# Основные понятия

## Знакомство с JSX

Объявление переменной, которая содержит JSX:



JSX производит «элементы» React.

React разделяет ответственность с помощью слабо связанных единиц, называемых «компоненты», которые содержат и разметку, и логику.

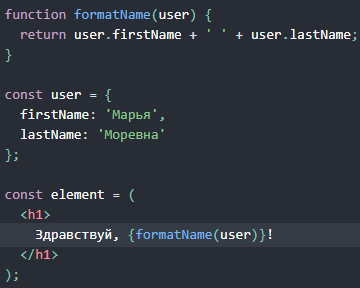
### Встраивание js выражений в JSX

В следующем примере мы объявляем переменную name и затем используем её внутри JSX, обрамляя фигурными скобками:



JSX допускает использование любых корректных JavaScript-выражений внутри фигурных скобок. Например, 2 + 2, user.firstName и formatName(user) являются допустимыми выражениями.

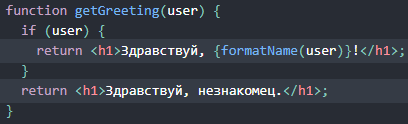
В примере ниже мы встраиваем результат вызова JavaScript-функции formatName(user) в элемент <h1>:



### JSX это тоже выражение

После компиляции каждое JSX-выражение становится обычным вызовом JavaScript-функции, результат которого — объект JavaScript.

Из этого следует, что JSX можно использовать внутри выражений if и циклов for, присваивать переменным, передавать функции в качестве аргумента и возвращать из функции.



### Использование атрибутов JSX

Чтобы использовать строковый литерал в качестве значения атрибута, используются кавычки:



Если же в значении атрибута требуется указать JavaScript-выражение, то используют фигурные скобки:



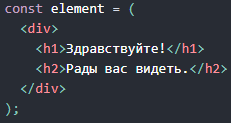
Для атрибутов используется стиль именования camelCase. Например, class становится *className* в JSX, а tabindex становится *tabIndex*.

### Использование дочерних элементов в JSX

Если тег пуст, то его можно сразу же закрыть с помощью /> точно так же, как и в XML:



Но JSX-теги могут и содержать дочерние элементы:



### JSX предотвращает атаки, основанные на инъекции кода

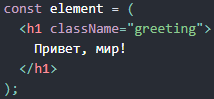
Данные, введённые пользователем, можно безопасно использовать в JSX:

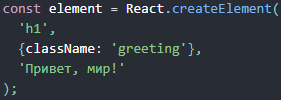


По умолчанию React DOM экранирует все выражения, включённые в JSX перед тем как отрендерить их.

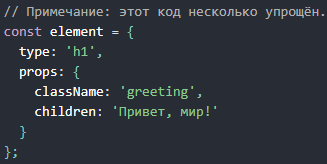
### JSX представляет собой объекты

Babel компилирует JSX в вызовы React.createElement(). Следующие два примера кода эквивалентны между собой:





React.createElement() проводит некоторые проверки с целью выявить баги в коде, но главное — создаёт объект похожий на такой:



## Рендеринг элементов

Элементы — мельчайшие кирпичики React-приложений, они описывают то, что вы хотите увидеть на экране:



В отличие от DOM-элементов, элементы React — это простые объекты, не отнимающие много ресурсов. React DOM обновляет DOM, чтобы он соответствовал переданным React-элементам.

**Примечание:**

Элементы можно перепутать с более известной концепцией «компонентов». С компонентами мы ознакомимся в [следующей главе](https://ru.reactjs.org/docs/components-and-props.html). Элементы — это то, «из чего сделаны» компоненты.

### Рендеринг элемента в DOM

Допустим, в вашем HTML-файле есть <div>:



Мы назовём его «корневым» узлом DOM, так как React DOM будет управлять его содержимым.

Обычно в приложениях, написанных полностью на React, есть только один корневой элемент. При встраивании React в существующее приложение вы можете рендерить во столько независимых корневых элементов, во сколько посчитаете нужным.

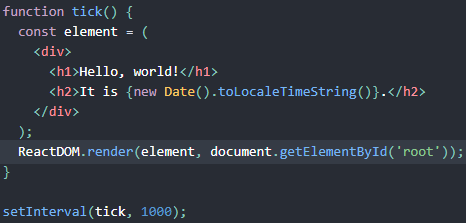
Для рендеринга React-элемента в корневой узел DOM вызовите [ReactDOM.render()](https://ru.reactjs.org/docs/react-dom.html#render) с React-элементом и корневым DOM узлом в качестве аргументов:



### Обновление элементов на странице

Элементы React [иммутабельны](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82). После создания элемента нельзя изменить его потомков или атрибуты. Элемент похож на кадр в фильме: он отражает состояние интерфейса в конкретный момент времени.

Пока что, мы знаем только один способ обновить интерфейс — это создать новый элемент и передать его в [ReactDOM.render()](https://ru.reactjs.org/docs/react-dom.html#render). Рассмотрим пример с часами:



В этом примере ReactDOM.render() вызывается каждую секунду с помощью колбэка в setInterval().

Примечание: На практике большинство React-приложений вызывают ReactDOM.render() только один раз. В следующем разделе вы узнаете, как можно обновлять интерфейс при помощи компонента с состоянием.

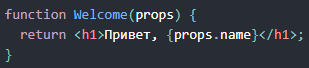
### React обновляет только то, что необходимо

React DOM сравнивает элемент и его дочернее дерево с предыдущей версией и вносит в DOM только минимально необходимые изменения.

## Компоненты и пропсы

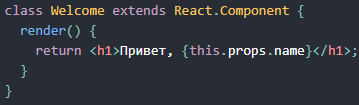
Во многом компоненты ведут себя как обычные функции JS. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают *React-элементы*, описывающие, что мы хотим увидеть на экране.

### Функциональные и классовые компоненты



Эта функция — компонент, потому что она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент.

Ещё компоненты можно определять как [классы ES6](https://developer.mozilla.org/ru-RU/docs/Web/JavaScript/Reference/Classes):



С точки зрения React, эти два компонента эквивалентны.

### Как отрендерить компонент

Элементы могут описывать и наши собственные компоненты, а не только DOM-теги:



Когда React встречает подобный элемент, он собирает все JSX-атрибуты и **дочерние элементы** в один объект и передаёт их нашему компоненту. Этот объект называется **«пропсы» (props)**.

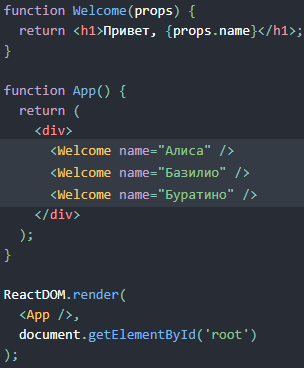
**Примечание: Всегда называйте компоненты с заглавной буквы.**

Если компонент начинается с маленькой буквы, React принимает его за DOM-тег. Например, <div /> это div-тег из HTML, а <Welcome /> это уже наш компонент Welcome, который должен быть в области видимости.

### Композиция компонентов

Компоненты могут ссылаться на другие компоненты в возвращённом ими дереве. Это позволяет нам использовать одну и ту же абстракцию — компоненты — на любом уровне нашего приложения. Неважно, пишем ли мы кнопку, форму или целый экран: все они, как правило, представляют собой компоненты в React-приложениях.

Например, компонент App может отрендерить компонент Welcome несколько раз:



В приложениях, написанных на React с нуля, как правило, есть один компонент App, который находится на самом верху. В случае, если вы переписываете существующее приложение на React, имеет смысл начать работу с маленького компонента типа Button и постепенно двигаться «вверх» по иерархии.

### Пропсы можно только читать

Компонент никогда не должен что-то записывать в свои пропсы — вне зависимости от того, [функциональный он или классовый](https://ru.reactjs.org/docs/components-and-props.html#function-and-class-components).

Функция (компонент) должна быть чистой – то есть, в том числе, она не должна записывать данные в свои же аргументы.

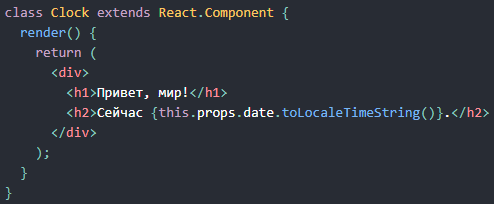
## Состояние и жизненный цикл

В главе «Рендеринг элементов» был пример с часами, которые перерисовывались с помощью ReactDom.render(); Далее будет показано, как сделать, чтобы компонент сам себя обновлял.

### Состояние

«Состояние» очень похоже на уже знакомые нам пропсы, отличие в том, что состояние контролируется и доступно только конкретному компоненту.

Необходимо создать классовый компонент:



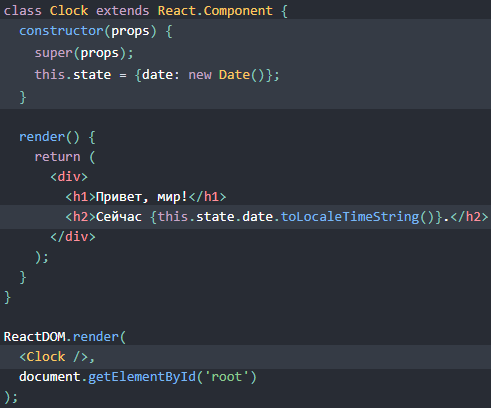
Обратите внимание: ***this***.props

Метод render будет вызываться каждый раз, когда происходит обновление. Так как мы рендерим <Clock /> в один и тот же DOM-контейнер, мы используем единственный экземпляр класса Clock — поэтому мы можем задействовать внутреннее состояние и методы жизненного цикла.

### Добавим внутреннее состояние в класс

Переместим date из пропсов в состояние в три этапа:

1. Заменим this.props.date на this.state.date в методе render()
2. Добавим [конструктор класса](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes#Constructor), в котором укажем начальное состояние в переменной **this.state**
3. Удалим пропс date из элемента <Clock />:



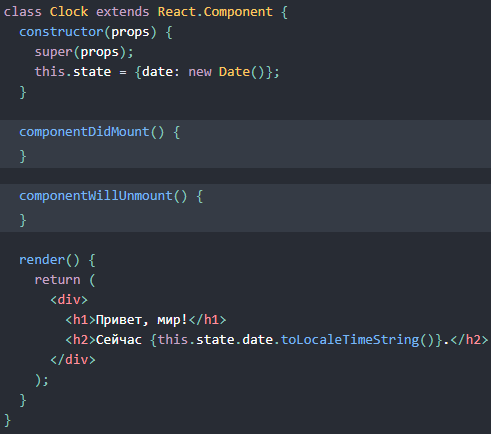
### Добавим методы жизненного цикла в класс

В приложениях со множеством компонентов важно освобождать используемые системные ресурсы, когда компоненты удаляются.

Первоначальный рендеринг компонента в DOM называется «**монтирование**» (mounting). Нужно [устанавливать таймер](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) всякий раз, когда это происходит.

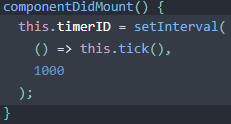
Когда DOM-узел, созданный компонентом, удаляется, происходит «**размонтирование**» (unmounting). Чтобы избежать утечки ресурсов, мы будем [сбрасывать таймер](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) при каждом «размонтировании». Т.к. таймер после удаления ссылки на него (ID) остается существовать, лол.

Объявим специальные методы, которые компонент будет вызывать при монтировании и размонтировании:



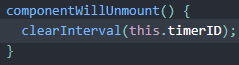
Эти методы называются «методами жизненного цикла» (lifecycle methods).

Метод **componentDidMount()** запускается после того, как компонент отрендерился в DOM — здесь мы и установим таймер:



Поля **this.props** и **this.state** в классах — особенные, и их устанавливает сам React. Вы можете вручную добавить новые поля, если компоненту нужно хранить дополнительную информацию (например, ID таймера).

Теперь нам осталось сбросить таймер в методе жизненного цикла **componentWillUnmount():**



Наконец, реализуем метод tick(). Он запускается таймером каждую секунду и вызывает **this.setState().**

**this.setState()** планирует обновление внутреннего состояния компонента:



### Алгоритм работы компоненты из примера

1. Когда мы передаём <Clock /> в ReactDOM.render(), React вызывает конструктор компонента. Clock должен отображать текущее время, поэтому мы задаём начальное состояние this.state объектом с текущим временем.
2. React вызывает метод render() компонента Clock. Таким образом React узнаёт, что отобразить на экране. Далее React обновляет DOM так, чтобы он соответствовал выводу рендера Clock.
3. Как только вывод рендера Clock вставлен в DOM, React вызывает метод жизненного цикла componentDidMount(). Внутри него компонент Clock указывает браузеру установить таймер, который будет вызывать tick() раз в секунду.
4. Таймер вызывает tick() ежесекундно. Внутри tick() мы просим React обновить состояние компонента, вызывая setState() с текущим временем. React реагирует на изменение состояния и снова запускает render(). На этот раз this.state.date в методе render() содержит новое значение, поэтому React заменит DOM. Таким образом компонент Clock каждую секунду обновляет UI.
5. Если компонент Clock когда-либо удалится из DOM, React вызовет метод жизненного цикла componentWillUnmount() и сбросит таймер.

### Как правильно использовать состояние

* **Не изменяйте состояние напрямую**

В следующем примере повторного рендера не происходит:

****

Вместо этого используйте *setState()*:

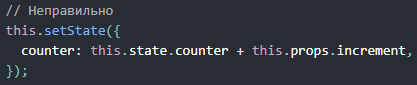


Конструктор — это **единственное** место, где вы можете присвоить значение **this.state** напрямую.

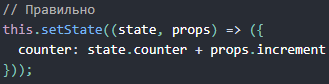
* **Обновления состояния могут быть асинхронными**

Поскольку **this.props** и **this.state** могут обновляться асинхронно, вы не должны полагаться на их текущее значение для вычисления следующего состояния.

Например, следующий код может не обновить счётчик:

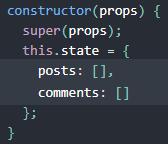


Правильно будет использовать вариант вызова setState(), который **принимает функцию**, а не объект. Эта функция получит предыдущее состояние в качестве первого аргумента и значения пропсов непосредственно во время обновления в качестве второго аргумента:



* **Обновления состояния объединяются**

Когда мы вызываем setState(), React объединит аргумент (новое состояние) c текущим состоянием.

****

Их можно обновлять по отдельности с помощью отдельных вызовов setState():



Состояния объединяются поверхностно, поэтому вызов this.setState({comments}) оставляет this.state.posts нетронутым, но полностью заменяет this.state.comments.

* **Однонаправленный поток данных**

В иерархии компонентов ни родительский, ни дочерние компоненты не знают, задано ли состояние другого компонента. Также не важно, как был создан определённый компонент — с помощью функции или с помощью класса.

Состояние часто называют «локальным», «внутренним» или инкапсулированным. Оно доступно только для самого компонента и скрыто от других.

Компонент может передать своё состояние вниз по дереву в виде пропсов дочерних компонентов:



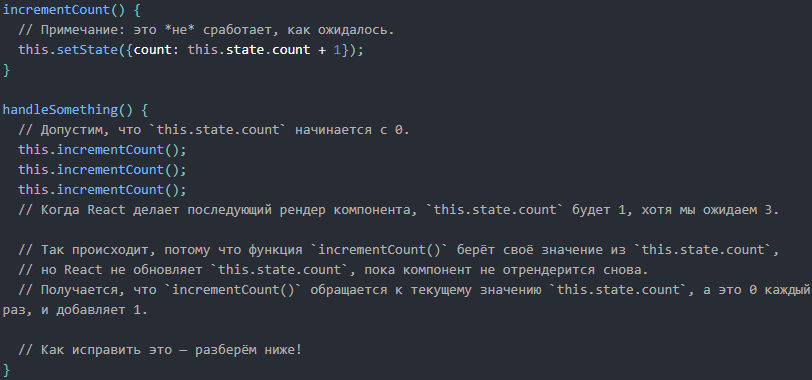
Компонент FormattedDate получает date через *пропсы*, но он не знает, откуда они взялись изначально — из *состояния* Clock, *пропсов* Clock или просто JavaScript-выражения.

### Об асинхронности this.setState

**Почему setState даёт неверное значение?**

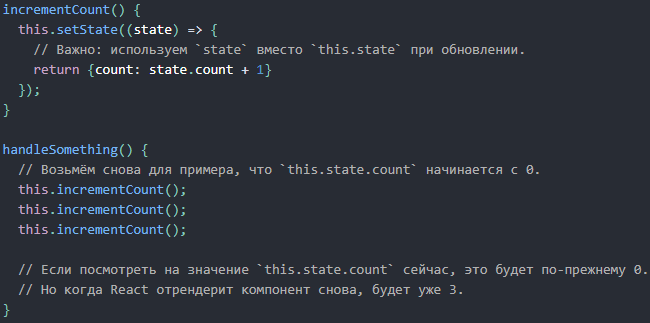
В React как this.props, так и this.state представляют значения, которые *уже были отрендерены*, например, то, что видите на экране.

Вызовы setState являются асинхронными, поэтому не стоит рассчитывать, что this.state вернет новое значение мгновенно после вызова setState. Необходимо добавить функцию, которая **сработает только после обновления состояния**, если нужно получить новое значение, основанное на текущем состоянии (ниже ошибочный пример).

****

**Как обновить состояние значениями, которые зависят от текущего состояния?**

Нужно добавить функцию вместо объекта к setState, которая будет срабатывать только на самой последней версии состояния (пример ниже).

****

**В чём разница между добавлением объекта или функции к setState?**

Добавление функции **даёт вам доступ к текущему состоянию внутри самой функции.** Так как setState вызовы «сгруппированы», это помогает связать изменения и гарантирует, что они будут выполняться друг за другом, а не конфликтовать.

<https://ru.reactjs.org/docs/faq-state.html#what-is-the-difference-between-state-and-props> и <https://ru.reactjs.org/docs/react-component.html#setstate>

## Обработка событий

### Особенности событий в React

Обработка событий в React-элементах очень похожа на обработку событий в DOM-элементах. Но есть несколько синтаксических отличий:

* События в React именуются в стиле *camelCase* вместо нижнего регистра.
* С JSX вы передаёте функцию как *обработчик события* вместо строки

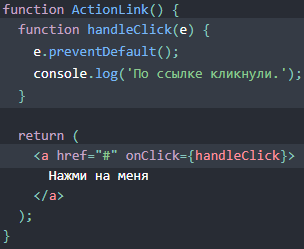
**HTML:**



**React:**



Ещё одно отличие — в React нельзя предотвратить обработчик события по умолчанию, вернув false. Нужно явно вызвать preventDefault:



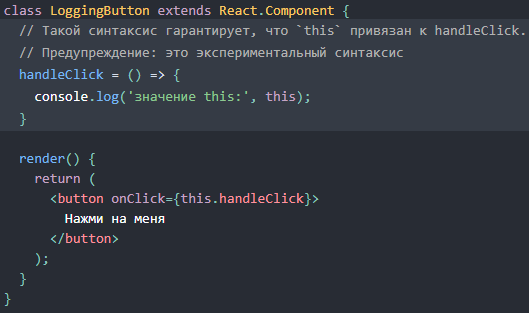
В приведённом выше коде ***e*** — это синтетическое событие. События React работают не совсем как нативные. Изучите руководство о **SyntheticEvent**, чтобы узнать о них больше.

При использовании React обычно не нужно вызывать addEventListener.

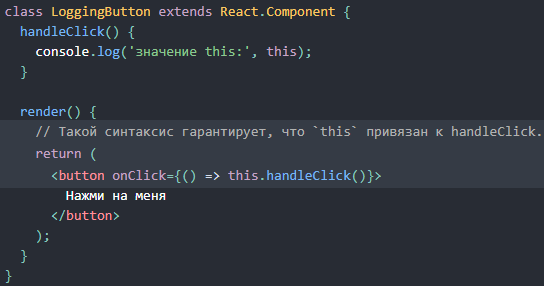
В компоненте, определённом с помощью ES6-класса, в качестве обработчика события обычно выступает один из методов класса:



* Если вы забудете привязать метод this.handleClick и передать его в onClick, значение this будет undefined в момент вызова функции. Обычно, если ссылаться на метод без () после него, например, onClick={this.handleClick}, этот метод нужно привязать.
* вы можете использовать экспериментальныq [синтаксис общедоступных полей классов](https://babeljs.io/docs/plugins/transform-class-properties/), чтобы правильно привязать колбэки:



* Можете попробовать стрелочные функции в колбэке:



Проблема этого синтаксиса в том, что при каждом рендере LoggingButton создаётся новый колбэк. Чаще всего это не страшно. Однако, если этот колбэк попадает как проп в дочерние компоненты, эти компоненты могут быть отрендерены снова.

### Передача аргументов в обработчики событий

Внутри цикла часто нужно передать дополнительный аргумент в обработчик события. Например, если id — это идентификатор строки, можно использовать следующие варианты:



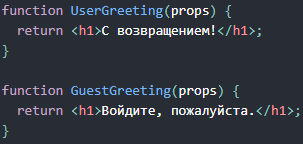
Обе строки эквивалентны. Используя стрелочную функцию, необходимо передавать аргумент явно, но с bind любые последующие аргументы передаются автоматически.

## Условный рендеринг

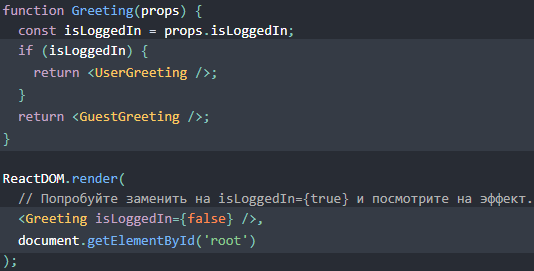
React позволяет разделить логику на независимые компоненты. Эти компоненты можно показывать или прятать в зависимости от текущего состояния.

### Пример

Имеет два компонента: Паттерн



Можно создать компонент Greeting, который отражает один из этих компонентов в зависимости от того, выполнен ли вход на сайт:

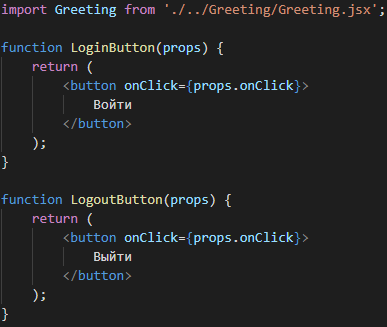


### Переменные-элементы

Элементы React можно сохранять в переменных. Это удобно, когда какое-то условие определяет, надо ли рендерить одну часть компонента или нет, а другая часть компонента остаётся неизменной.

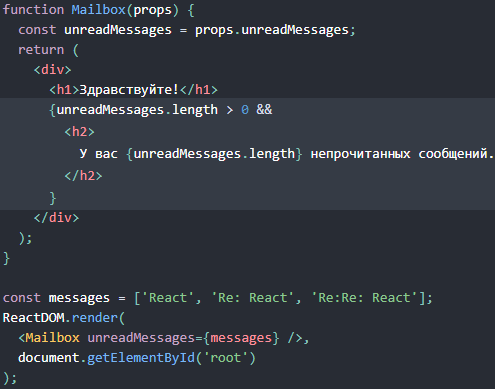
**Пример:**

LoginControl будет рендерить либо <LoginButton />, либо <LogoutButton /> в зависимости от текущего состояния. А ещё он будет всегда рендерить <Greeting /> из предыдущего примера. Паттерн

****

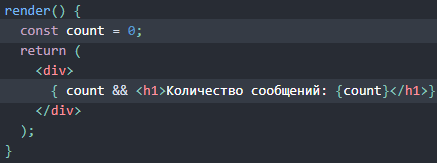
### Встроенные условия if с логическим оператором &&

Вы можете внедрить любое выражение в JSX, заключив его в фигурные скобки. Это правило распространяется и на логический оператор && языка JavaScript, которым можно удобно вставить элемент в зависимости от условия: паттерн



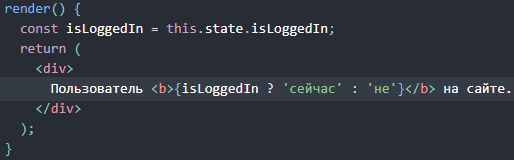
Это работает, потому что «&&» находит и возвращает первое ложное значение, либо, если все операнды были истинными, возвращается последний.

Обратите внимание, что ложное выражение, как ожидается, пропустит элемент после &&, но при этом выведет результат этого выражения. В примере ниже метод render вернёт <div>0</div>:



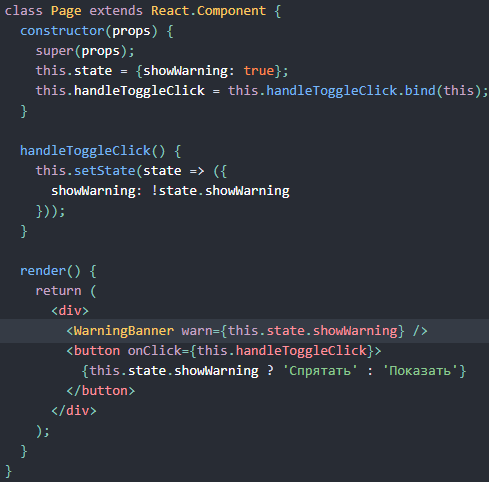
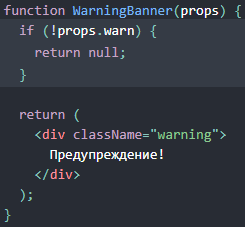
### Встроенные условия if-else с тернарным оператором

Есть ещё один способ писать условия прямо в JSX. Вы можете воспользоваться тернарным оператором [condition ? true : false](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80):



### Предотвращение рендеринга компонента

В редких случаях может потребоваться позволить компоненту спрятать себя, хотя он уже и отрендерен другим компонентом. Чтобы этого добиться, верните null вместо того, что обычно возвращается на рендеринг: паттерн

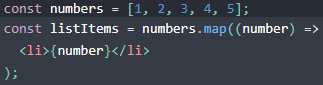


## Списки и ключи

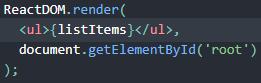
### Рендер нескольких компонентов

Вы можете создать коллекцию элементов и [встроить её в JSX](https://ru.reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx) с помощью фигурных скобок { }.

К примеру, пройдём по массиву numbers, используя функцию JavaScript [map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map), и вернём элемент <li> в каждой итерации. Получившийся массив элементов сохраним в listItems:



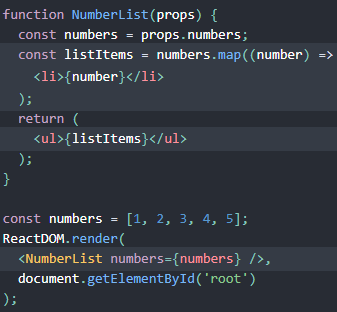
Теперь мы включим массив listItems внутрь элемента <ul> и [отрендерим его в DOM](https://ru.reactjs.org/docs/rendering-elements.html#rendering-an-element-into-the-dom):



### Простой компонент-список

Как правило, вы будете рендерить списки внутри какого-нибудь [компонента](https://ru.reactjs.org/docs/components-and-props.html).

Мы можем отрефакторить предыдущий пример с использованием компонента, который принимает массив numbers и выводит список элементов:

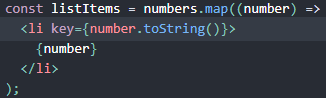


Когда вы запустите данный код, то увидите предупреждение о том, что у каждого элемента массива должен быть ключ (key):



**«Ключ»** — это специальный строковый атрибут, который нужно указывать при создании списка элементов.

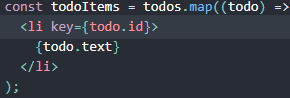
Добавим каждому элементу в списке атрибут key:



### Ключи

Ключи помогают React определять, какие элементы были изменены, добавлены или удалены. Их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени.

Лучший способ выбрать ключ — это использовать строку, которая будет явно отличать элемент списка от его соседей. Чаще всего вы будете использовать ID из ваших данных как ключи:



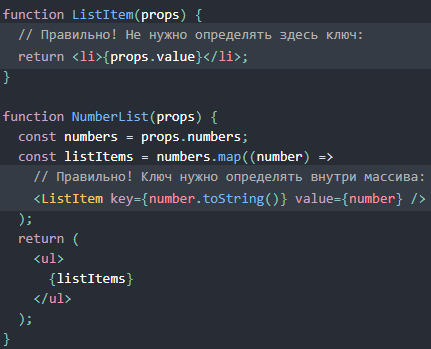
### Извлечение компонентов с ключами

* **Ключи всегда определяем внутри Map:**

Неправильно:



Правильно:



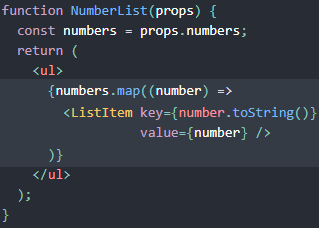
* **Ключи должны быть уникальными среди соседних элементов**

Ключи внутри массива должны быть уникальными только среди своих соседних элементов. Им не нужно быть уникальными глобально. Можно использовать один и тот же ключ в двух разных массивах.

*Ключи служат подсказками для React, но они никогда не передаются в ваши компоненты.*

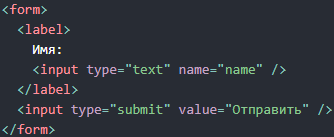
### Встраивание map() в JSX

JSX позволяет [встроить любое выражение](https://ru.reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx) в фигурные скобки, так что мы можем включить результат выполнения map():

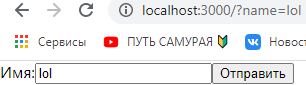


## Формы

В React HTML-элементы формы ведут себя несколько отлично от остальных DOM-элементов, так как у элементов формы изначально есть внутреннее состояние. К примеру, в эту HTML-форму можно ввести имя:



По умолчанию браузер переходит на другую страницу при отправке HTML-форм, в том числе и этой:



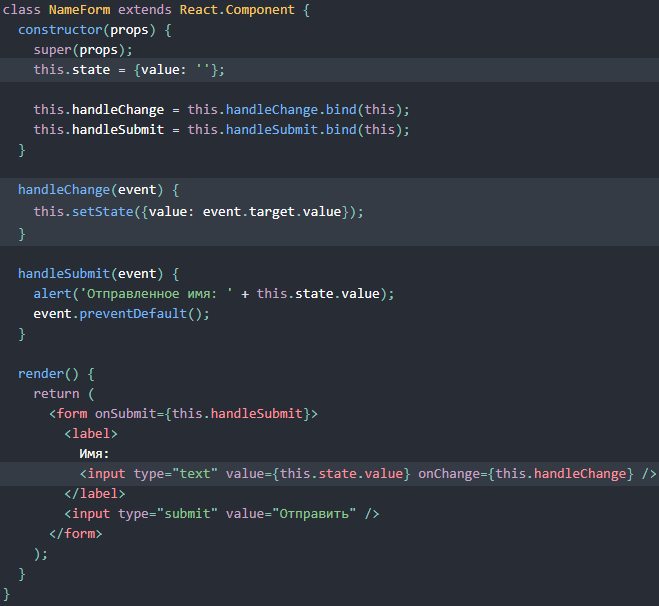
Однако, чаще форму удобнее обрабатывать с помощью JS-функции, у которой есть доступ к введённым данным. Стандартный способ реализации такого поведения называется «управляемые компоненты».

### Управляемые компоненты

В **HTML** элементы формы, такие как <input>, <textarea> и <select>, обычно сами управляют своим состоянием и обновляют его когда пользователь вводит данные. В **React** мутабельное состояние обычно содержится в свойстве state и обновляется только через [setState()](https://ru.reactjs.org/docs/react-component.html#setstate).

Можно скомбинировать оба подхода и сделать состояние React-компонента *«единственным источником правды»*. Тогда React-компонент будет рендерить форму и контролировать её поведение в ответ на пользовательский ввод. Значение элемента формы input в этом случае *будет контролировать React*, а сам элемент будет называться **«управляемый компонент».**

**Пример:**



Мы установили атрибут value для поля ввода, и теперь в нём всегда будет отображаться значение this.state.value. Состояние React-компонента стало «источником истины». А так как каждое нажатие клавиши вызывает handleChange, который обновляет состояние React-компонента, значение в поле будет обновляться по мере того, как пользователь печатает.

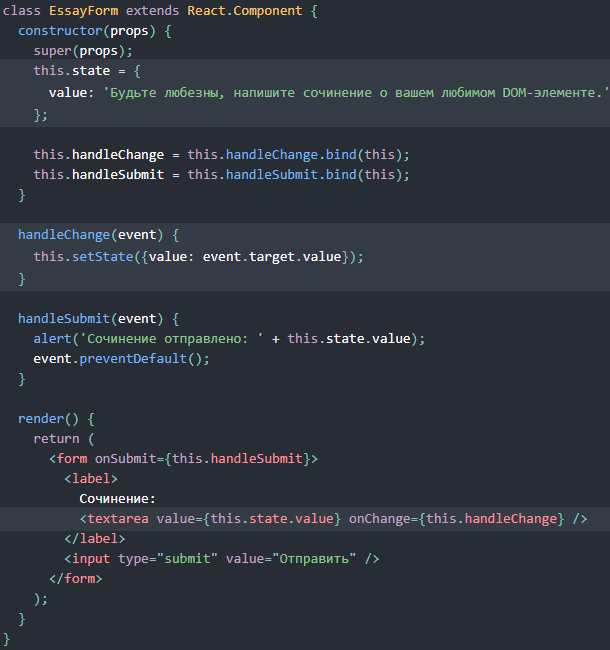
В управляемом компоненте значение поля ввода всегда определяется состоянием React. Хотя это означает, что вы должны написать немного больше кода, теперь вы сможете передать значение и другим UI-элементам или сбросить его с других обработчиков событий.

### Тег textarea

**HTML**-элемент *<textarea>* в качестве текста отображает дочерний элемент:

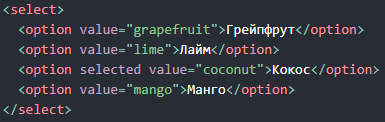


**В React** *<textarea>* использует атрибут *value.* Таким образом, форму с <textarea> можно написать тем же способом, что и форму с однострочным <input>:

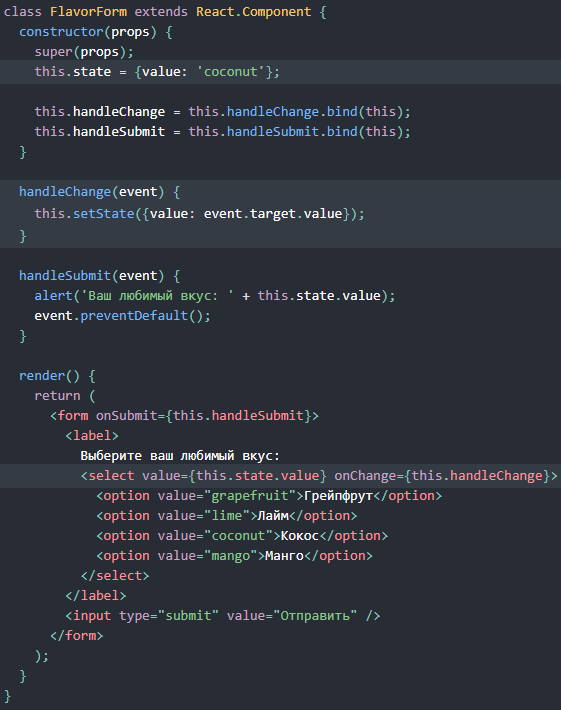


### Тег select

В **HTML** <select> создаёт выпадающий список. HTML-код в этом примере создаёт выпадающий список вкусов:



«Кокос» выбран по умолчанию из-за атрибута selected. React вместо этого атрибута использует value в корневом теге select. В управляемом компоненте так удобнее, потому что обновлять значение нужно только в одном месте (state). Пример:



**Примечание**

В атрибут value можно передать массив, что позволит выбрать несколько опций в теге select:



Подводя итог, <input type="text">, <textarea>, и <select> работают очень похоже. Все они принимают атрибут value, который можно использовать, чтобы реализовать управляемый компонент.

### Загрузка файла

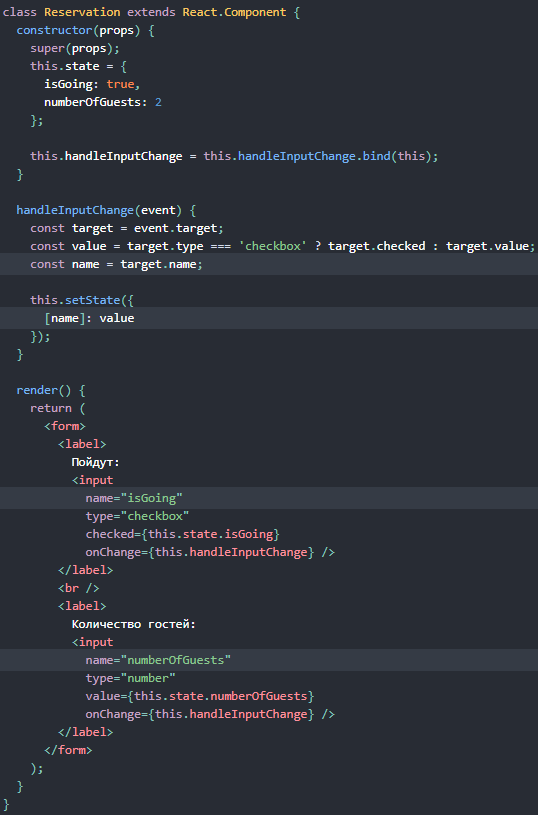
В HTML <input type="file"> позволяет пользователю выбрать один или несколько файлов для загрузки с устройства на сервер или управлять им через JavaScript с помощью [File API](https://developer.mozilla.org/ru/docs/Web/API/File/Using_files_from_web_applications).



Так как значение такого элемента доступно только для чтения, это **неуправляемый** React-компонент.

### Обработка нескольких элементов input

Если вам нужны несколько управляемых элементов input, вы можете назначить каждому из них атрибут name, что позволит функции-обработчику решать, что делать, основываясь на значении event.target.name:



### Значение null управляемого компонента

Если установить [управляемому компоненту](https://ru.reactjs.org/docs/forms.html#controlled-components) проп value, то пользователь не сможет изменить его значение без вашего желания. Однако если value задать как undefined или null то его по-прежнему можно редактировать:



### Полноценные решения

Если вы ищете полноценное решение, которое может валидировать ввод, запомнить посещённые поля формы и обработать её отправку, присмотритесь к [Formik](https://jaredpalmer.com/formik). Эта библиотека построена на принципах управляемых компонентов и управления состоянием, так что не пренебрегайте их изучением.

### Альтернативы управляемым компонентам

[неуправляемые компоненты](https://ru.reactjs.org/docs/uncontrolled-components.html) — альтернативная техника реализации ввода данных в форму.

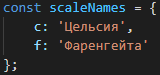
## Подъём состояния

### Проблема

Часто несколько компонентов должны отражать одни и те же изменяющиеся данные. Мы рекомендуем поднимать общее состояние до ближайшего общего предка.

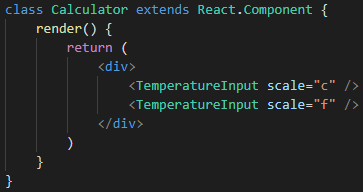
**Пример**

* Имеем компонент **TemperatureInput:**





* Рендерим два таких компонента в другом компоненте:



Сейчас у нас есть два поля, но когда вводим температуру в одно из них, другое поле не обновляется (нет синхронизации).

Также компонент **Calculator** захочет отрендерить еще один компонент, на основании введенной темпрературы в одно из полей, но пока это невозможно.

### Поднятие состояния

В React совместное использование состояния достигается перемещением его до ближайшего предка компонентов, которым оно требуется. Это называется «подъём состояния». Мы удалим внутреннее состояние из TemperatureInput и переместим его в Calculator.

Если Calculator владеет общим состоянием, он становится «источником истины» текущей температуры для обоих полей ввода. Он может предоставить им значения, которые не противоречат друг другу. Поскольку пропсы обоих компонентов TemperatureInput приходят из одного и того же родительского компонента Calculator, два поля ввода будут всегда синхронизированы.

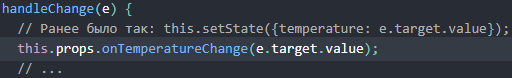
### Шаги поднятия

1. Во-первых, мы заменим *this.state.temperature* на *this.props.temperature (****value****)*

[Пропсы доступны только для чтения](https://ru.reactjs.org/docs/components-and-props.html#props-are-read-only). Когда temperature находилась во внутреннем состоянии, TemperatureInput мог вызвать this.setState() для изменения его значения. Теперь, когда temperature находится в родительском компоненте в качестве пропа, TemperatureInput не может контролировать его.

1. Прокидываем обработчик *onChange* через проп:

Теперь, когда TemperatureInput хочет обновить свою температуру, он вызывает this.props.onTemperatureChange:



Пропсы onTemperatureChange и temperature будут предоставлены родительским компонентом Calculator. Он будет обрабатывать изменения, модифицируя собственное внутреннее состояние, тем самым повторно отрендеривая оба поля ввода с новыми значениями.

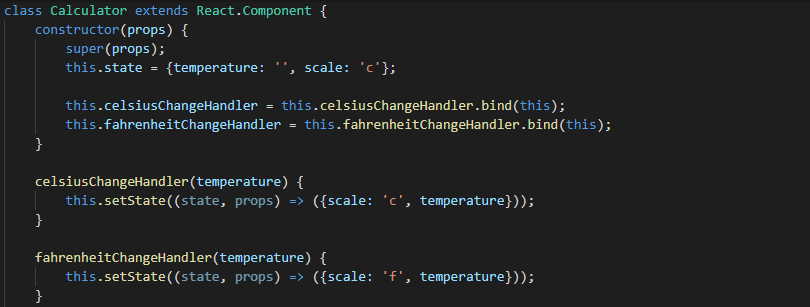
**Примечание:**

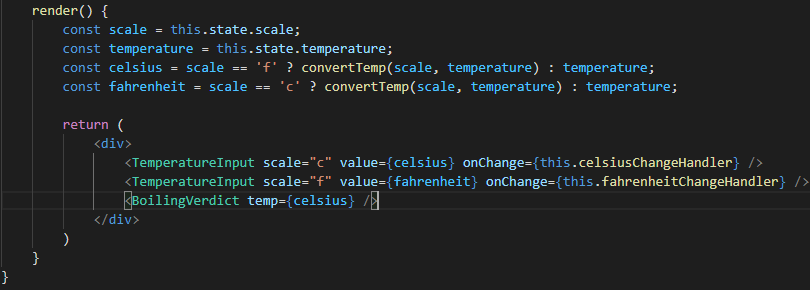
В пользовательских компонентах нет особого смысла в именах пропсов temperature или onTemperatureChange. Мы могли бы назвать их как-то иначе, например, value и onChange, т. к. подобные имена — распространённое **соглашение**.

### Изменение родительского компонента

Теперь перейдём к компоненту ***Calculator***.

Мы будем хранить текущие значения temperature и scale во внутреннем состоянии этого компонента.





* ***TemperatureInput:***



Компонент TemperatureInput меняет стейт Calculator с помощью метода, который в него прокинул Calculator. Значение второго инпута рассчитывается по формуле на основании первого инпута (поэтому нет необходимости держать в стейте значения обоих полей).

### Итого

Для любых изменяемых данных в React-приложении должен быть один «источник истины». Обычно состояние сначала добавляется к компоненту, которому оно требуется для рендера. Затем, если другие компоненты также нуждаются в нём, вы можете поднять его до ближайшего общего предка. Вместо того, чтобы пытаться синхронизировать состояние между различными компонентами, вы должны полагаться на [однонаправленный поток данных](https://ru.reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down).

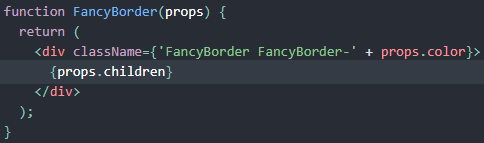
## Композиция против наследования

React имеет мощную модель композиции, поэтому для повторного использования кода между компонентами мы рекомендуем использовать композицию вместо наследования.

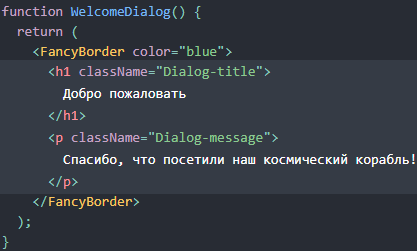
### Вставка children

Некоторые компоненты не знают своих потомков заранее. Это характерно для таких компонентов, как Sidebar или Dialog, которые представляют из себя «коробку», в которую можно что-то положить.

Для таких компонентов мы рекомендуем использовать специальный проп children, который передаст дочерние элементы сразу на вывод:

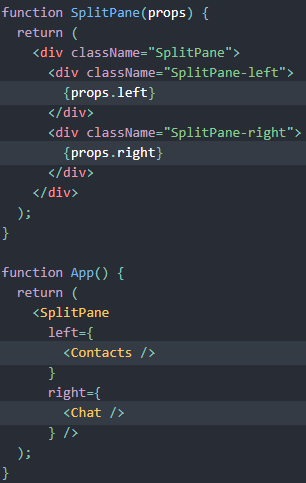


Это позволит передать компоненту произвольные дочерние элементы, вложив их в JSX:



Всё, что находится внутри JSX-тега <FancyBorder>, передаётся в компонент FancyBorder через проп children. Поскольку FancyBorder рендерит {props.children} внутри <div>, все переданные элементы отображаются в конечном выводе.

Иногда в компоненте необходимо иметь **несколько мест для вставки**. В таком случае можно придумать свой формат, а не использовать children:

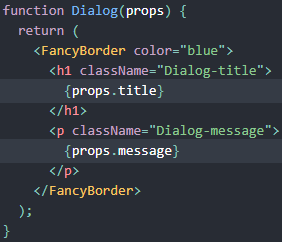


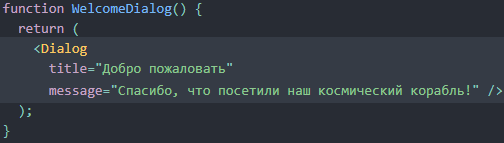
Такие React-элементы, как <Contacts /> и <Chat /> являются просто объектами, **поэтому их можно передать в виде пропсов**, как и любые другие данные.

### Специализация

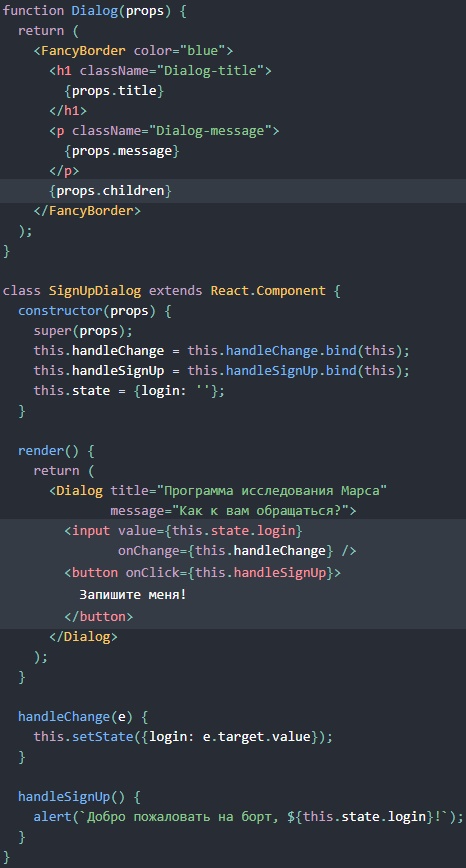
Некоторые компоненты можно рассматривать как «частные случаи» других компонентов. Например, WelcomeDialog может быть частным случаем Dialog.

В React это можно сделать через композицию, где «частный» вариант компонента рендерит более «общий» и настраивает его с помощью пропсов:





Композиция хорошо работает и для компонентов, определённых через классы:



## Ф***илософия React***

<https://ru.reactjs.org/docs/thinking-in-react.html>

Пример написан тобой в репозитории

# Продвинутые темы

## Доступность контента

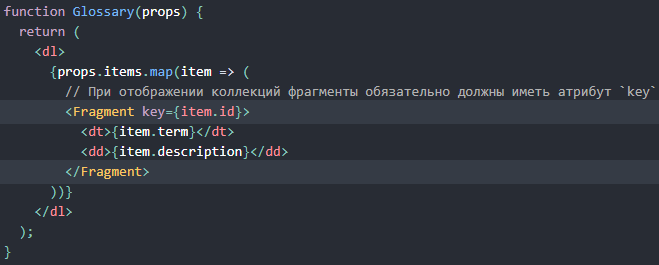
### Семантическая вёрстка (Fragment)

Семантическая вёрстка — это основа доступности контента в веб-приложениях. Используя различные HTML-элементы можно улучшить восприимчивость и понятность ваших сайтов. Это позволяет сделать контент доступным без особых усилий.

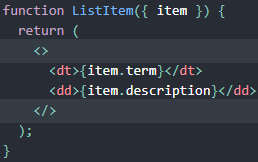
Бывают случаи, когда семантическая вёрстка нарушается. Например, при добавлении элемента <div> в JSX для обеспечения работоспособности кода на React. Особенно часто это случается при работе со списками (<ol>, <ul>, <dl>) или таблицами (<table>). В такой ситуации рекомендуется использовать [фрагменты](https://ru.reactjs.org/docs/fragments.html), чтобы сгруппировать элементы, как это показано в примере:



**Коллекцию элементов** можно преобразовать в массив фрагментов или любых других элементов:



Если нет необходимости использовать пропсы, то можно применять [сокращённую запись фрагментов](https://ru.reactjs.org/docs/fragments.html#short-syntax):



### Доступность контента в формах (label)

Каждый элемент управления, например, <input> или <textarea>, должен иметь подпись, обеспечивающую доступность контента. Подписи нужно выполнять так, чтобы их могли использовать экранные считывающие устройства и другие технические средства реабилитации.

Эти стандартные для HTML способы могут использоваться в React напрямую, однако не забывайте, что атрибут for в JSX записывается как **htmlFor**:



### Управление фокусом

Приложение с доступным контентом должно функционировать при использовании только клавиатуры.

* **Фокус клавиатуры и контур элемента**

Фокус клавиатуры указывает на элемент в структуре DOM, который в данный момент готов принимать ввод с клавиатуры. Обычно такой элемент выделяется контуром:

****

Если вы заменяете стандартную реализацию фокуса своей, удалить контуры элементов можно с помощью CSS, установив outline: 0.

* **Механизмы быстрого перехода к нужному контенту**

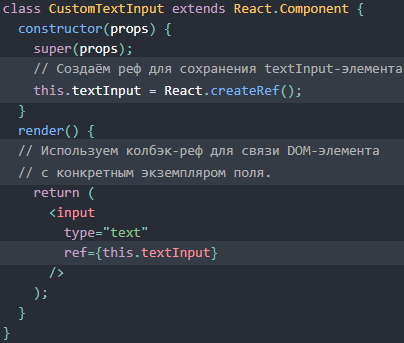
Также на сайте нужно реализовать механизмы, которые помогают пользователям быстро переходить к нужному контенту с помощью клавиатуры.

Элементы семантической вёрстки, например, <main> или <aside>, нужно использовать в качестве секционной разметки, предназначенной для быстрого перехода между логическими частями сайта.

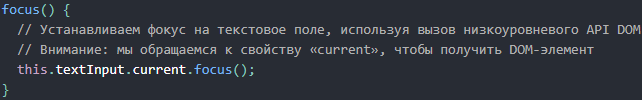
* **Программное управление фокусом**

React-приложения во время своей работы постоянно изменяют структуру DOM. Иногда из-за этого фокус клавиатуры может быть потерян или может перейти на неправильный элемент. Чтобы исправить такую ситуацию, нужно запрограммировать перевод фокуса клавиатуры на нужный элемент. Например, после закрытия модального окна перевести фокус клавиатуры на кнопку, которая его открыла.

Чтобы управлять фокусом в React, можно использовать [рефы на DOM-элементы](https://ru.reactjs.org/docs/refs-and-the-dom.html). При таком подходе мы сначала создаём в классе компонента реф на элемент в JSX:



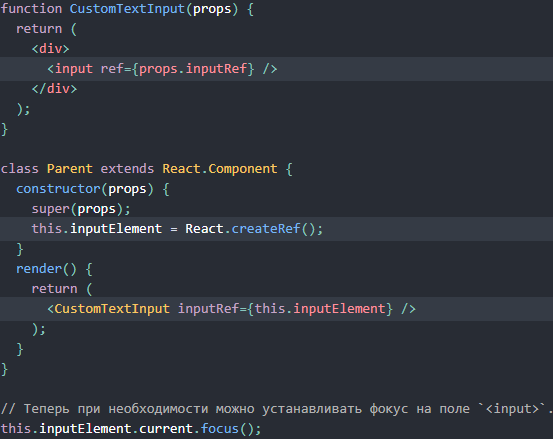
Теперь при необходимости можно установить фокус на этот элемент из любого места компонента:



**Иногда родительскому компоненту нужно установить фокус на элемент дочернего компонента.**

Сделать это можно с помощью [рефа на родительский компонент](https://ru.reactjs.org/docs/refs-and-the-dom.html#exposing-dom-refs-to-parent-components), который присваивается специальному свойству дочернего компонента и используется для ссылки из родительского компонента на DOM-элемент дочернего.

То есть создали пустой реф в родителе и привязали его к инпуту уже в дочерней компоненте с помощью прокинутого в нее пропса с пустым рефом:

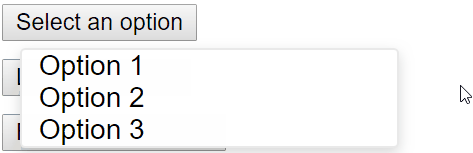
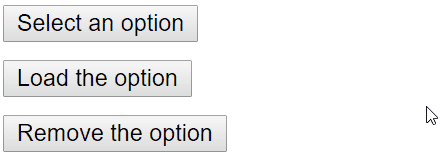


Если для расширения функциональности компонент оборачивается компонентом высшего порядка, то рекомендуется [**перенаправлять рефы**](https://ru.reactjs.org/docs/forwarding-refs.html) обёрнутого компонента с помощью React-функции **forwardRef**. В случае, когда сторонний HOC не поддерживает перенаправление рефов, **описанный выше подход** всё равно можно использовать в качестве запасного варианта.

### Работа с событиями мыши

Позаботьтесь, чтобы вся функциональность, связанная с событиями мыши, была доступна при работе только с клавиатурой. Если приложение сильно зависит от мыши, многие пользователи, которые используют только клавиатуру, не смогут работать с ним.

Пример:

до: после: 

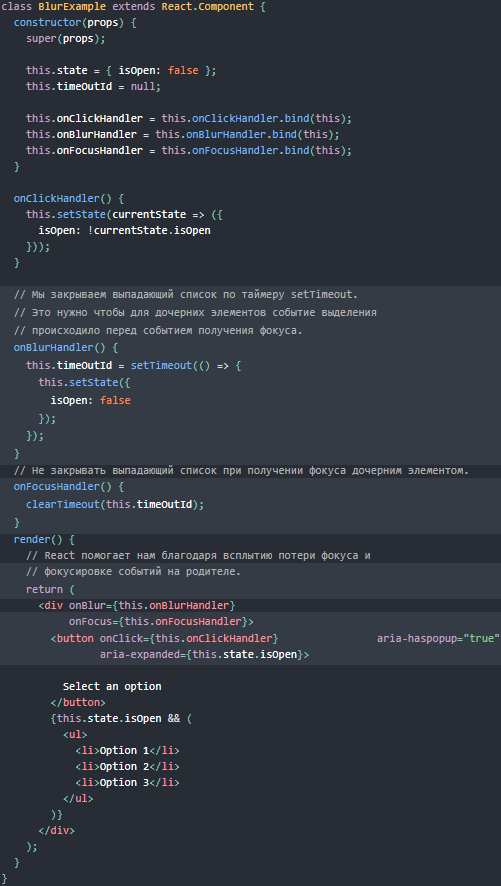
Тут показан паттерн закрытия всплывающего списка при щелчке мышью за пределами этого элемента.

Обычно такая функциональность реализуется событием click объекта window, обработчик которого закрывает выпадающий список:



Для пользователей, работающих только с клавиатурой, такая реализация будет причиной нарушения работы программы при попытке перехода к следующему элементу с помощью клавиши Tab. Потому что объект window никогда не получит событие click. В итоге мы имеем заблокированную функциональность и невозможность продолжения работы с приложением.

Это можно исправить используя обработчики сопутствующих событий, например, onBlur или onFocus:

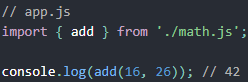
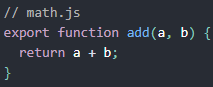


## Разделение кода

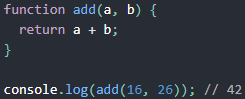
### Бандлинг

Большинство React-приложений «собирают» свои файлы такими инструментами, как [Webpack](https://webpack.js.org/), [Rollup](https://rollupjs.org/) или [Browserify](http://browserify.org/). Сборка (или «бандлинг») — это процесс выявления импортированных файлов и объединения их в один «собранный» файл (часто называемый «bundle» или «бандл»). Этот бандл после подключения на веб-страницу загружает всё приложение за один раз. Пример:

**Приложение:**

**Бандл:**



**Примечание:**

Ваши бандлы будут выглядеть не так, как мы только что показали.

Если вы используете [Create React App](https://create-react-app.dev/), [Next.js](https://github.com/zeit/next.js/), [Gatsby](https://www.gatsbyjs.org/) и т.п., то у вас уже будет настроенный Webpack для бандлинга приложения.

### Разделение кода

Бандлинг — это хорошо, но по мере роста приложения, бандл тоже будет расти. Особенно если подключены крупные сторонние библиотеки. Нужно следить за кодом, который вы подключаете, чтобы не сделать приложение настолько большим, что его загрузка займёт слишком много времени.

Чтобы предотвратить разрастание бандла, стоит начать «разделять» его. Разделение кода — это возможность, поддерживаемая такими бандлерами как [Webpack](https://webpack.js.org/guides/code-splitting/), [Rollup](https://rollupjs.org/guide/en/#code-splitting) или Browserify (с [factor-bundle](https://github.com/browserify/factor-bundle)), которая может создавать несколько бандлов и загружать их по мере необходимости.

### Динамический импорт

Это лучший способ внедрить разделение кода в приложение:

**До:**

****

**После:**

****

### React.lazy