронного рендера, возможно, данное требование будет снято. 🐶

## getSnapshotBeforeUpdate

Метод `getSnapshotBeforeUpdate` вызывается непосредственно перед тем, как результат рендера коммитится в DOM (перед фазой «commit»). Данное временное окно открывает возможность компоненту снять «отпечаток» значений необходимых свойств DOM перед тем, как значение последних потенциально изменится. Например, проскроленное расстояние от начала страницы. 📏

🔍 Хозяйке на заметку:

Этот метод не вызовется, если `shouldComponentUpdate` вернёт `false`.

Возвращаемое методом значение станет третьим параметром метода «жизненного цикла» `componentDidUpdate`.

 Пример кода 8.2:

```
1  class ScrollingList extends Component {
2    listRef = React.createRef();
3
4    getSnapshotBeforeUpdate(prevProps, prevState) {
5      if (prevProps.list.length < this.props.list.length) {
6        return this.listRef.current.scrollHeight;
7      }
8
9      return null;
10   }
11
12   componentDidUpdate(prevProps, prevState, snapshot) {
13     if (snapshot !== null) {
14       this.listRef.current.scrollTop +=
15         this.listRef.current.scrollHeight - snapshot;
16     }
17   }
18
19   render() {
20     return <div ref = { this.listRef }>{ /* ...длинные СПИСОК... */
21   }</div>;
22   }
23 }
```

В примере кода 8.2 важным шагом будет прочитать свойство `scrollHeight` в методе `getSnapshotBeforeUpdate`, так как после выполнения данного метода React «закоммитит» отрендеренную разметку в DOM и `scrollHeight` получит новое, мутированное значение, в то время, когда нам необходимо знать текущее для того, чтобы правильно высчитать позицию скrolла.

**!** Важно:

Из метода `getSnapshotBeforeUpdate` нужно обязательно вернуть «значение» (`null`) или объект с описанием снимка состояния необходимой части DOM. Если этого не сделать, возможно неожиданное поведение.

## componentDidUpdate



Метод `componentDidUpdate` вызывается непосредственно после того, как React «закоммитил» отрендеренную разметку в DOM. Данный метод не вызывается при инициализации компонента.

🔍 Хозяйке на заметку:

Этот метод не вызовется, если `shouldComponentUpdate` вернёт `false`.

Подпись метода:

```
1 | componentDidUpdate(prevProps, prevState, snapshot)
```

`componentDidUpdate` хорошо подходит для операций над DOM, после обновления компонента или других «сайд-эффектов».

Если компонент имплементирует `getSnapshotBeforeUpdate` и возвращает значение из него, это значение станет доступно посредством обращения к третьему параметру метода `getSnapshotBeforeUpdate`.

## componentWillUnmount

Метод `componentWillUnmount` вызовется непосредственно в конце стадии «unmounting» — перед удалением компонента из DOM.

Этот метод — подходящее место для очистки таймеров, отмены сетевых запросов или снятия других подписок, созданных в методе `componentDidMount`.

## componentDidCatch

Метод `componentDidCatch` особенный. От других методов «жизненного цикла» отделяет связь с механикой `error boundaries` — компонентов React, служащих эквивалентом конструкции `try/catch` в JavaScript. 🕵️

```
1 | componentDidCatch(error, info)
```

`Error boundaries` «ловят» необработанные ошибки, возбуждённые дочерними компонентами ниже по дереву компонентов, и могут эти ошибки логировать и отображать фолбэкный UI.

`Error boundaries` пока что могут «ловить» ошибки только в:

- Методе `render`;
- Методах «жизненного цикла»;
- Конструкторе.

🔍 Хозяйке на заметку:

Если в React-приложении пропустить необработанную ошибку, приложение удалится из DOM на 100%.

Классовый компонент, имплементирующий метод "жизненного цикла"

`componentDidCatch`, автоматически становится `error boundary`.

🖥️ Пример кода 8.3:

```
1  import React, { Component } from 'react';
2  import { logErrorToMyService } from '../helpers';
3
4  class ErrorBoundary extends Component {
5      state = {
6          hasError: false
7      }
8
9      componentDidCatch (error, info) {
10         this.setState(() => ({
11             hasError: true
12         }));
13
14         logErrorToMyService(error, info);
15     }
16
17     render () {
18         if (this.state.hasError) {
19
20             return <h1>Something went wrong.</h1>;
21         }
22
23         return this.props.children;
24     }
25 }
```

В примере кода 8.3 на строке кода 9 объявлен метод «жизненного цикла» `componentDidCatch`. Первый параметр — строка `error` — содержит описание ошибки. Второй параметр — объект `info` — содержит `stack trace`, сгенерированный исключением. 🚗

Компонент `ErrorBoundary` можно использовать в следующем виде:

```
1  <ErrorBoundary>
2    <App />
3  </ErrorBoundary>
```

! Важно:

`Error boundaries` наследуют поведение конструкции `try/catch` ввиду того, что могут «ловить» ошибки только в компонентах, находящихся **ниже** в дереве компонентов. `Error boundary` не умеют "ловить" ошибки на одном уровне с собой.

## Компонент строгого режима

В `React v16.3` ввели новый отладочный компонент, помогающий отследить потенциальные проблемные части приложения. 🚔 Наподобие `Fragment`, компонент `StrictMode` не рендерит никакого UI, однако включает дополнительные проверки для дочерних компонентов.

 Пример кода 8.4:

```
1 import React from 'react';
2 import Header from './components'
3
4 export default class App extends React.Component {
5   render () {
6     return (
7       <React.StrictMode>
8         <Header />
9       </React.StrictMode/>
10    );
11  }
12 }
```

В примере кода 8.4 на строке кода 7 использован компонент строгого режима `strictMode`. В результате React сообщит о любых местах, которые потенциально не вписываются в будущую модель асинхронного рендеринга и могут вызывать проблемы.



## Подведём итоги

React открывает нам особый API для гранулированного тюнинга поведением компонента. Этот API называется методы «жизненного цикла».

Эти методы можно понять, распределив их по двум категориям:

1. «Стадии»:

- `Mounting`:
  - constructor;
  - static `getDerivedStateFromProps`;
  - render;
  - `componentDidMount`;

- `Updating` :
  - `static getDerivedStateFromProps`;
  - `shouldComponentUpdate`;
  - `render`;
  - `getSnapshotBeforeUpdate`;
  - `componentDidUpdate`;
- `Unmounting` :
  - `componentWillUnmount`;
- `Error handling` :
  - `componentDidCatch`;

## 2. «Фазы» :

- `Render` :
  - `constructor`;
  - `static getDerivedStateFromProps`;
  - `shouldComponentUpdate`;
  - `render`;
- `Pre-commit` :
  - `getSnapshotBeforeUpdate`;
- `Commit` :
  - `componentDidMount`;
  - `componentDidUpdate`;
  - `componentWillUnmount`.

Существует также особенный метод `«жизненного цикла»`, помогающий обрабатывать исключения в React, однако пока что еще не везде.

Также существует компонент-отладчик `StrictMode`, сообщающий о всех местах, потенциально вызывающих проблемы после внедрения механизма асинхронного рендеринга.

Спасибо, что остаёшься с нами! 🙌 В следующей части конспекта мы поработаем с `«синтетическими событиями»` React. До встречи! 🤖

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